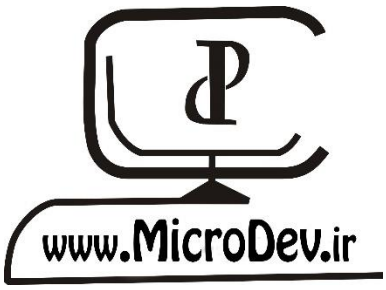
ASP.NET Core 3.1

برای بازار کار

مولفین:
زهرابیات قلی لاله
علی بیات قلی لاله



ASP.NET Core

برای بازار کار

زهرا بیات قلی‌لاله

علی بیات قلی‌لاله

ASP.NET Core

برای بازار کار

مولفین : زهرا بیات قلی‌لاله – علی بیات قلی‌لاله

طراح جلد : زهرا بیات قلی‌لاله

مشخصات ظاهری : ۲۷۰ص

سال انتشار: بهمن ۹۸

قیمت : رایگان

فهرست

خلاصه فصل

فصل دوم : Middleware و مدیریت خطاها

Middleware چیست؟

نحوه تعریف Middleware در Pipeline

Exception Handling چیست؟

مدیریت خطا با استفاده از DeveloperExceptionPageMiddleware و ExceptionHandlerMiddleware

Hosting Environment چیست؟

تنظیم محیط میزبانی

ایجاد صفحه سفارشی

مدیریت خطاها با استفاده از StatusCodePagesMiddleware

تمرین

Interview Questions

Quiz

Answers

خلاصه فصل

فصل سوم : MVC Design Pattern

MVC چیست؟

اجزای MVC

سیستم MVC چطور کار می‌کند؟

مزایای MVC Design Pattern

پیاده‌سازی MVC در وب اپلیکیشن ASP.NET Core

Modelچیست؟

Validation Attributeها

ایجاد Model

Controller و اکشن‌متد چیست؟

تمرین

Interview Questions

Quiz

Answers

خلاصه فصل

فصل چهارم: سیستم Routing

Routing چیست؟

مزایای سیستم Routing

سیستم Routing چگونه کار می‌کند؟

قسمت‌های یک الگوی مسیر

روش‌های Mapping

Conventional Routing

Attribute Routing

برنامه MVC با چندین مسیر

Constraint بر روی مسیرها

چطورConstraintها را اعمال نماییم؟

تمرین

Interview Questions

Quiz

Answers

خلاصه فصل

فصل پنجم: رندر کردن HTML با استفاده از Razor view

View چیست؟

Razor چیست؟

چرا یادگیری Razor مهم است؟

نحوه استفاده از Razor

روشهای انتقال داده به View

انتقال داده با View Model

ایجاد View Model

انتقال داده با استفاده از ViewData

انتقال داده با استفاده از ViewBag

نوشتن عبارات با سینتکس Razor

✓ متغیرها

- ✓ عبارات شرطی
- ✓حلقه‌ها
- ✓ بلوک های کد
- ✓کامنت

Layout چیست؟

مزایای Layout

چطور از Layout استفاده کنیم؟

Section چیست؟

Partial view چیست؟

ایجاد یک Partial View

استفاده از Partial View

استفاده از Partial Viewهای Strongly Type

ViewStart چیست؟

ایجاد فایل ViewStart

ViewImports چیست؟

ایجاد فایل ViewImports

تمرین

Interview Questions

Quiz

Answer

خلاصه فصل

فصل ششم: Tag Helperها چیست؟

Tag Helperچیست؟

فعال کردن Tag Helper در اپلیکیشن

استفاده از Tag Helperها

Environment Tag Helper

Script Tag Helper وLink Tag Helper

Form Tag Helper

Label Tag Helper

ایجاد یک Tag Helper سفارشی

تمرین

Interview Questions

Quiz

Answer

خلاصه فصل

فصل هفتم: تزریق وابستگی چیست؟

DI چیست؟

اهداف و مزایای DI چیست؟

تزریق وابستگی در ASP.NET Core

استفاده از تزریق وابستگی

مراحل ایجاد کدهای Loosely coupled

طول عمر سرویس چیست؟

پیادهسازیهای مختلف از یک سرویس

تمرین

Interview Questions

Quiz

Answer

خلاصه فصل

فصل هشتم: ایجاد WebAPI در ASP.NET Core

Web API چیست و چه زمانی باید از آن استفاده کنید؟

REST چیست و HTTP چگونه کار میکند؟

Controller و اکشن‌متدها

ایجاد اولین اپلیکیشن Web API

افزودن Domain Modelها

Entity Framework Core وDapper

رجیستر DbContext

Data Seeding چیست؟

ایجاد و آپدیت دیتابیس با Migration

پیاده‌سازی Command

استفاده از Authorization در اپلیکیشن

افزودن فرم لاگین

Seeding داده‌های کاربر

Interview Questions

Quiz

Answer

خلاصه فصل

تقدیم به

تقدیم به تمام دوستداران برنامه‌نویسی که آماده‌ی استفاده از تمام قابلیت‌های خود برای یادگیری هستند.

با تشکر

از جناب آقای مهندس فرهاد افتخاری مدیر عامل شرکت TechClass، که در به سرانجام رساندن این کتاب همراه ما بودند، بسیار سپاسگزاریم.

در باره این کتاب

این کتاب برای برنامه‌نویسان سی‌شارپ که علاقمند به یادگیری سریع ASP.NET Core برای ورود به بازار کار هستند نوشته شده است.

ASP.NET Core یک فریم ورک قدرتمند است که با پیشرفت‌های چشمگیر خود به انتخاب اول بسیاری از شرکت‌های نرم‌افزاری تبدیل شده است و هم اکنون یکی از مدعی های برتر در معیارهای مختلف TechEmpower[1] است.

شما با مطالعه این کتاب، به سرعت با تمام چیزهایی که برای ایجاد یک برنامه کاربردی ASP.NET Core نیاز است آشنا می‌شوید و می‌توانید برای ورود به بازار کار آماده شوید.

1 Techempower نام سایتی است که فریمورک های مختلف را بررسی می‌کند و نتایج مقایسه بین فرمورک‌های مختلف را در قالب نمودار نشان می دهد.

پیش نیازها

برای مطالعه این کتاب شما باید پیش نیازهای زیر را داشته باشید:

۱) **دانستن اینکه وب اپلیکیشن چیست؟**
۲) **داشتن یک تجربه اولیه از ایجاد یک وب اپلیکیشن با سی‌شارپ.**

نحوه خواندن این کتاب

هر فصل این کتاب شامل ۵ بخش است، لطفا برای خواندن این کتاب مراحل زیر را دنبال کنید:

۱) **محتوای هر فصل:** این بخش شامل شرح فصل می‌باشد.
۲) **تمرین:** در پایان هر فصل، سوالاتی در مورد فصل جدید پرسیده می‌شود که تحقیق درباره آن و پاسخ دادن به آن‌ها، می‌تواند یک شروع خوب در یادگیری فصل بعدی باشد.
۳) **پرسش‌های مصاحبه‌ای:** این بخش شامل سوالاتی است که در مصاحبه شغلی از شما پرسیده می‌شود.
۴) **آزمون:** بخش آزمون شامل سوالات ۴ گزینه‌ای است که شما می‌توانید در این قسمت یادگیری خود را محک بزنید. **یک خبر خوب!! در پایان آزمون پاسخ سوالات مشخص شده است.**
۵) **خلاصه فصل:** این بخش شامل چکیده فصل می‌باشد.

ابزارهای مورد نیاز

تنها ابزاری که برای یادگیری به آن نیازی دارید Visual Studio 2019 است.

نکته!!

هنگام نصب Visual Studio، اطمینان حاصل کنید که کامپوننت‌های ASP.NET و NET Core . را انتخاب کرده‌اید.

فصل اول : چرا ASP.NET Core؟

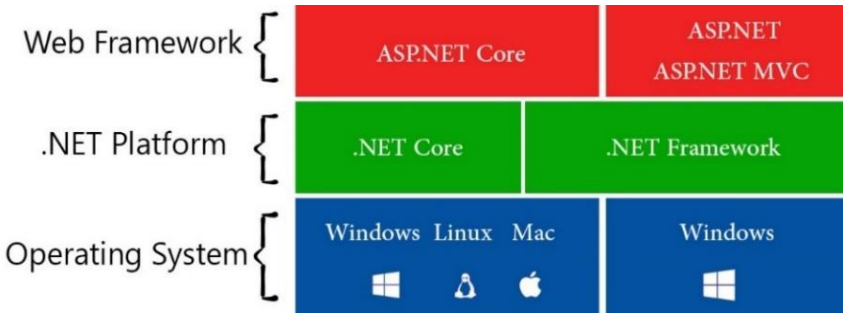
آنچه خواهید آموخت:

- ASP.NET Core چیست؟
- چرا ASP.NET Core را انتخاب کنیم؟
- ایجاد اولین اپلیکیشن ASP.NET Core
- اجرای وب اپلیکیشن
- درک ساختار پروژه

ASP.NET Core چیست؟

ASP.NET Core با اصول طراحی نرم‌افزار مدرن، بر روی پلت فرم جدید NET Core. ایجاد شده است. که در آن فقط کامپوننت‌هایی که نیاز دارید را اســتفاده می‌کنید و می‌توانید برنامه خود را تا جایی که امکان دارد جمع جور و با عملکرد بالا بسازید.

ASP.NET Core این امکان را به شما می‌دهد که وب‌اپلیکیشن‌های خود را در ویندوز، لینوکس و Mac ایجاد و اجرا نمایید.



چرا ASP.NET Core را انتخاب کنیم؟

ASP.NET Core یک وب‌فریم‌ورک قدرتمند است که از آن می‌توان برای ایجاد سریع‌تر، راحت‌تر و امن‌تر انواع اپلیکیشــن‌ها اســتفاده نمود. این وب‌فریم‌ورک پر از ویژگی‌های جالب اســت که در اینجا می‌خواهم چند نمونه از مهمترین آن‌ها را بیان کنم:

- **ASP.NET Core یک فریم‌ورک مدرن، Open Source , Scalable و با Performance بالا است.**
- **ASP.NET Core یک معماری ماژولار برای Maintenance راحت‌تر دارد.**
- **ASP.NET Core مدیریت Cross-Site Request Forgery (CSRF) را برعهده می‌گیرد.**
- **و می تواند روی هر پلت فرمی اجرا شود.**

اما این فریم‌ورک جذاب، برخی پیشرفت‌های زیربنایی هم داشته که در ادامه چند نمونه ذکر شده است:

- **Middleware Pipeline برای تعریف رفتارهای اپلیکیشن.**
- **داشتن یک Dependency Injection توکار.**
- **ترکیب زیرساخت UI (MVC) و API (Web API) .**
- **سیستم پیکربندی بسیار گسترده.**

- **قابل Scalable بودن برای پلت‌فرم‌های Cloud (با استفاده از برنامه‌نویسی غیرهمزمان)**

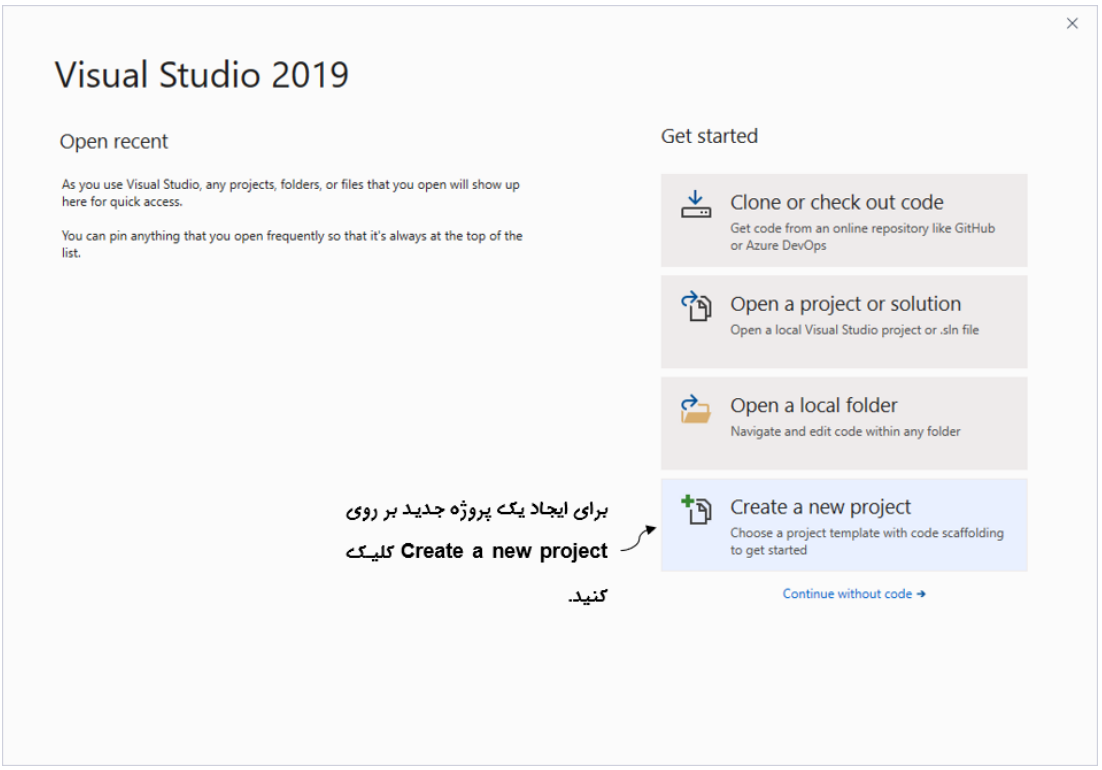
از اینکه بعضی مفاهیم ناآشناست نگران نباشید، خیلی زود با این مباحث آشنا خواهید شد.

ایجاد اولین برنامه ASP.NET Core

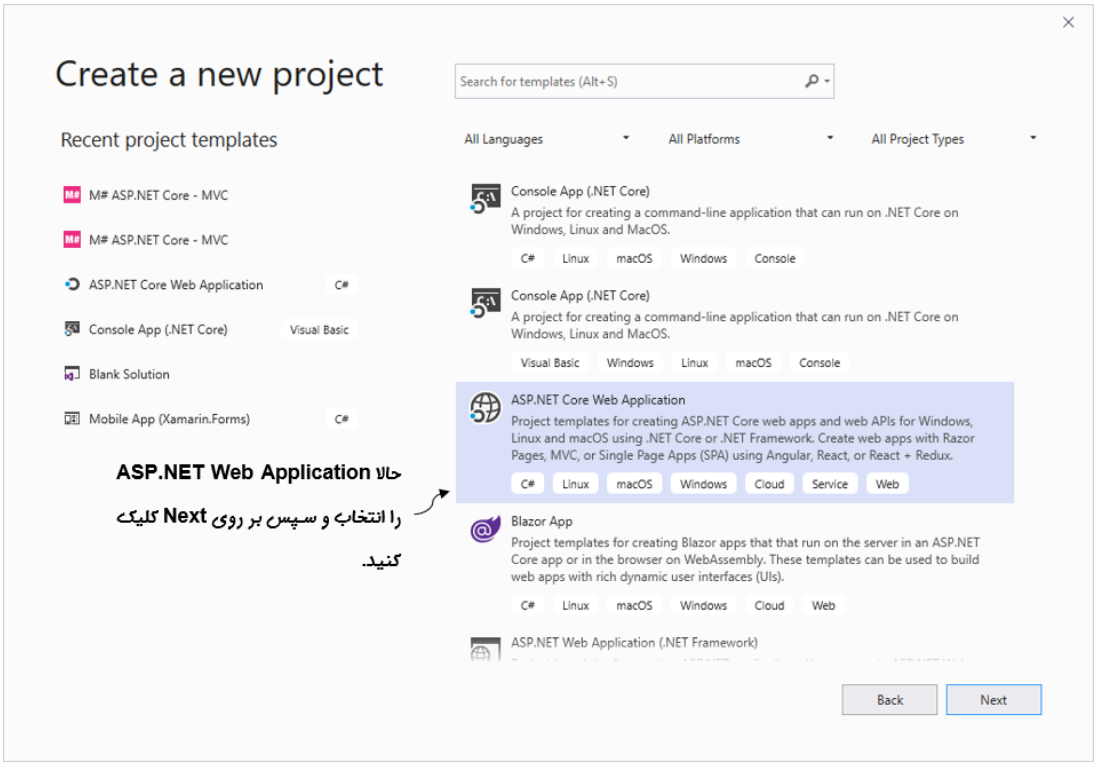
تا اینجا همه چیز عالی بود، بیایید با هم اولین وب اپلیکیشن خود را ایجاد کنیم.

برای درک بهتر مفاهیم می‌خواهم با ویژوال استودیو ۲۰۱۹ یک اپلیکیشن Empty از ASP.NET Core ایجاد کنم.

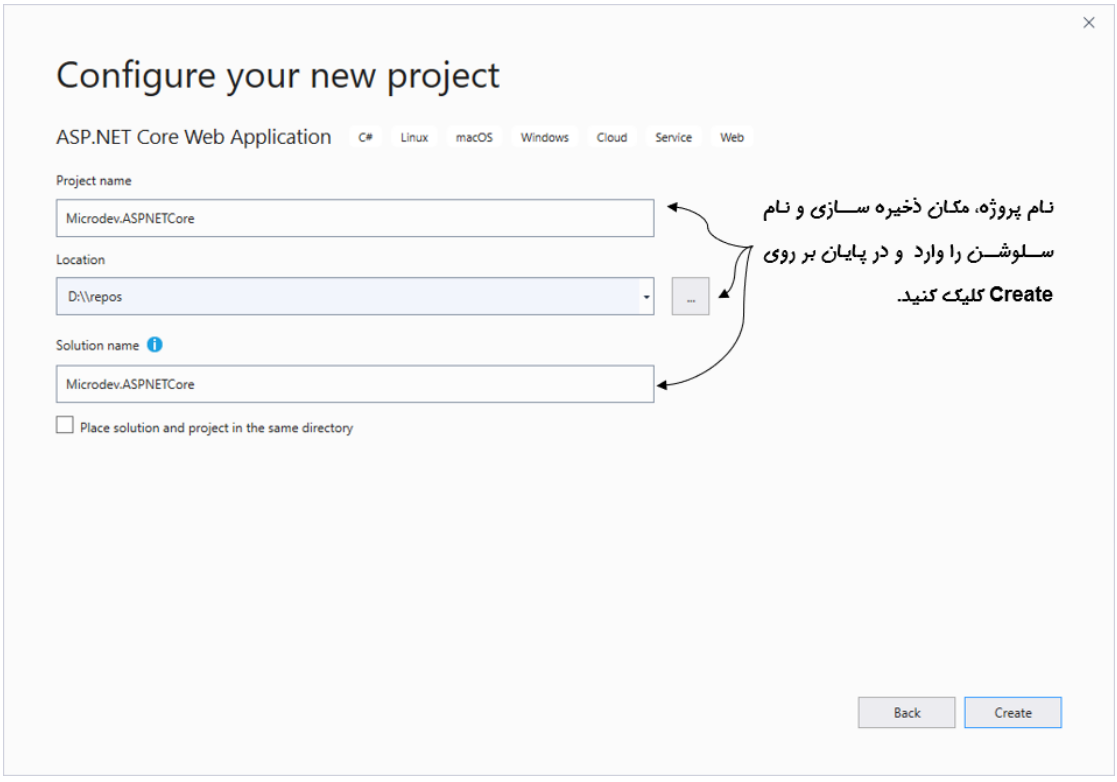
- **ویژوال استدیو ۲۰۱۹ را باز کنید و بر روی Create a new project کلیک کنید.**



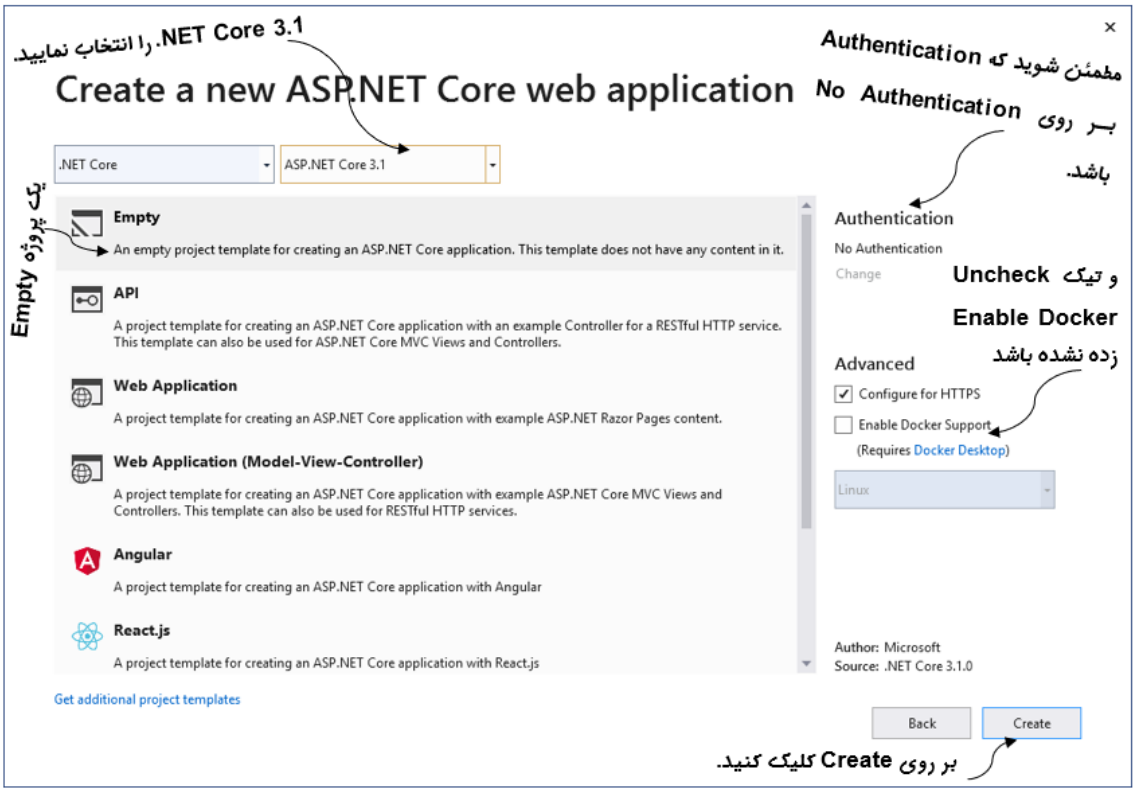
- **در کادر بعدی ASP.NET Core Web Application را انتخاب و بر روی Next کلیک کنید.**



- حالا نام پروژه را Microdev.ASPNETCore بگذارید، سپس مکان ذخیره‌سازی و نام Solution را همانند تصویر وارد و بر روی Create کلیک نمایید.



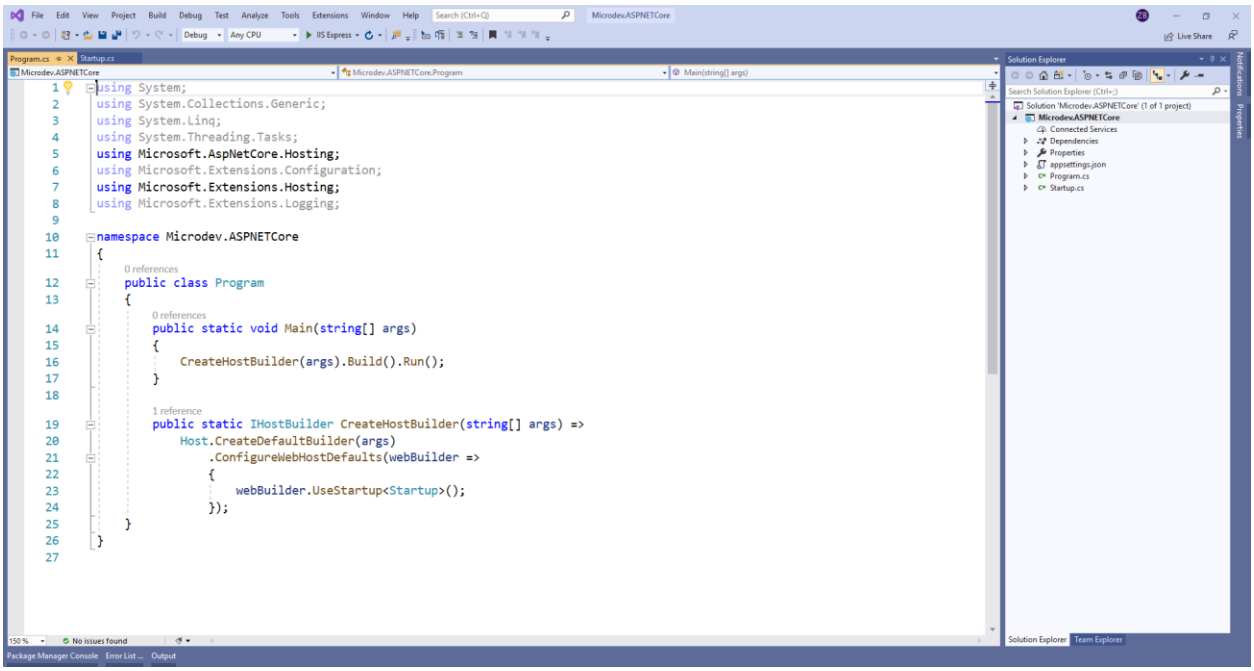
- در کادر بعدی Empty و Asp Net Core 3.1 را انتخاب کنید.



نکته!!

مطمئن شوید که Enable Docker Support تیک نخورده و اپلیکیشن بر روی No Authentication تنظیم شده است.

منتظر بمانید تا Visual Studio اپلیکیشن شما را ایجاد نماید.



هورا!!! اولین وب اپلیکیشن ایجاد شد.

اجرای وب اپلیکیشن

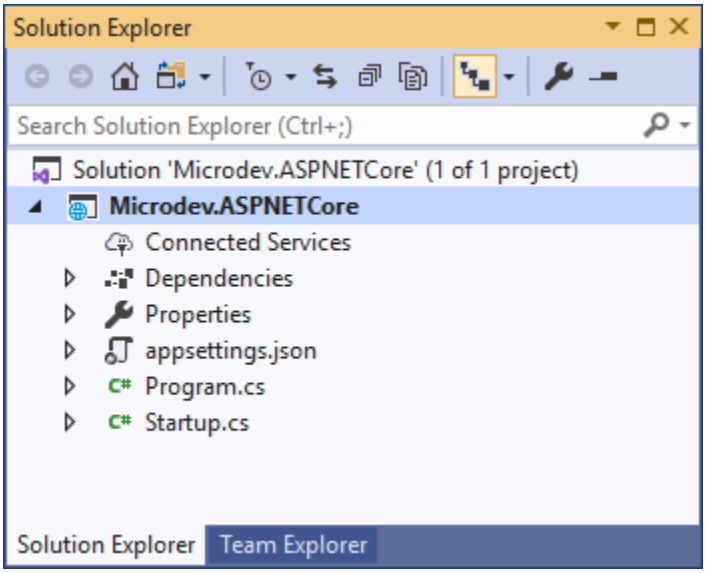
تازه بازی شـروع شـد، بیایید با هم برنامه را اجرا کنیم. لطفاپروژه را Build و سـپس F5 را بزنید. (یا اینکه بر روی پیکان سبز رنگ در نوار ابزار کنار IIS Express کلیک نمایید)

مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter1/Sample1

درک ساختار پروژه

بیایید به عناصر درون Solution نگاه بیندازیم:



- **Dependencies و Connected Services**: این دو قسمت برای نمایش تمام وابستگی‌های NuGet packages ، وابستگی‌های Client-Side و سرویس‌های از را دور وابسته به پروژه می‌باشد.
- **Properties**: فولدر Properties تنها یک فایل launchSettings.json برای کنترل نحوه اجرای Visual Studio و Debug اپلیکیشن دارد.
- **Program.cs و Startup.cs**: این دو فایل هم، برای راه‌اندازی وب سرور و Pipeline استفاده می‌شود.

کلاس Program

کلاس Program مسئول پیکربندی بسیاری از زیرساخت‌های اپلیکیشن شماست و تمامی اپلیکیشن‌های ASP.NET Core با این فایل شروع می‌شوند.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
```

متد Main برای ایجاد و پیکربندی یک آبجکت builder، متد CreateHostBuilder را فراخوانی و سپس متد Build و Run آن را صدا می زند.

کلاس Startup پیکربندی اپلیکیشن شما را تعریف می کند.

```
            {
                webBuilder.UseStartup<Startup>();
            });
    }
```

متد Main نقطه شـــروع برنامه اســـت که در آن بســ یاری از پیکربندی‌های برنامه انجام می‌شـــود. هنگامی که اپلیکیشن شما Start می شود، این متد باید یک Host را پیکربندی و اجرا کند، که این Host وظیفه مدیریت Startupو Lifetime اپلیکیشـــن را به عهده دارد و حداقل باید پیکربندی ســـرور و Request processing pipeline را انجام دهد.

شـاید بپرسـید Host چیسـت؟ برای کنترل شـروع و پایان اپلیکیشـن، مدیریت چرخه عمر اپلیکیشـن و تمام منـابع وابســـتـه بـه آن(مثـل Dependency injection، Logging، Configuration) پیـاده‌ســـازی‌هـای IHostedService را درون یک object به نام Host کپسوله می کنیم.

پیکربندی، ساخت و اجرای Host معمولا توسط کدی که در کلاس Program نوشته شده انجام می‌شود. ابتدا متد Main، متدی با نام CreateHostBuilder را جهت پیکربندی و ساخت شی Builder فراخوانی و در پایان متد Build و Run شی Builder را صدا می‌زند.

نکته!!

اگر از Entity Framework Core اسـتفاده می‌کنید، باید Signature متد CreateHostBuilder را تغییرندهید، زیرا Entity Framework Core tools برای اینکه بتولندبدون اجرای اپلیکیشـــن، پیکربندی Host را انجام دهد، از متد CreateHostBuilder استفاده می‌کند.

نکته!!

کلاس Startup که در متد جنریک <>UseStartup رفرنس داده شـــده، جاییســـت که پیکربندی سرویس‌ها و تعریف Middleware Pipeline انجام می‌شود.

کلاس Startup

اپلیکیشـن‌های ASP.NET Core برای پیکربندی برخی رفتارهای اپلیکیشـن از یک کلاس Startup اسـتفاده می‌کنند. این کلاس از هیچ کلاس پایه‌ای ارث‌بری نمی‌کند و هیچ اینترفیســی را هم پیاده‌ســازی نخواهد کرد و کار پیکربندی سرویس و تعیین Middleware Pipeline را انجام می دهد.

کلاس Startup برای بارگذاری و پیکربندی کامپوننت‌های اپلیکیشن، شامل دو متد ConfigureServices و Configure می‌باشد.

نکته!!

HostBuilder ایجاد شده در کلاس Program ابتدا متد ConfigureServices و سپس Configure را صدا می زند، بنابراین تمام سرویس های رجیستر شده در متد ConfigureServices در متد Configure قابل دسترسی هستند.

نکته!!

سرویس، اشاره به کلاسی دارد که بتواند یک Functionality را در اپلیکیشن فراهم کند.

متد ConfigureServices

متد ConfigureServices یک متد اختیاریست، که رجیستر کردن سرویس‌های اپلیکیشن با استفاده از سیستم تزریق وابستگی (Dependency Injection) ASP.NET Core درون این متد انجام می شود. بنابراین این متد جایی است که تزریق وابستگی در آن پیکربندی می شود.

تزریق وابستگی چیست؟ تزریق وابستگی یک تکنیک بسیار مهم و کاربردیست که باعث می‌شود Instanceهای یک کلاس در زمان اجرا، ایجاد شده و شما به راحتی کدهای Tastable و Loosely Coupled بنویسید.

برای مثال: فرض کنید شما یک کلاس EmpolyeeService و یک کنترلر EmpolyeeController دارید، که این کنترلر در زمان اجرا نیاز به یک Instanceای از EmpolyeeService دارد.

رویکرد کلی این است که هر زمان که نیاز به یک سرویس داشتید از کلمه کلیدی new استفاده کنید و یک Instance از آن سرویس ایجاد نمایید. متاسفانه، مشکل این رویکرد این است که کد شما به پیاده‌سازی خاصی وابسته (**Tightly couple**) می‌شود و نگهداری[۲] و تست اپلیکیشن سخت خواهد شد.

[۲] Maintain

Tightly couple چیست؟ **Tightly couple** یعنی یک کلاس به پیاده سازی کلاس دیگر وابسته باشد، در این روش تغییر کلاس کمکی بدون دست زدن به کلاس اصلی غیر ممکن خواهد بود. چون در کلاس اصلی مستقیما به کلاس کمکی رفرنس داده شده است.

اما راه حل این مشکل: بعد از نوشتن سرویس، شما می‌توانید وابستگی‌های خود را اعلام کرده و اجازه دهید تا کلاسی دیگر این وابستگی‌ها را برای شما فراهم کند. این تکنیک با نام تزریق وابستگی[۳] یا معکوس شدن کنترل (IOC[4]) شناخته می‌شود.

در برنامه های ASP.NET Core، رجیستر شدن سرویس ها در متد ConfigureServices انجام می‌شود و هر زمان که از یک ویژگی جدید ASP.NET Core در اپلیکیشن‌تان استفاده کنید، باید به این متد برگردید و سرویس‌های موردنیاز را اضافه نمایید.

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
   {
     services.AddScoped<EmpolyeeService>();
   }
}
```

رجیستر شدن سرویس با استفاده از اینترفیس IServiceCollection انجام می‌شود.

هر زمان که‌به سرویس EmpolyeeService نیاز داشته باشید، این سرویس ایجاد خواهد شد.

پارامتر IServiceCollection (در ورودی این متد) شامل لیستی از سرویس‌های موردنیاز اپلیکیشن است.

AddScoped چیست؟ AddScoped متدیست که طول عمر یک سرویس را مشخص می‌کند. این طول عمر بدین صورت است که با هر بار درخواست وب، یک Instance جدید از کلاس EmpolyeeService ایجاد خواهد شد.

نکته!!

اگر فراموش کنید یک سرویس را رجیستر نمایید، در زمان اجرا یک InvalidOperationException دریافت خواهید کرد.

---

[۳] Dependency Injection

[4] Inversion Of Control (IOC) Principle

متد Configure

Configure متدیست که با استفاده از اکستنشن متدهای موجود بر روی IApplicationBuilder ماژول‌هایی را به Request Pipeline اضافه و رفتارهای اپلیکیشن را تعریف می‌نماید. به این ماژول ها Middleware گفته می شود.

IApplicationBuilder برای ایجاد Middleware استفاده می شود. Pipeline

IWebHostEnvironment شیی است که جزییات محیط جاری را در خود نگه می‌دارد.

زمانیکه در محیط Development هستید رفتار متفاوت از اپلیکیشن روی می دهد.

متد app.UseEndpoints یک Middleware داخلی است.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });

}
```

همانطورکه می‌بینید این متد معمولا دو پارامتر می‌پذیرد: **IApplicationBuilder** و **IWebHostEnvironment**.

- **پارامتر IApplicationBuilder، ترتیب اجرای Middlewareها را تعریف می‌کند، بنابراین Middleware تنها با این اینترفیس به Pipeline اپلیکیشن اضافه خواهد شد. برای مثال UseDeveloperExceptionPage()، اکستنشن متدی بر روی آرگومان IApplicationBuilder است.**
- **پارامتر IWebHostEnvironment شی‌ست، که اطلاعاتی در مورد محیط جاری را در خود نگه می‌دارد، بنابراین از آن می‌توان برای ارائه رفتارهای متفاوت در محیط‌های توسعه استفاده نمود.**

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Middleware چیست؟**
- ✓ **مدیریت خطاها در ASP.NET Core چطور انجام می‌شود؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What is ASP.NET Core?

Q2: What are some characteristics of ASP.NET Core?

Q3: What is the difference between ASP.NET and ASP.NET MVC?

Q4: What are the benefits of ASP.NET Core?

Q5: How to configure your ASP.NET Core app?

Q6: Explain startup process in ASP.NET Core?

Q7: What is the file extension of ASP.NET web service?

Q8: Explain usage of Dependency Injection in ASP.NET Core?

Q9: What is Kestrel?

Q10: What does Host do?

Quiz

Q1: ASP.NET Core is an _____ framework.

1. Licensed
2. Open-sourced
3. Obsolete
4. UI

Q2: ASP.NET Core supports which of the following platforms?

1. Windows
2. Linux
3. Mac
4. All of the above

Q3: ASP.NET Core which does not have the following features?

1. No scalable
2. Open source
3. High-performance
4. All of the above

Q4: Which of the following is an entry point of ASP.NET Core application?

1. Main method of Program class
2. Configure method of Startup class
3. ConfigureService method of Startup class
4. Application_start method of Global.asax

Q5: By default, static files can be served from folder.

1. bin
2. wwwroot
3. Any folder under the root folder
4. StaticFiles

Q6: The host for ASP.NET Core web application is configured in file.

1. Program.cs
2. Startup.cs
3. Middleware
4. None of the above

Q7: The Startup class must include _____ method.

1. ConfigureService
2. Main
3. BuildWebHost
4. Configure

Q8: The method in Startup class is used to registering services with IoC container.

1. ConfigureService
2. Configure
3. Main
4. All of the above

Q9: ASP.NET Core web application uses as an internal web server by default.

1. IIS
2. Apache
3. Kestrel
4. nginx

Q10: What is IWebHostEnvironment?

1. IWebHostEnvironment is object contains details about the current environment.
2. IWebHostEnvironment is used to define the order in which middleware executes.
3. IWebHostEnvironment is where you can configure some of your app’s behavior.
4. All of the above

Answers

1-Correct Answer: Open-sourced

2-Correct Answer: All of the above

3-Correct Answer: no scalable

4-Correct Answer: Main method of Program class

5-Correct Answer: wwwroot

6-Correct Answer: Program.cs

7-Correct Answer: Configure

8-Correct Answer: ConfigureService

9-Correct Answer: Kestrel

10-Correct Answer: IWebHostEnvironment is object contains details about the current environment.

خلاصه فصل

- ✓ **ASP.NET Core تکنولوژی ای است که با اصول طراحی نرم‌افزار مدرن، بر روی پلت فرم جدید .NET Core ایجاد شده است. بنابراین شما می‌توانید وب‌اپلیکیشن‌های خود را در ویندوز، لینوکس یا MacOS بسازید و اجرا کنید.**
- ✓ **مهمترین ویژگی‌های ASP.NET Core عبارتند از: Modern, High-Performance ،Open Source ،Scalable.**
- ✓ **تمام اپلیکیشن‌های ASP.NET Core با یک فایل Program.cs شروع می‌شوند که این کلاس مسئول پیکربندی بسیاری از زیرساخت‌های اپلیکیشن شماست.**
- ✓ **کلاس Startup جاییست که می‌توانید برخی رفتارهای اپلیکیشن را پیکربندی کنید. Startup.cs دو متد ConfigureServices و Configure برای بارگذاری و پیکربندی کامپوننت‌ها دارد.**
- ✓ **متد ConfigureServices جهت رجیستر کردن سرویس‌های اپلیکیشن با استفاده از سیستم تزریق وابستگی ASP.NET Core است.**
- ✓ **متد Configure جهت تعریف Middleware Pipeline برای اپلیکیشن است و شما می‌توانید با این متد ماژول‌ها را به Pipeline اضافه نمایید.**

فصل دوم : Middleware و مدیریت خطاها

آنچه خواهید آموخت:

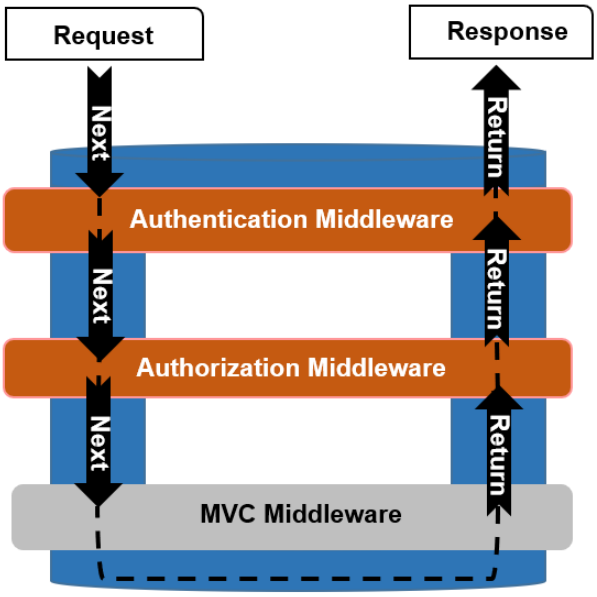
- Middleware چیست؟
- نحوه تعریف Middleware در Pipeline
- Exception Handling چیست؟
- Hosting Environment چیست؟
- تنظیم محیط میزبانی
- مدیریت خطاها

Middleware چیست؟

Middleware یک کلاس است که می‌تواند یک HTTP Request یا HTTP Response را مدیریت کند.

زمانیکه اپلیکیشن یک Request دریافت می‌کند، Middlewareها به صورت پیوسته اجرا می‌شوند. در این زمان در Pipeline، هر Middleware می‌تواند قبل از ارسال درخواست به Middleware بعدی یک درخواست ورودی را تغییر یا مدیریت کند. بنابراین قبل از ارسال Response به کاربر، هر Middleware امکان بازبینی یا ویرایش آن را دارد.

نتیجه‌گیری: Middlewareها به شما این امکان را می‌دهند تا با کامپوننت‌های کوچک و متمرکز، رفتارهای یک اپلیکیشن پیچیده را مدیریت کنید.



همانطور که در تصویر بالا می‌بینید، یکی از نکات مهم در شکل‌گیری Pipeline این است که، Pipeline دوطرفه است، بنابراین Request از یک جهت Pipeline عبور می‌کند تا بالاخره یک Middleware بتواند یک Response برایش تولید کند. بعد از ایجاد Response، دوباره از طریق Pipeline، Middlewareها را از انتها به ابتدا طی خواهد کرد.

در پایان، اولین و آخرین قطعه Middleware، Response را به ASP.NET Core Web Server برمی‌گردانند.

نتیجه گیری: با توجه به توضیحات فوق، در Pipeline اولویت Middlewareها بسیار مهم است.

نکته!!

هر Middleware به Request اصلی و تغییراتی که توسط Middleware قبلی در Pipeline ایجاده شده، دسترسی دارد، بنابراین برای ایجاد یک اپلیکیشن، باید در Pipeline چندین Middleware را با هم ترکیب نمایید.

نحوه تعریف Middleware در Pipeline

زمانیکه اپلیکیشن یک Request دریافت می‌کند، ASP.NET Core از طریق Middleware Pipeline آن را به جریان می‌اندازد و این جریان تا زمانی ادامه دارد که یک Middleware Component بتواند آن را Handle کند. بنابراین می‌توان **گفت:** برای تعیین رفتار اپلیکیشن و نحوه پاسخ به Requestها، Middleware Pipeline یکی از مهم‌ترین بخش‌های پیکربندی است و شما می‌توانید بسیاری از موارد اپلیکیشن را با Middleware مدیریت کنید.

Middleware Pipeline در تمام اپلیکیشن‌های ASP.NET Core، درون متد Configure کلاس Startup و با استفاده از شی IApplicationBuilder تعریف می‌شود. **برای مثال:** MVC یکی از مهمترین Middlewareهاست که برای شما تمام Html Pageها و API Responseها را تولید می‌کند.

در نسخه .NET Core 2.2. با استفاده از اکستنشن متد UseMvcWithDefaultRoute می‌توانستید از این Middleware استفاده نمایید اما در .NET Core 3.1. باید از اکستنشن متدهای UseRouting و UseEndpoints استفاده نمایید.

در نسخه .NET Core 2.2

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
      app.UseMvcWithDefaultRoute();
}
```

MVCMiddleware در .NET Core 2.2

در نسخه .NET Core 3.1

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
       app.UseDeveloperExceptionPage();
    }

     app.UseRouting();
```

MVCMiddleware در .NET Core 3.1

```
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
```

نکته!!

به صورت پیش‌فرض در اکثر اپلیکیشن‌های ASP.NET Core MVC یک قرارداد مشترک جهت نمایش صفحه اصلی اپلیکیشن وجود دارد و آن قرارداد این است که، صفحه اصلی با استفاده از اکشن متد HomeController.Index صدا زده شود. بنابراین با قرارداد پیش‌فرض هر یک از مسیرهای زیر به اکشن متد Index در کنترلر HomeController می‌رسد.

- **/**
- ***/home***
- ***/home/index***

در نسخه .NET Core 3.1 اکستنشن متد UseEndpoints این قرارداد را برای اپلیکیشن، میسر می‌سازد.

Exception Handling چیست؟

اگر شما یک Senior Developer باشید و کدهای عالی هم بنویسید، به محض انتشار و Deploy کردن اپلیکیشن‌تان، کاربران، خواسته یا ناخواسته راهی برای شکستن آن می‌یابند. بنابراین می‌توان به **قطعیت گفت**: خطاها، واقعیت زندگیبه تمامی اپلیکیشن‌هاست و مدیریت خطا در هر اپلیکیشنی، امری واقعا ضروریست.

یک اپلیکیشن خوب باید خطاها را در کوتاه‌ترین زمان ممکن، شناسایی و بهترین بازخورد را به کاربر نشان دهد. بنابراین مهم است که مطمئن شویم، در زمان thrown شدن یک خطا، برنامه به شکست نمی خورد و از همه مهمتر، کاربران، خطاهای اپلیکیشن را در یک قالب کاربرپسند دریافت می‌کنند.

اما بهترین راه حل چیست؟

روش‌های متفاوتی برای مدیریت خطاها وجود دارد، اما مناسب‌ترین روش برای مدیریت خطاها و ارائه قابلیت‌های موردنیاز اپلیکیشن، استفاده از Middlewareهاست.

مایکروسافت چند Middleware برای مدیریت Exceptionها و خطاهای Status Code ارائه داده، که با ترکیب آن‌ها می‌توان مطمئن شد که بروز هر گونه خطا، باعث شکستن اپلیکیشن و نابودی جزییات امنیتی نخواهد شد.

- **DeveloperExceptionPageMiddleware: این Middleware، هنگام ایجاد اپلیکیشن یک Feedback سریع از خطاها را ارائه می‌دهد.**
- **ExceptionHandlerMiddleware: این Middleware، در محیط Production خطا را در یک صفحه‌ی کاربرپسند ارائه خواهد داد.**
- **StatusCodePagesMiddleware: این Middleware، کدهای وضعیت خطا را در یک صفحه خطای کاربرپسند نمایش می‌دهد.**

مدیریت خطا با استفاده از DeveloperExceptionPageMiddleware و ExceptionHandlerMiddleware

Exceptionها معمولا زمانی رخ می‌دهند که شرایط غیرمنتظره پیش بیاید، **به طور مثال:** NullReferenceException یک Exception عمومیست که مطمئنا همه‌ی شما آن را تجربه کرده‌اید و می‌دانید زمانی رخ می‌دهد که تلاش برای دسترسی به شی دارید که هنوز مقداردهی نشده است.

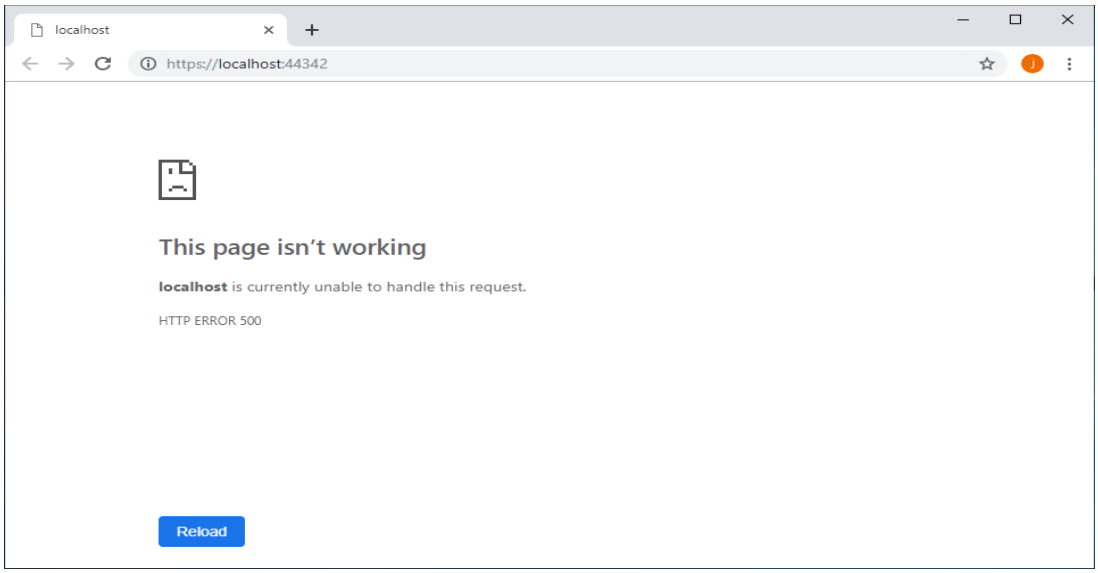
برای درک چگونگی مدیریت خطاها، اولین کاری که باید انجام دهید این است که یک اکسپشن در برنامه خود throw کنید. به همین منظور من یک NullReferenceException در متد Configure قرار دادم.

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;


namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {

        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
        env)
        {
```

در اینجا یک NullReferenceException را با استفاده از متد app.Run در اپلیکیشن قرار دادیم.

```
            app.Run(async (context) =>
              {
                  throw new NullReferenceException();
              });
        }
    }
}
```

حالا اگر اپلیکیشن را اجرا کنید، باید خروجی پایین را ببینید:



معمولا در هنگام توسعه یک اپلیکیشن اگر خطایی رخ دهد، برنامه‌نویس باید تا جایی که امکان دارد از جزییات خطا اطلاع یابید، اما همانگونه که در تصویر بالا می‌بینید دیدن این جزییات شما را خوشحال نمی‌کند و برایتان چندان مفید نخواهد بود.

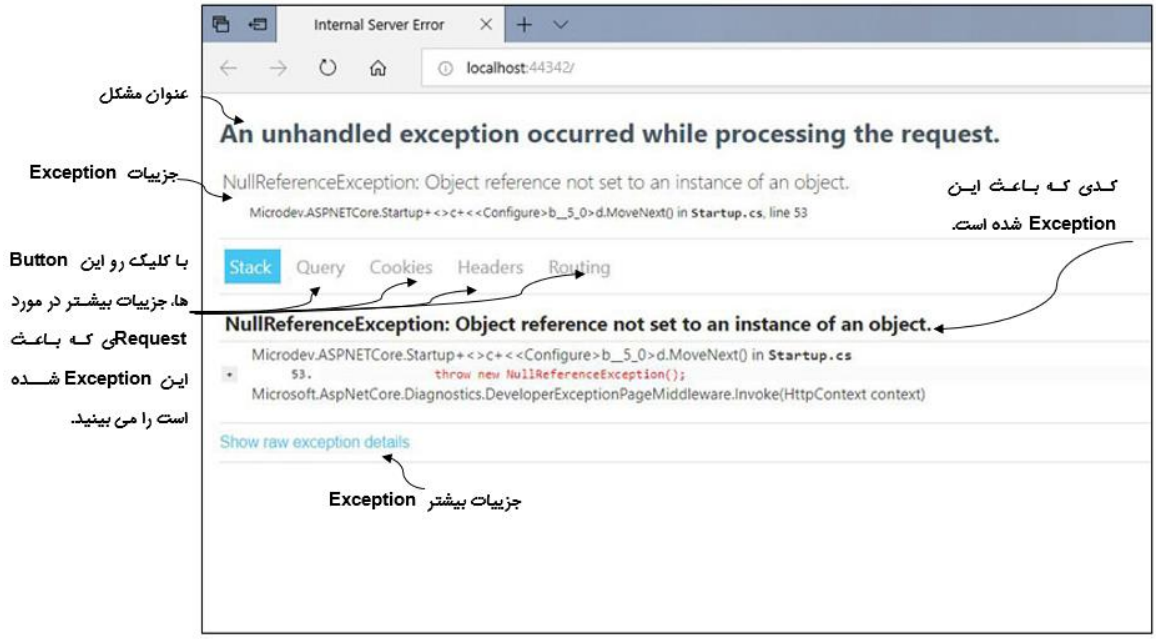
برای حل این مشکل، مایکروسافت DeveloperExceptionPageMiddleware را ارائه داده، تا بتوانید با استفاده از متد app.UseDeveloperExceptionPage() جزییات خطا را ببینید.

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;


namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
```

```
        {

        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
       env)
        {
            app.UseDeveloperExceptionPage();

            app.Run(async (context) =>
              {
                  throw new NullReferenceException();
              });
        }
    }
}
```

این Middleware شـــامـل جزییـات مختلفی درمورد Request و Exception است.

حالا بعد از اجرای اپلیکیشـــن، یک اکســـپشـــن thrown می‌شـــود و ســـپس Pipeline به Middleware DeveloperExceptionPageMiddleware می‌رسد و اکسپشن به دام می‌افتد.



همانطور که می‌بینید این صـــفحه حاوی جزییات بیشـــتری(از جمله: ردیابی Stack، کدی که ســـبب رخ دادن اکسـپشـن شـده و جزییاتی در مورد Request مانند کوکی‌ها و Headerهایی که ارسـال شـده اسـت) در مورد Request و Exception است.

برای حل مشکل، داشتن این جزییات بسیار ارزشمند است، اما این جزییات تنها باید به برنامه‌نویس نمایش داده شود زیرا کاربران هرگز نباید جزییات بیشتری از درخواست شما و موارد ضروری بدانند و این یک ریسک امنیتی بسیار بزرگ است.

پس چاره چیست؟ پاسخ این سوال را Hosting Environment می‌دهد. با ما همراه باشید تا در مورد این راه حل بیشتر بدانیم.

Hosting Environment چیست؟

یکی از جذاب‌ترین ویژگی ASP.NET Core که همراه با فریم ورک ارائه شـــده، وجود مدیریت محیط‌های میزبانی(Hosting Environment) است که سبب می‌شود تا شما بدون هیچ‌گونه مشکلی، با محیط‌های متعدد کار کنید.

به زبان ساده‌تر، محیط‌های میزبانی این امکان را به شما می‌دهد تا بدون نیاز به تغییر محیط، یک اپلیکیشن را در محیط‌های مختلف تست و اجرا نمایید.

اکثر اپلیکیشن‌هایی که ما در حال توسعه آن هستیم، باید در زمان‌ها و محیط‌های مختلف، رفتارهای متفاوت داشته باشند. **به عنوان مثال:** هرگز نباید جزییات خطای اپلیکیشن را به کاربر نهایی نمایش داد زیرا این کار یک خطر امنیتی است و این اطلاعات تنها برای برنامه‌نویسان (جهت بررسی مشکل) ارزشمند است.

پس با توجه به توضیحات بالا، ما هر اپلیکیشنی که ایجاد می‌کنیم باید بتواند در محیط‌های چندگانه اجرا شود و به طور پیش‌فرض هر اپلیکیشـــن باید حداقل ۲ یا چند محیط را مدیریت کند.**به عنوان مثال:** محیط Development و Production و در بعضی موارد محیط Staging هم اضافه می‌شود.

شما در مثال بالا هنگام بروز خطا، جزییات اپلیکیشن را به کاربران نمایش دادید و سپس متوجه شدید این جزییات تنها باید برای برنامه‌نویسان به نمایش درآید. برای حل این مشکل باید از DeveloperExceptionPage تنها زمانی که اپلیکیشن در محیط Development قرار دارد، استفاده کنید.

سوال: اگر DeveloperExceptionPage برای محیط Production مناسب نیست، پس چطور باید اکسپشن‌ها را در این محیط مدیریت کنیم؟

برای پاسخ به این سوال به مثال زیر توجه نمایید:

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;


namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
          services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
       env)
        {
            if (env.IsDevelopment())
            {
               app.UseDeveloperExceptionPage();
            }
            else
            {
               app.UseExceptionHandler("/Home/Error");
            }

           app.UseRouting();
           app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });

           app.Run(async (context) =>
            {
                throw new NullReferenceException();
            });
        }
    }

}
```

زمانیکه در محیط **Development** قرار دارید **DeveloperExceptionPageMiddleware** به **Pipeline** اضافه می‌شود.

و زمانیکه در محیط **Production** قرار دارید، **ExceptionHandlerMiddleware** به **Pipeline** اضافه خواهد شد.

همانطور که می‌بینید، راه‌حل این مشکل استفاده از اینترفیس IWebHostEnvironment و Middleware ← ExceptionHandlerMiddleware است.

IWebHostEnvironment جهت ارئه رفتارهای متفاوت در مورد جزییات محیط فعلی به Configure پاس داده می‌شود، بنابراین شما می‌توانید با استفاده از این اینترفیس، بین محیط‌های مختلف جابه‌جا شوید. حالا اگر اکسپشنی رخ دهد، و اپلیکیشن در محیط Development باشد، صفحه‌ی خطا با جزییات نمایش داده می‌شود وگرنه کاربر با صفحه‌ای سفارشی که شامل جزییات مورد نیاز است روبرو خواهد شد.

توجه داشته باشید، در ورودی ExceptionHandlerMiddleware مسیر صفحه‌ای که باید به کاربر نمایش داده شود را قرار دهید. در مثال بالا **/Home/Error** اشاره به همین مسیر دارد.

app.UseExceptionHandler("/Home/Error");

ExceptionHandlerMiddleware بعد از برخورد با هر گونه اکسپشن، این مسیر را فرخوانی خواهد کرد، بنابراین باید قبل از اجرای اپلیکیشن، دو مورد مهم را تنظیم نمایید:

۱- تنظیم محیط میزبانی

۲- ایجاد یک صفحه سفارشی

تنظیم محیط میزبانی

برنامه شما چطور محیط میزبانی را در زمان اجرا مشخص می‌کند؟

ASP.NET Core با استفاده از یک متغیر محیطی به نام ASPNETCORE_ENVIRONMENT محیط جاری را شناسایی و از اینترفیس IWebHostEnvironment جهت بررسی مقدار آن استفاده می‌نماید. **به‌طور مثال:** اگر این متغیر برابر Development باشد، بدین معنی‌ست که اپلیکیشن در مد Development اجرا شده است.

نکته!!

شما می‌توانید IWebHostEnvironment را هر جایی از اپلیکیشن Inject کنید، اما من توصیه می‌کنم که در داخل سرویس‌های خود از آن استفاده نکنید.

نکته!!!

اگر اپلیکیشن ASP.NET Core شما نتواند متغیر ASPNETCORE_ENVIRONMENT را هنگام Startup بیابد،به طور پیش‌فرض، محیط Production را در نظر می‌گیرد. این‌بدین خاطر است که

هنگامی که Configuration Providerهایتان را Deploy می‌نمایید، به طور پیش‌فرض از محیط درست استفاده کنید.

اینترفیس IWebHostEnvironment جهت ارائه رفتارهای متفاوت برای محیط جاری، تعدادی Property ارائه داده است:

- **ContentRootPath**: جهت گرفتن یا تنظیم مسیر پوشه ای که حاوی محتوای اپلیکیشن است، می‌توان از این Property استفاده نمود.
- **WebRootPath**: این Property مکان wwwroot (که شامل فایل‌های استاتیک است) را مشخص می‌کند.
- **EnvironmentName**: این Property مقدار متغیر ASPNETCORE_ENVIRONMENT را تنظیم می‌نماید. معمولا، مقدار آن یکی از سه حالت "Development", "Staging", "Production" است و ASP.NET Core هم، تعدادی متد کمکی برای کار کردن با این سه مقدار ارائه داده است:
  - ✓ **IWebHostEnvironment.IsDevelopment()**
  - ✓ **IWebHostEnvironment.IsStaging()**
  - ✓ **IWebHostEnvironment.IsProduction()**
  - ✓ **HostingEnvironment.IsEnvironment(string environmentName)**

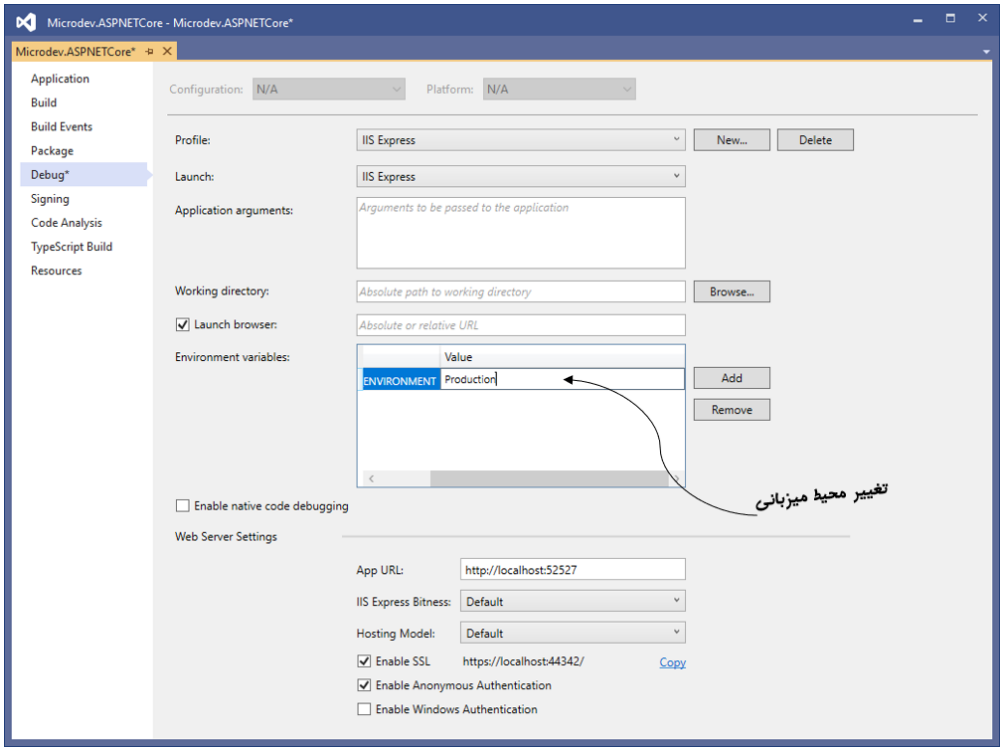
نکته!!

مقدار EnvironmentName در فرایند Bootstrapping اپلیکیشن و قبل از ایجاد ConfigurationBuilder، مشخص می‌شود. این Property در زمان تنظیم Middleware Pipeline، مکان مشترکی برای سفارشی کردن اپلیکیشن براساس محیط است.

خوب، مقدمه کافیست، می خواهم هر چه سریعتر روش‌های تنظیم محیط میزبانی را به شما یاد دهم.

به طور پیش‌فرض، Visual Studio از حالت Development استفاده می‌کند، اما اگر می‌خواهید این Mode را تغییر یا یک محیط جدید، پیکربندی نمایید، باید متغیر Environment را با یکی از دو روش زیر تنظیم نمایید:

۱) **روش اول**: در Solution بر روی Properties دابل کلیک کنید و از کادر باز شده تب Debug را برگزینید. حالا همانند تصویر زیر، مقدار ASPNETCORE_ENVIRONMENT را از Development به Prouduction تغییر دهید و سپس فایل را ذخیره کنید.



۲) **روش دوم:** شـــما به راحتی می‌توانید فایل launchSettings.json را در مســـیر ➔Solution Properties باز و مقدار این متغیر را ویرایش کنید:

اســـتفاده از این Environment variableها، یک روش عالی برای تســـت اپلیکیشـــن در محیط‌های مختلف است. شما در اینجا، حتی می‌توانید یک پروفایل Debugging جدید اضافه نمایید و به سرعت، بین محیط‌های تستی مختلف سویچ کنید.

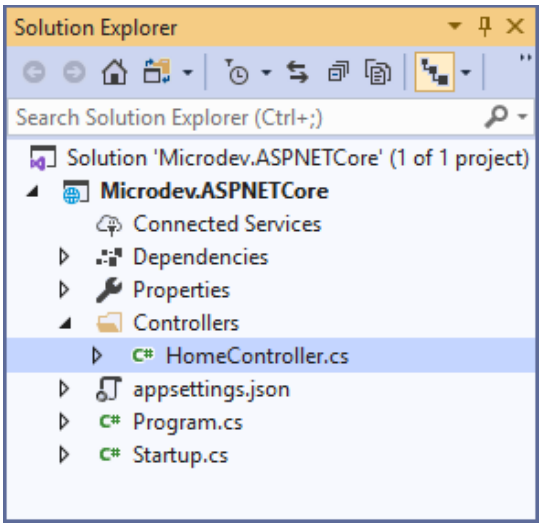
نکته!!

بخش iisSettings شـامل تمام تنظیمات مرتبط به IISExpress اسـت، در حالی که بخش profiles شامل تنظیمات Kestrel می‌باشد.

ایجاد صفحه سفارشی

برای ایجاد یک صفحه سفارشی باید مراحل زیر را دنبال نمایید:

- **در Solution یک Folder به نام Controllers ایجاد کنید.**
- **سپس درون این Folder یک کلاس به نام HomeController اضافه نمایید.**

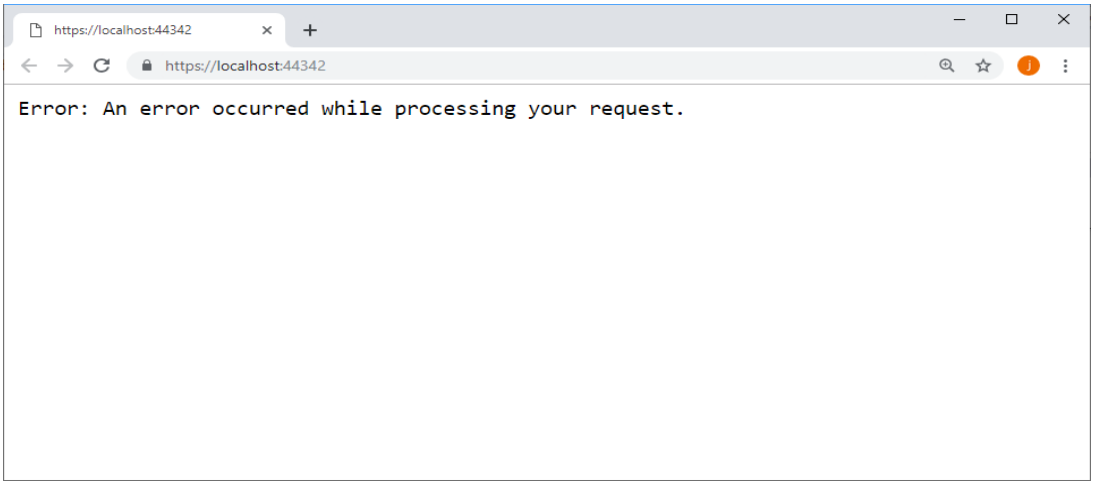


حالا همانند کد پایین، در کلاس HomeController یک متد به نام Error قرار دهید:

```
namespace Microdev.ASPNETCore.Controllers
{
    public class HomeController
    {
        public string Error()
        {
          return "Error: An error occurred while processing your request.";
        }
    }
}
```

این متد یک متن خطا برمی گرداند.

لطفا اپلیکیشن را اجرا نمایید.



همانطور که می‌بینید، در صفحه خطای سفارشی، شما می‌توانید به جای نمایش جزییات کامل اکسپشن، خطا را در قالب یک پیام مناسب به کاربران ارائه دهید.

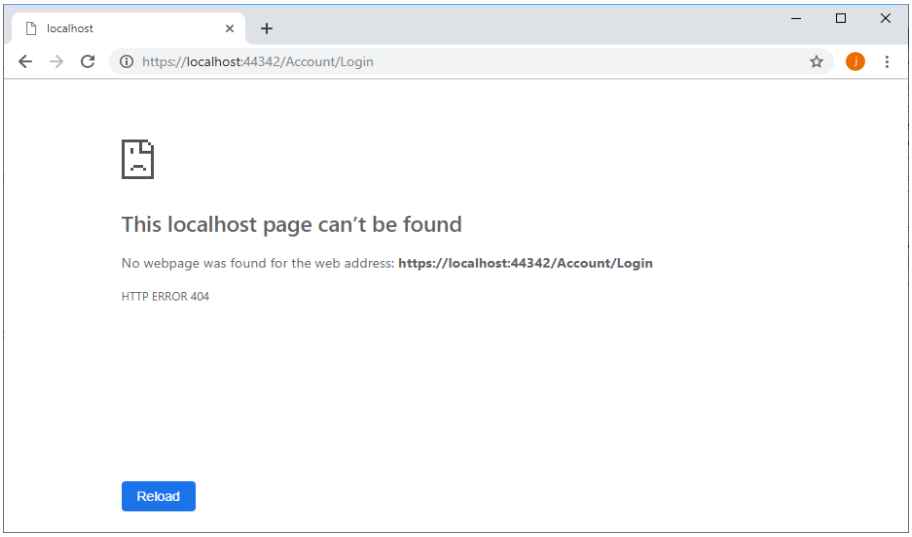
مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter2/Sample1

مدیریت خطاها با استفاده از StatusCodePagesMiddleware

همانطور که قبلا گفته شد، مدیریت خطاها در هنگام توسعه هر اپلیکیشنی، جزیی از ضروریات است و شما باید انواع خطاها را به بهترین نحوه مدیریت کنید. مطمئنا شما هم می‌دانید که اپلیکیشن می‌تواند طیف گسترده‌ای از HTTP status codeهایی که نمایانگر انواع خاصی از خطاها هستند را برگرداند. **برای مثال:** خطای 404 که بسیار مرسوم است و اغلب زمانی اتفاق می‌افتد که کاربر یک URL نامعتبر را درخواست کند.

بدون مدیریت این Status Codeها، کاربران یک صفحه خطای عمومی که بسیار گیج کننده هست را می‌بینند و ممکن است تصور کنند که اپلیکیشن شما خراب شده است. برای روشن‌تر شدن این موضوع، برنامه خود را اجرا کنید و مانند تصویر زیر یک آدرس FAKE وارد کنید:



مایکروسافت یک راه حل خوب برای حل این مشکل ارائه داده و آن استفاده از Middlewareی به نام StatusCodePagesMiddleware می‌باشد.

این Middleware برای مدیریت Status Codeها یک صفحه خطا ارائه می‌دهد که البته شما می‌توانید یک صفحه سفارشی که با سایر قسمت‌های اپلیکیشن‌تان همخوانی دارد را به کاربر نمایش دهید.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }


        public void Configure(IApplicationBuilder app, IWebHostEnvironment
        env)
        {
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }

            app.UseRouting();
```

زمانی که در مد Development هستید، StatusCodePagesMiddleware به Pipeline اضافه می‌شود.

```
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

با استفاده از این متد، Middleware هر Responseی که HTTP Status Code آن بین 4xx یا 5xx باشد و هیچ Response Body نداشته باشد را رهگیری می‌کند.

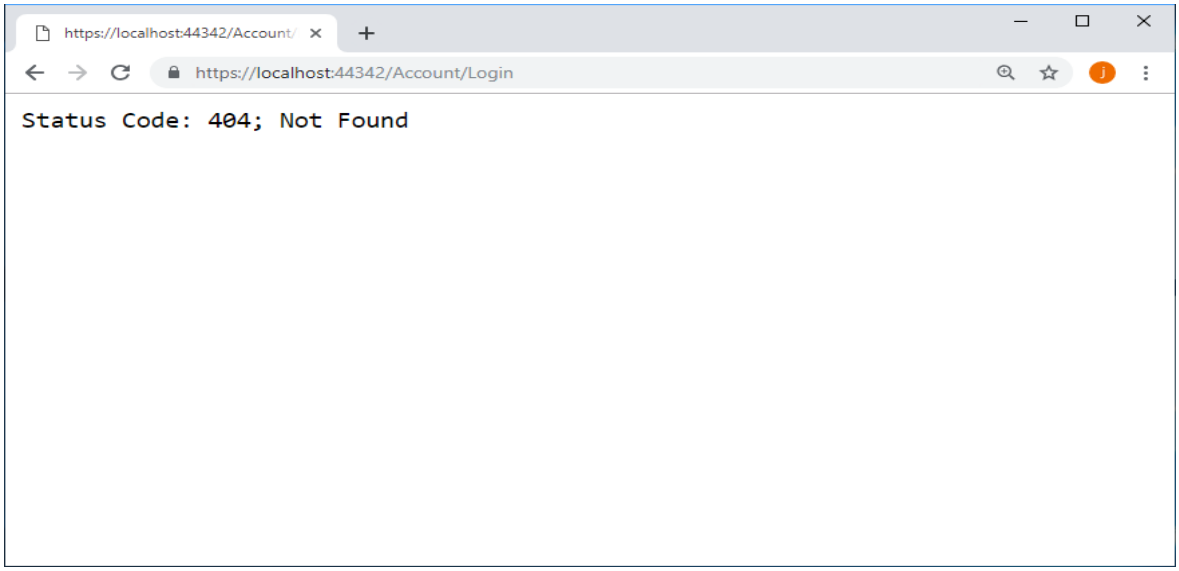
این قابلیت فوق‌العاده به شما امکان می‌دهد تا برای انواع خطاها، صفحات داینامیک ارائه دهید.

نکته!!

قبل از اجرای برنامه، متغیر محیطی ASPNETCORE_ENVIRONMENT را به Development تغییر دهید.

حالا لطفا اپلیکیشن را اجرا و آدرس FAKE پایین را وارد نمایید.

https://localhost:44342/Account/Login



اما برای استفاده از StatusCodePageMiddleware در مد Production می‌توانید از UseStatusCodePagesWithReExecute استفاده کنید. این Middleware از تکنیک مشابه به ExceptionHandlerMiddleware استفاده می‌کند.

app.UseStatusCodePagesWithReExecute("/error/{0}");

UseStatusCodePagesWithReExecute اکستنشن متد StatusCodePageMiddleware را تنظیم می‌کند، تا هر زمان که Response Codeی بین 4xx یا 5xx یافت شد، با استفاده از مسیر خطایی که ارائه شده، Pipeline را دوباره اجرا کند.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
          services.AddControllersWithViews();
        }

       public void Configure(IApplicationBuilder app, IWebHostEnvironment
      env)
        {
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseMvcWithDefaultRoute();
        }
    }
}
```

زمانیکه اپلیکیشن در مد **Production** باشد، **StatusCodePagesWithReExecuteMiddleware** به **Pipeline** اضافه خواهد شد.

حالا در کلاس HomeController متد Error را تغییر دهید:

```
namespace Microdev.ASPNETCore.Controllers
{
    public class HomeController
    {
        public string Error(int id)
        {
           return $"{id} Error: Oops! We couldn't find the page you
          requested";
        }
```

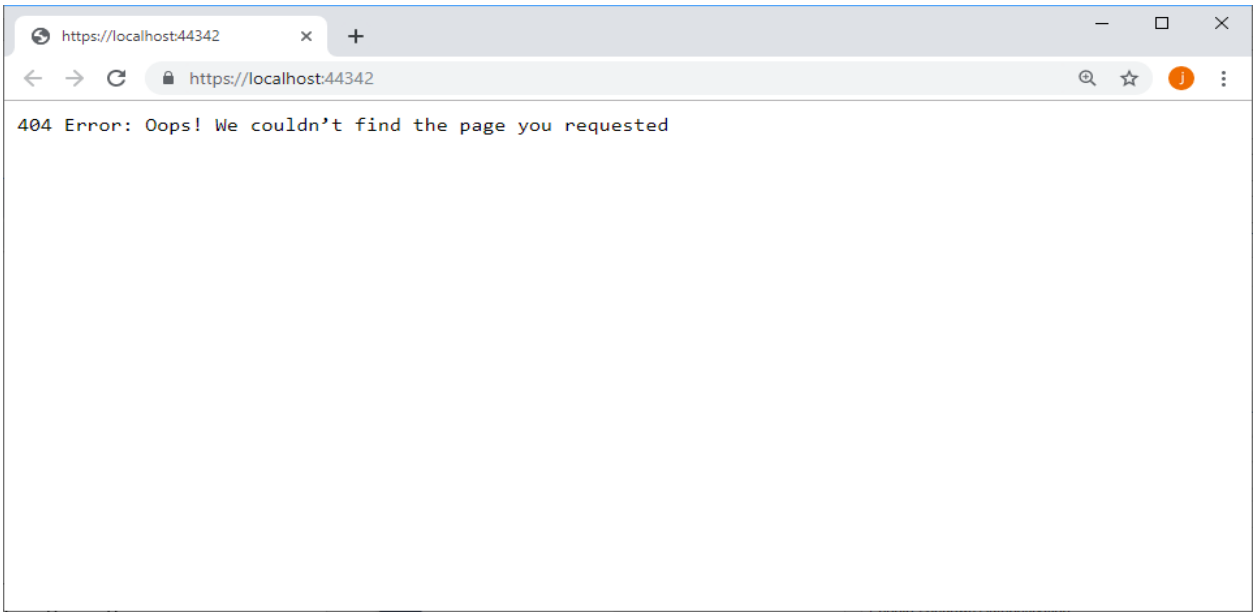
این متد یک متن خطا برمی‌گرداند.

```
    }
}
```

نکته!!

قبل از اجرای اپلیکیشن، متغیر ASPNETCORE_ENVIRONMENT را به Production تغییر دهید.

حالا اپلیکیشن را اجرا کنید.



نکته!!

مسیر "{0}/error/" حاوی یک فرمت string token, {0} است. هنگامیکه مسیر دوباره اجرا می‌شود، Middleware این Token را با عدد Status Code جایگزین می‌کند. برای مثال یک خطای 404، مسیر /error/404 را اجرا می‌کند.

مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter2/Sample2

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **MVC چیست؟**
- ✓ **سیستم MVC چگونه کار می‌کند؟**
- ✓ **Validation و DataAnnotations attribute چیست؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: Explain Middleware in ASP.NET Core?

Q2: What are the benefits of Middleware?

Q3: What is the difference between middleware and HTTP module?

Q4: Explain the purpose of Configure method.

Q5: Explain Hosting Environment in ASP.NET Core?

Q6: How does the ASP.NET Core handle the errors?

Q7: What are cross-cutting concerns?

Q8: What are different ASP.NET Core diagnostic middleware and how to do error handling?

Q9: What is the role of IWebHostEnvironment interface in ASP.NET Core?

Q10: What is launchsetting.json in ASP.NET Core?

Quiz

Q1: Which of the following is executed on each request in ASP.NET Core application?

1. Startup
2. Middlewares
3. Main method
4. All of the above

Q2: Middlwares can be configured in _____ method of Startup class.

1. Configure
2. ConfigureService
3. Main
4. ConfigureMiddleware

Q3: Which of the following is environment variable in ASP.NET Core application?

1. ASPNET_ENV
2. ENVIRONMENT_VARIABLE
3. ASPNETCORE_ENVIRONMENT
4. ENVIRONMENT

Q4: Which of the following isn't Helper methods in ASP.NET Core application?

1. IWebHostEnvironment.IsDevelopment()
2. IWebHostEnvironment.IsStaging()
3. IWebHostEnvironment.IsProduction()
4. IWebHostEnvironment.IsTest()

Q5: IWebHostEnvironment interface to checks value:

1. ASPNET_ENV
2. ENVIRONMENT_VARIABLE
3. ASPNETCORE_ENVIRONMENT
4. ENVIRONMENT

Q6: Which of the following extension method allow us to configure custom error handling route?

1. UseExceptionHandler
2. UseDeveloperExceptionPage
3. UseWelcomePage
4. Run

Q7: Which of the following extension method allow us to configure status code error handling route?

1. UseExceptionHandler
2. UseWelcomePage
3. StatusCodePagesMiddleware
4. StatusCodePagesWithReExecute

Q8: Middlewares can be configured using instance of type _____.

1. IWebHostEnvironment
2. ILoggerFactory
3. IApplicationBuilder
4. IMiddleware

Q9: By default, Visual Studio uses _____mode.

1. Development
2. Production
3. Staging
4. Testing

Q10: Using _____interface, you can switch back and forth between different environments during testing.

1. IWebHostEnvironment
2. ILoggerFactory
3. IApplicationBuilder
4. ILoggerProvider

Answers

1-Correct Answer: Middlewares

2-Correct Answer: Configure

3-Correct Answer: ASPNETCORE_ENVIRONMENT

4-Correct Answer: IWebHostEnvironment.IsTest()

5-Correct Answer: ASPNETCORE_ENVIRONMENT

6-Correct Answer: UseExceptionHandler

7-Correct Answer: StatusCodePagesMiddleware

8-Correct Answer: IApplicationBuilder

9-Correct Answer: Development

10-Correct Answer: IWebHostEnvironment

خلاصه فصل

- ✓ **Middlewareها کلاس‌های سی‌شارپی هستند که می‌توانند HTTP Request یا Response HTTP را مدیریت کنند.**
- ✓ **Middleware کامپوننت‌های کوچکی هستند که زمانیکه اپلیکیشن یک HTTP Request را دریافت می‌کند به صورت پشت سرهم[۵] اجر می‌شوند.**
- ✓ **Middleware Pipeline دو طرفه است، Requestها از طریق هر Middleware در یک مسیر عبور می‌کنند و Responseها به ترتیب در جهت معکوس از طرف دیگر خارج می شوند.**
- ✓ **Middleware می‌تواند یک درخواست HTTP ورودی را پردازش کند، آن را تغییر دهد و سپس آن را به Middleware بعدی منتقل کند.**
- ✓ **هنگام توسعه یک اپلیکیشن، DeveloperExceptionPageMiddleware می‌تواند اطلاعات زیادی در مورد خطا به ما دهد، اما فراموش نکنید که در مد Production هرگز نباید استفاده شود.**
- ✓ **ExceptionHandlerMiddleware این امکان را به شما می‌دهد تا در زمان بروز خطا، پیغام خطای سفارشی را نمایش دهید.**
- ✓ **StatusCodePagesMiddleware به شما این امکان را می‌دهد تا زمانیکه Pipeline یک Status Code برمی‌گرداند، شما هم برای مدیریت خطا یک پیغام خطای سفارشی برگردانید.**

---

[۵] Sequence

فصل سوم : MVC Design Pattern

آنچه خواهید آموخت:

- MVC چیست؟
- سیستم MVC چطور کار می‌کند؟
- مزایای MVC Design Pattern
- پیاده‌سازی MVC در وب اپلیکیشن ASP.NET Core
- Model چیست؟
- Validation Attributeها
- Controller و Action Method چیست؟

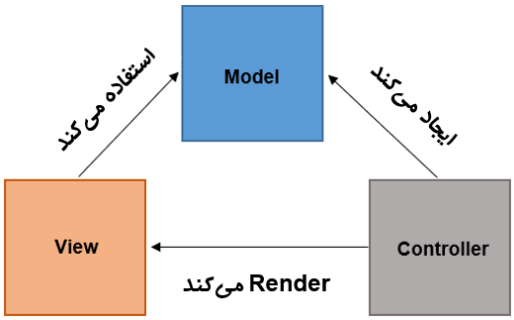
MVC چیست؟

MVC یک الگوی محبوب طراحی نرم‌افزار است که در بسیاری از سیستم‌های نرم‌افزاری استفاده می‌شود و هدفش Separation of concern است.

این الگو، ماژول نرم‌افزاری را به سه لایه‌ی متمایز، تقسیم می‌کند که هر لایه مسئول یک جنبه واحد از سیستم است و ترکیب این لایه‌ها می‌تواند برای تولید UI مورد استفاده قرار گیرد.

اجزای MVC

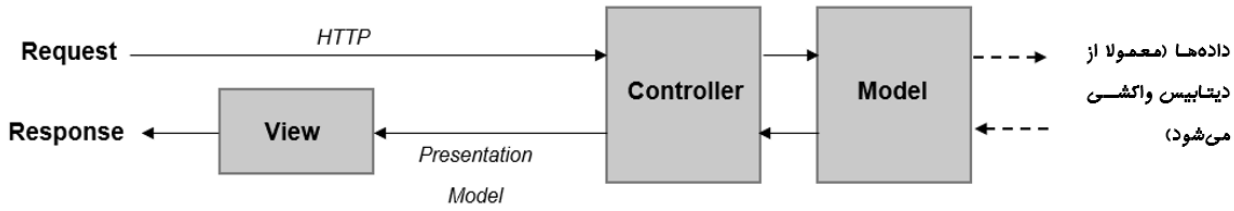
- **Model**: Model نمایانگر داده‌هایی است که باید نمایش داده شوند. یک مدل می‌تواند یک Object یا یک Type یا مجموعه‌ای از چند Object باشد.
- **View**: قالبی است که اطلاعات ارائه شده توسط مدل را نمایش می‌دهد.
- **Controller**: مدل را به روز می‌کند و View مناسب را انتخاب می‌نماید.



سیستم MVC چطور کار می‌کند؟

MVC ما را قادر می‌سازد تا وب اپلیکیشن‌های داینامیک را با استفاده از مفهوم Separation of concern بسازیم. این سیستم بدین صورت عمل می‌نماید:

۱) **Controller به عنوان نقطه‌ی ورود تعامل کاربر با اپلیکیشن است، که Request را دریافت می‌کند. بنابراین، کاربر تنها با Controller در ارتباط است.**
۲) **هنگامی که یک Request دریافت می‌شود، بسته به ماهیت Request، کنترلر یا داده‌های درخواست شده را از Model می‌گیرد یا داده‌هایی مدل را Update می‌نماید.**
۳) **در مرحله‌ی بعد، Controller جهت نمایش اطلاعات یک View را انتخاب و مدل رابه آن انتقال می‌دهد.**
۴) **حالا این View برای تولید UI از اطلاعات موجود در مدل استفاده می‌کند.**



مزایای MVC Design Pattern

MVC مزایای زیر را به ارمغان آورده است:

۱. **اولین مزیت:** در این الگوی طراحی، View, Model و Controller از یکدیگر جدا شده است، بنابراین، اولین مزیت وجود مفهوم Separation of concern است که اجازه می‌دهد تا اعضای تیم‌های مختلف، تنها بر روی بخش‌هایی از اپلیکیشن که با مجموعه مهارت‌های مربوط به آنها هماهنگ است، تمرکز کنند.

۲. **دومین مزیت:** Model مستقل از View است، این مزیت باعث می‌شود تا تست‌پذیری بهبود یابد. بنابراین می‌توان اطمینان حاصل نمود که مدل بدون هیچ وابستگی به ساختار UI، به راحتی قابل تست می‌باشد.

پیاده‌سازی MVC در وب اپلیکیشن ASP.NET Core

ASP.NET Core یک Web Framework با ماژول‌های Reusable است، که MVC یکی از پرکاربردترین ماژول های آن می‌باشد.

ASP.NET Core از Middlewareهایی به نام UseRouting و UseEndpoints برای پیاده‌سازی MVC استفاده کرده است. این Middlewareها اصلی‌ترین نقطه‌ی ورود کاربران برای ارتباط با اپلیکیشن است و قابلیت‌های زیادی را جهت ایجاد هر چه سریع‌تر و کارآمدتر اپلیکیشن به شما ارائه می‌دهند.

برای استفاده از این قابلیت فوق‌العاده شما به دو مرحله نیاز دارید:

۱) **اضافه کردن سرویس AddControllersWithViews در متد ConfigureServices.**

۲) **افزودن UseRouting و UseEndpoints در متد Configure.**

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
           services.AddControllersWithViews();
        }

         public void Configure(IApplicationBuilder app,
         IWebHostEnvironment env)
         {
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
             });
        }
    }

}
```

شما باید AddControllersWithViews () را در اینجا اضافه نمایید.

افزودن UseRouting() و MapControllerRoute().

نکته!!

در وب اپلیکیشن‌های ASP.NET Core، MvcMiddleware معمولا پایانی‌ترین قسمت Middleware در Pipeline است.

Model چیست؟

Model نمایانگر تمام منطق بیزینس و نحوه آپدیت و همچنین اصلاح وضعیت داخلی اپلیکیشن است. دو نوع Model وجود دارد:

۱) **View Model:** View Modelها برای انتقال اطلاعات از Controller به View استفاده می‌شوند.

۲) **Domain Model**: Domain Modelها حاوی داده‌هایی در حوزه بیزینس هستند که جهت ایجاد، ذخیره‌سازی و تغییر داده‌ها با عملگرها و Ruleها همراه می‌شوند.

نکته!!

عملکرد اصلی Domain Modelها، توصیف دامنه‌ی شماست، بنابراین شما می‌توانید از هر کلاس دات‌نت به عنوان یک Domain Model استفاده کنید، اما معمولا بهتر است کلاس‌هایی را ایجاد کنید که واقعا نقش مدل را ایفا می‌کنند.

از آنجا که اعتبارسنجی، موضوع بسیار مهمی است و شما در ایجاد هر اپلیکیشنی باید آن را در نظر بگیرید، پس قبل از ایجاد اولین Model، اجازه دهید کمی در مورد اعتبارسنجی داده‌ها صحبت کنیم.

Validation Attributeها

Validation Attributeها به شما اجازه می‌دهند، قوانینی مشخص کنید تا Propertyهای مدل، مطابق با آن رفتار کنند.

نکته هیجان‌انگیز در مورد Validation Attributeها این است که آن‌ها، همیشه حالت مورد انتظار Model را اعلام می‌کنند، بنابراین شما همیشه می‌توانید داده‌های ارائه شده توسط کاربران را Validate نمایید.

جهت درک چگونگی کارکرد Validationها، قبل از هر کاری باید نحوه استفاده‌ی Attributesها یا بهتر بگویم DataAnnotationها در مدل را بشناسیم.

DataAnnotation چیست؟ DataAnnotationها، متادیتاهایی را مشخص می‌کند که این متادیتاها توصیف داده و همچنین تعیین قوانین و خصوصیاتی که داده‌ها باید پیروی کنند را برعهده دارد.

نکته!!

DataAnnotation علاوه بر اعتبارسنجی، برای اهداف دیگر هم استفاده می‌شود. به عنوان مثال: Entity Framework برای قوانین زمان ایجاد جداول دیتابیس(از روی کلاس‌های C#) و تعریف Type ستون‌ها از DataAnnotationها استفاده می‌کند.

من برخی از DataAnnotations رایج را در ادامه بیان کردم:

- ✓ **[EmailAddress]** : اعتبارسنجی می‌کند که‌یک Property از قالب آدرس ایمیل معتبر برخوردار است یا خیر؟
- ✓ **[StringLength(max)]** : اعتبارسنجی می‌کند که یک رشته حداکثر تعداد کاراکتر را دارد؟
- ✓ **[MinLength(min)]** : اعتبارسنجی می‌کند که یک رشته حداقل تعداد کاراکتر را دارد؟
- ✓ **[Phone]** : اعتبارسنجی می‌کند که یک Property دارای قالب شماره تلفن معتبر است؟
- ✓ **[Range(min, max)]** : اعتبارسنجی می‌کند که مقدارِیک Property بین مقدار min و max است؟
- ✓ **[RegularExpression(regex)]** : اعتبارسنجی می‌کند که‌یک Property با الگوی regex مطابقت دارد؟
- ✓ **[Url]** : اعتبارسنجی می‌کند که یک Property از قالب URL معتبر برخوردار است.
- ✓ **[Required]** : اعتبارسنجی می‌کند که Property حتما مقداردهی شده باشد.

نکته!!

اگر طرفدار رویکرد Attribute-based نیستید، می‌توانید زیرساخت اعتبارسنجی خود را جایگزین کنید. برای مثال: جهت بهروری از ویژگی‌های DataAnnotationها، می‌توانید از کتابخانه محبوب FluentValidation استفاده کنید.

نکته!!

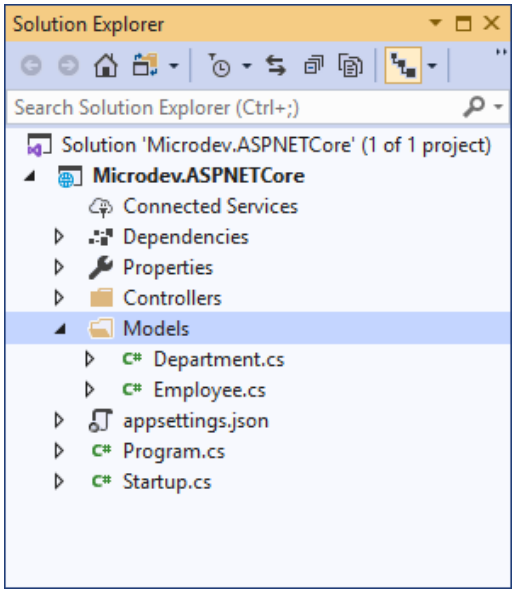
اعتبارسنجی در MvcMiddleware قبل از اجرای اکشن‌متد و بعد از Model Binding رخ می‌دهد.

ایجاد Model

هنگامی که یک پروژه ASP.NET Core MVC جدید ایجاد می‌کنید بهتر است مدل‌های خود را در فولدری به نام Models قرار دهید. من در این پروژه می‌خواهم، در مسیر Models یک کلاس به نام Employee و کلاسی دیگر به نام Department ایجاد کنم:

۱) **یک Folder به نام Models ایجاد نمایید.**

۲) **سپس در این Folder یک کلاس با نام Employee و سپس کلاس دیگر با نام Department اضافه نمایید.**



کلاس Employee:

```
using System.ComponentModel.DataAnnotations;

namespace Microdev.ASPNETCore.Models
{
    public class Employee
    {
        public int EmployeeId { get; set; }

        [Required]
        [MaxLength(64)]
        public string FirstName { get; set; }

        [Required]
        [MaxLength(64)]
        public string LastName { get; set; }

        [Required]
        public decimal Salary { get; set; }

        public int? BossId { get; set; }

        public Employee Boss { get; set; }

        [Required]
        public int DepartmentId { get; set; }

        [Required]
        public Department Department { get; set; }
```

```
    }
}
```

کلاس **Department**:

```
using System.ComponentModel.DataAnnotations;

namespace Microdev.ASPNETCore.Models
{
    public class Department
    {
        public int DepartmentId { get; set; }

        [Required]
        [Display(Name = "CompanyName")]
        public string Name { get; set; }

    }

}
```

همانطور که می‌بینید، در این دو کلاس جهت اعمال برخی ویژگی‌های خاص، از Data Annotationها استفاده شده است.

تا اینجای پروژه آموختیم Data Annotation چیست و چگونه باید از آن‌ها استفاده کرد. از این به بعد کار ما ساده‌تر خواهد شد، زیرا برای Validate یک ش ی، می‌توانیم از Propertyی به نام ModelState.IsValid که توسط کلاس ControllerBase ارائه می‌شود استفاده کنیم.

به مثال پایین دقت نمایید:

- **یک کلاس EmployeeController در فولدر Controllers ایجاد نمایید.**
- **سپس کدهای پایین را در آن قرار دهید.**

```
using Microdev.ASPNETCore.Models;
using Microsoft.AspNetCore.Mvc;

namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController: Controller
    {
        [HttpPost]
        public IActionResult CreateEmployee(Employee employee)
        {
            if (ModelState.IsValid)
            {
                // بیزینس شما
```

در صورتیکه نتیجه‌ی **ModelState.IsValid** برابر **True** باشد، بیزینس اعمال می‌شود.

```
            return View(employee);
        }
        else
        {
            return BadRequest(ModelState);
        }
    }


}
}
```

در مثال بالا، ما یک مدل Employee از ورودی اکشن‌متد دریافت می‌کنیم، در صورتی که نتیجه ModelState.IsValid برابر True باشد(یعنی داده‌های Model ما معتبر باشد) بیزینس مورد نظر اعمال خواهد شد وگرنه یک BadRequest برگردانده می‌شود.

نکته!!

ما درباره HttpPost در فصل API صحبت خواهیم کرد.

Controller و اکشن‌متد چیست؟

Controller یک کلاس با تعدادی اکشن‌متد است که می‌تواند از کلاس پایه System.AspNetCore.Mvc.Controller ارث‌بری کند.

Controllerها نقطه ورود برای تعامل کاربران و مکان مدیریت Requestها هستند و سه Role در اپلیکیشن MVC را فراهم می‌کنند:

1) **View موردنظر را انتخاب می‌کنند.**
2) **واسطی بین Model و View را فراهم می‌نمایند.**
3) **داده‌ها را قبل از انتقال، پردازش می‌کنند.**

براساس قرارداد، بهتر است تمامی Controllerها درون یک Folder به نام Controllers قرار گیرند و نام آن‌ها با پسوند Controller خاتمه یابد. **به عنوان مثال: HomeController.**

نکته!!

کلاس پلیه Controller، از کلاس پلیه ControllerBase ارث‌بری می‌کند، بنابراین کلاس‌هایی که از Controller ارث‌بری می‌کنند، در حقیقت از کلاس پلیه‌ی ControllerBase هم ارث‌بری می‌نمایند، پس این کلاس‌ها به طور خودکار Attribute [Controller] را دریافت خواهند کرد.

همانطور که گفته شد، درون Controller یک یا چند متد وجود دارد که معمولا یک شی از نوع IActionResult را برمی‌گردانند و وظیفه اصلی Controller هم انجام برخی عملیات از طرف کاربر و سپس انتقال داده‌های Model به Action Result است. پس می‌توان نتیجه گرفت، منطق اپلیکیشن، درون Controller انجام می‌شود و سپس این Controller مدلی ایجاد می‌کند که وضعیت برنامه و منطق بیزینس را در خود جای می‌دهد.

اکشن متد چیست؟ یک اکشن‌متد، متدیست که در پاسخ به یک Request اجرا شده و معمولا یک IActionResult که حاوی دستورالعمل‌هایی برای رسیدگی به Request است را برمی‌گرداند.

هنگامی که اپلیکیشن یک Request دریافت می‌کند، اکشن‌متد شما اجرا می‌شود و برای برگرداندن HTTP Response باید یک Result تولید کند.

نکته!!

Action Result، داده‌های مدل را فراهم نموده و این داده‌ها را به فرمت خروجی تبدیل می‌کند.

بیایید در HomeController که قبلا ایجاد کردیم یک اکشن‌متد ساده اضافه نماییم.

```
using Microsoft.AspNetCore.Mvc;

namespace Microdev.ASPNETCore.Controllers
{
    public class HomeController: Controller
    {

        public IActionResult Index()
        {
            return View();
        }


        public string Error(int id)
        {
           return $"{id} Error: Oops! We couldn't find the page you
           requested";
        }
```

این اکشن‌متد یک View با نام Index برمی‌گرداند.

```
    }
}
```

در این مثال، اکشن متد Index هیچ نیازی به پارامتر ندارد چون یک متد ساده است و قرار است به کاربر تنها یک View برگرداند. اما همه چیز به همین جا ختم نمی‌شود چون برخی Requestهای ارسالی، دارای پارامترهایی هم هستند. **به عنوان مثال:** برای نمایش جزئیات یک کارمند، Request باید پارامتر EmployeeId را بفرستد. پس با توجه به این موضوع، یک اکشن‌متد باید بتواند برای انجام برخی بیزینس‌ها، پارامترهایی را در ورودی خود بپذیرد.

لطفا همانند مثال پایین، در EmployeeController یک اکشن‌متد به نام GetEmployee اضافه نمایید.

```
using Microdev.ASPNETCore.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController : Controller
    {
        public List<Employee> Employees { get; private set; }

        public EmployeeController()
        {
            Employees = new List<Employee>
            {
                new Employee{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    Salary=1000000
                },
                new Employee{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    Salary=1000000
                },
            };

        }

        public IActionResult GetEmployee(int employeeId)
        {
```

لیستی از کارمندان که در سازنده این کنترلر مقداردهی شده است.

```
            var employee = Employees.FirstOrDefault(x => x.EmployeeId ==
           employeeId);
               return Json(employee);
        }

        [HttpPost]
        public IActionResult CreateEmployee(Employee employee)
        {
            if (ModelState.IsValid)
            {
                // بیزینس شما

                return View(employee);
            }
            else
            {
                return BadRequest(ModelState);
            }
        }

    }
}
```

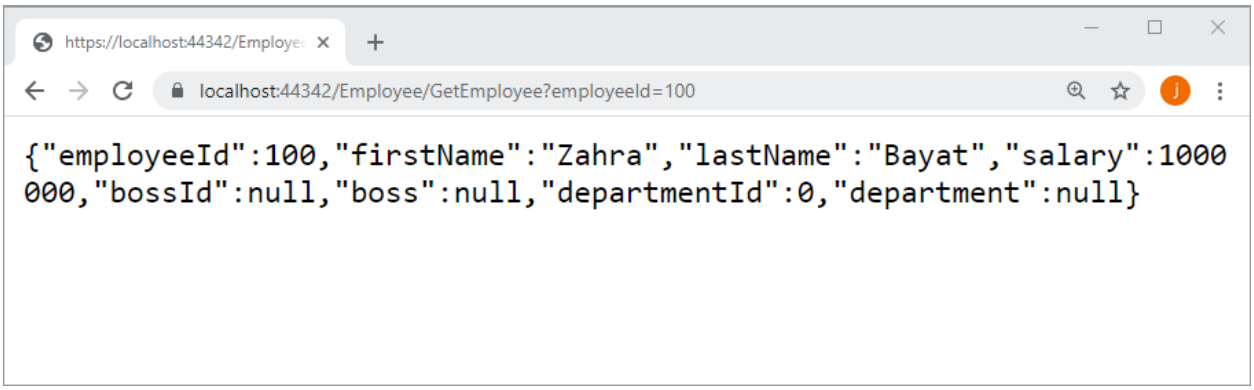
برگرداندن اطلاعات اولین کارمندی که شناسه آن با ورودی یکی است.

توجه داشته باشید قبل از اجرای اپلیکیشن، متغیر ASPNETCORE_ENVIRONMENT را به Development تغییر دهید.

حالا اپلیکیشن را اجرا کنید و URL زیر را در مرورگر وارد نمایید:

https://localhost:44342/Employee/GetEmployee?employeeId=100

Request بالا، متد GetEmployee را اجرا و ۱۰۰ را به عنوان یک پارامتر به این متد پاس می‌دهد.



حالا اکشن‌متد تصمیم می‌گیرد چه Responseی تولید کند. **به عنوان مثال**: همانگونه که در تصویر بالا می‌بینید، یک اکشن‌متد می‌تواند ViewResultی برگرداند که سبب اجرای Razor view و تولید HTML شود.

مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter3/Sample1

نگران نباشید، ما بزودی درمورد ViewResult بحث خواهیم کرد.

نکته!!

کلاس معمولی یا POCO می‌تواند با استفاده از Attribute [Controller] به عنوان Controller شناخته شود. و همچنین می‌توان به کلاس‌هایی که دارای پسوند "Controller" هستند Attribute [NonController] را اعمال کنید تا این قرارداد نفی شود. برای مثال: کلاس Departmant یک کلاس معمولیست که با Attribute [Controller] به یک Controller تبدیل شده است.

```
[Controller]
public class Department
{
 // ...
}
```

و در مثال زیر کلاس EmployeeController با استفاده از Attribute [NonController] به یک کلاس معمولی تبدیل شده و دیگر نمی‌تواند به عنوان یک Controller مورد استفاده قرار گیرد.

```
[NonController]
 public class EmployeeController: Controller
 {
  // ...
 }
```

ما می‌توانیم یک اکشن‌متد را هم با استفاده از Attribute [NonAction] محدود کنیم تا به عنوان اکشن‌متد عمل نکند. همچنین برای تغییر نام اکشن‌متد در زمان اجرا، می‌توانیم [ActionName ("Your Name")] را اعمال نماییم.

```
 public class EmployeeController : Controller
 {
```

```
    [ActionName("Employee Detail")]
    public IActionResult GetEmployee(int employeeId)
    {
        //Your business…
         return View(model);
     }

     [NonAction]
     public IActionResult CreateEmployee(int employeeId)
    {
     //Your business…
      return View(model);
    }


}
```

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Routing چیست؟**
- ✓ **سیستم Routing چگونه کار می‌کند؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: Explain Middleware in ASP.NET Core?

Q2: How does an MVC system work?

Q3: What are the benefits of Middleware?

Q4: What is the difference between middleware and HTTP module ?

Q5: Can you explain Model, Controller and View in MVC?

Q6: What are Actions in MVC?

Q7: What are Validation Annotations?

Q8: What is the use of ViewModel in MVC?

Q9: What are the different validators in ASP.NET?

Q10: What is Separation of Concerns in ASP.NET MVC?

Quiz

Q1: MVC stands for _____.

1. Model, Vision & Control
2. Model, View & Controller
3. Model, ViewData & Controller
4. Model, Data & Controller

Q2: ASP.NET Core implements MVC using a middleware called_____.

1. UseExceptionHandler
2. UseRouting and UseEndpoints
3. UseWelcomePage
4. UseMvcWithDefaultRoute

Q3: Which of the following is TRUE?

1. The controller redirects incoming request to model.
2. The controller executes an incoming request.
3. The controller controls the data.
4. The controller renders html to view.

Q4: _____ are the main entry point and handle requests initiated from user interaction.

1. Actions
2. Controllers
3. Views
4. Program.cs

Q5: Which of the following is TRUE?

1. Action method can be static method in a controller class.
2. Action method can be private method in a controller class.
3. Action method can be protected method in a controller class.
4. Action method must be public method in a controller class.

Q6: The _____ uses the data contained in the model to generate the UI.

1. Action
2. View
3. Controller
4. Model

Q7: _____ attributes can be used for data validation in MVC.

1. DataAnnotations
2. Fluent API

3. DataModel
4. HtmlHelper

Q8: _____ updates the model and selects the appropriate view.

1. Action
2. View
3. Controller
4. DataAnnotations

Q9: _____ is the most important piece of middleware that serves as the main entry point for users to interact with your app.

1. ExceptionHandlerMiddleware
2. StaticMiddleware
3. WelcomePageMiddleware
4. Routing and Endpoints

Answers

1-Correct Answer: Model, View & Controller

2-Correct Answer: UseRouting and UseEndpoints

3-Correct Answer: The controller executes an incoming request.

4-Correct Answer: Controllers

5-Correct Answer: Action method must be public method in a controller class.

6-Correct Answer: View

7-Correct Answer: DataAnnotations

8-Correct Answer: Controller

9-Correct Answer: MvcMiddleware

خلاصه فصل

- ✓ Model-View-Controller (MVC)یک الگوی طراحی نرم‌افزار است که از آن برای پیاده‌سازی وب اپلیکیشن‌ها استفاده می‌شود.
- ✓ MVC Design pattern از سه کامپوننت اصلی Model, View, Controller تشکیل شده است.
- ✓ Model: Modelها داده‌هایی که باید نمایش داده شوند را نگه می‌دارند.
- ✓ یک Model می‌تواند یک Object ساده یا یک Complex Type باشد.
- ✓ View: قالبی است که داده‌های ارائه شده توسط Model را نشان می‌دهد.
- ✓ Controller: Model را آپدیت می‌کند و View مناسب را انتخاب می‌نماید.
- ✓ یک Controller از تعدادی اکشن‌متد تشکیل شده است که می‌تواند در پاسخ به یک Request فراخوانی شود.
- ✓ Controllerهای ASP.NET Core معمولا از کلاس Controllerیا ControllerBase ارث‌بری می‌کنند و نام آن‌ها به کلمه Controller ختم می‌شوند.
- ✓ کلاس پایه Controller متدهای بسیاری را برای ایجاد ActionResult در اختیار شما قرار می‌دهد.
- ✓ اکشن‌متدها می‌توانند پارامترهایی داشته باشند که مقادیر آنها از Propertyهای Request ورودی گرفته می‌شود.
- ✓ DataAnnotationها به شما امکان می‌دهند مقادیر مورد انتظار را تعریف کنید.
- ✓ اعتبارسنجی به طور خودکار پس از Model Binding اتفاق می‌افتد، اما شما باید با استفاده از ویژگی ModelState، نتیجه اعتبارسنجی رابه طور دستی بررسی کنید و مطابق‌با اکشن‌متد خود عمل نمایید.

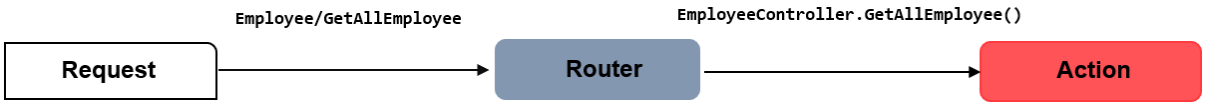
فصل چهارم: سیستم Routing

آنچه خواهید آموخت:

- Routing چیست؟
- مزایای سیستم Routing
- سیستم Routing چگونه کار می‌کند؟
- قسمت‌های یک الگوی مسیر
- روش‌های Mapping
- Conventional Routing و Attribute Routing
- برنامه MVC با چندین مسیر
- Constraint بر روی مسیرها
- چطور Constraintها را اعمال کنیم؟

Routing چیست؟

Routing فرآیند Mapping یک HTTP Request ورودی به یک اکشن-کنترلر خاص است. بنابراین وظیفه-ی Routing تطابق بخش‌های یک URL به یک متد در کنترلر است.



مزایای سیستم Routing

- **سیستم Routing سبب انعطاف‌پذیری در اپلیکیشن و ایجاد یک مدیریت قدرتمند بر روی URLها می‌شود.**
- **با این سیستم، شما به آسانی می‌توانید URLها را جهت Map شدن به اکشن‌متدهای دلخواه ایجاد کنید.**
- **Routing شما را قادر می‌سازد تا به طور صریح و بدون اتصال به یک طرح یا ساختار فایل، URLهای موردنیاز خود را جهت هدایت اپلیکیشن تعریف نمایید.**
- **این سیستم، اکشن‌متدها را از URLها جدا می‌کند و باعث می‌شود بدون دست زدن به اکشن‌متدها، URLهای مورد نیاز اپلیکیشن را تنها با تغییر سیستم مسیریابی تغییر دهید.**
- **مورد دیگر برای استفاده قاطع از مسیریابی، این است که URLها User Friendlyتر می‌شوند.**

سیستم Routing چگونه کار می‌کند؟

سیستم Routing به عنوان بخش مهمی از دیزاین پترن MVC در ASP.NET Core شناخته شده است زیرا Request ورودی را به یک اکشن‌متد خاص در یک Controller متصل می‌نماید.

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
   });
```

مسیرها با استفاده از یک عبارت لامبدا (به صورت آرگومان) به متد UseEndpoints پاس داده می‌شوند. این عبارت، مسیرها را با استفاده از Objectی که اینترفیس IRouteBuilder را پیاده‌سازی کرده، تعریف می‌کند.

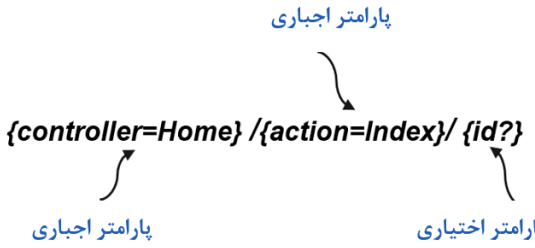
قسمت‌های یک الگوی مسیر

یک مسیریاب یک الگو مسیر را به تعدادی بخش تقسیم می‌کند، که به طور معمول هر بخش توسط کاراکتر / جدا می‌شود.

- ✓ **بخش اول: با نام Controller انتخاب شده، Map می‌شود.**
- ✓ **بخش دوم: با نام اکشن‌متد موردنظر، Map خواهد شد.**
- ✓ **بخش سوم: یک پارامتر اختیاری به نام id را مشخص می‌کند که وارد کردن آن اجباری نیست اما در صورت وجود، Router مقداری برای پارامتر id ضبط می‌کند.**

نکته!!

پارامترهای {controller} و{action} اجباری هستند و شما نمی‌توانید یک پارامتر اختیاری را قبل از یک پارامتر اجباری قرار دهید، زیرا مسیریاب از مقادیر اجباری جهت تصمیم‌گیری استفاده می‌کند و هیچ راهی برای تعیین پارامتر اجباری وجود ندارد.



هنگامی که یک URL صدا زده می‌شود، موتور مسیریابی تلاش می‌کند تا متن URL را (در مکان{controller}) با یک Controller تعریف شده در وب اپلیکیشن، مطابقت دهد. اگر موتور مسیریابی نتواند چیزی برای تطبیق بیابد، خطا نمایش داده خواهد شد. در غیر این صورت، نتیجه مسیریابی، یک اکشن‌متد و Controller مرتبط با آن خواهد بود.

نکته!!

شـــما می‌توانید برای Map کردن اکشـــن‌متدها به URLهای مختلف، الگوهای متفاوتی از مســـیریابی تعریف کنید.

روش‌های Mapping

Mapping (بین URLها، Controllerها و اکشن‌متدها) یا در کلاس Startup.cs و یا با استفاده از Attribute Route تعریف خواهد شد. بنابراین دو راه برای ایجاد مسیر در یک اپلیکیشن ASP.NET Core MVC وجود دارد:

۱) **استفاده از Conventional Routing.**

۲) **قرار دادن Attribute Routing بر روی Controllerها.**

هر کدام از تکنیک‌های فوق، URLهای مورد انتظار شما را با استفاده از الگوهای مسیر تعریف می‌کنند.

Conventional Routing

معمولا در بیشتر اپلیکیشن‌ها، مسیرهای مبتنی بر Convention تعریف می‌شوند و شما تا زمانی می‌توانید از این روش استفاده کنید که، ساختار کدهایتان مطابق با مسیرهای تعریف شده باشد.

این روش درک URLها را قابل فهم‌تر کرده و بسیاری از وب اپلیکیشن‌های مبتنی بر HTML- MVC معمولا از این روش برای مسیریابی استفاده می‌کنند.

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
        {
```

شما باید AddControllersWithViews () را در اینجا اضافه نمایید.

```
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }

}
```

افزودن UseRouting() و MapControllerRoute().

مسیرها با استفاده از این عبارت lambda ایجاد می شوند.

این Conventionها تصمیم‌گیری و نگهداری وب اپلیکیشن‌ها را ساده‌تر می‌کند.

نکته!!

از آنجا که این الگوی مسیریابی مرسوم‌تر است، ما با این الگو به پروژه ادامه خواهیم داد.

Attribute Routing

علاوه بر Conventional Routing شما می‌توانید با قرار دادن Attribute [Route] در بالای اکشن‌متدهای خود، از مسیریابی مبتنی بر Attribute استفاده کنید. این روش انعطاف‌پذیری بیشتر دارد، زیرا شما می‌توانید به صراحت یک URL را برای یک اکشن‌متد خاص تعریف کنید.

این رویکرد شفاف‌تر، انعطاف‌پذیرتر و اغلب مفیدتر از رویکرد مبتنی بر Convention است و معمولا از این روش در Web APIها استفاده می‌شود.

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Routing;
using Microdev.ASPNETCore.Models;

namespace Microdev.ASPNETCore.Controllers
{
    [ApiController]
    [Route("[controller]")]
```

در زمان مسیریابی [controller] با عبارت Employee جایگزین خواهد شد.

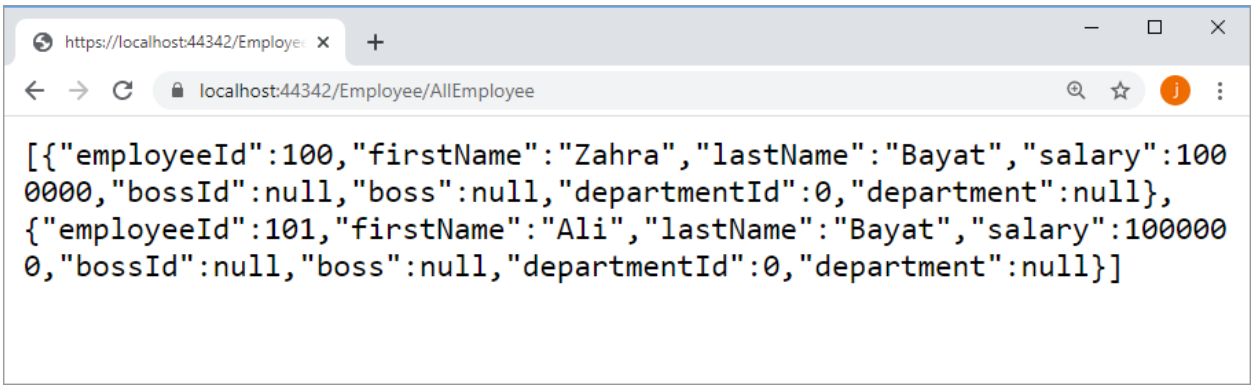
```
public class EmployeeController: Controller
{
    public List<Employee> Employees { get; private set; }
    public EmployeeController()
    {
        Employees = new List<Employee>
        {
            new Employee{
                EmployeeId = 100,
                FirstName = "Zahra",
                LastName = "Bayat",
                Salary=1000000
               },
            new Employee{
                EmployeeId = 101,
                FirstName = "Ali",
                LastName = "Bayat",
                Salary=1000000
            },
        };

    }
    public IActionResult GetEmployee(int employeeId)
    {
      var employee=  Employees.FirstOrDefault(x => x.EmployeeId ==
      employeeId);
        return OK(employee);
    }

    [Route("AllEmployee")]
    public IActionResult GetAllEmployee()
    {
        return Json(Employees);
    }

   [HttpPost]
    public IActionResult CreateEmployee(Employee employee)
    {
        if (ModelState.IsValid)
        {
            // بیزینس شما

            return View(employee);
        }
        else
        {
            return BadRequest(ModelState);
        }
    }
```

زمانیکه URL –– "/Employee/AllEmployee" درخواست می‌شود، متد GetAllEmployee اجراخواهد شد.

```
    }
}
```

اکنون برنامه را اجرا و URL زیر را در مرورگر وارد کنید:

https://localhost:44342/Employee/AllEmployee



نکته!!

شما در اپلیکیشن‌های خود می‌توانید Conventional Routing را با Attribute Routing ترکیب کنید، اما معمولا برای کنترلرهای MVC، از Conventional Routing و جهت نوشتن کنترلرها Web API از Attribute Routing استفاده خواهد شد.

نتیجه‌گیری: Conventional Routing برای اپلیکیشن‌های سنتی(که خروجی HTML دارند) مناسب است و Attribute Routing اپلیکیشن‌های Web API را خواناتر می‌کند.

مسیر پروژه نمونه انجام شده در Github:

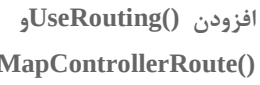
https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter4/Sample1

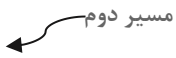
برنامه MVC با چندین مسیر

یک برنامه MVC معمولا برای پردازش URLهای ورودی باید مسیرهای متعددی داشته باشد. با این مسیرها شما می‌توانید در تعریف الگوهای خود، هر تعداد Route Parameter که نیاز است تعیین کنید و در زمان Model Binding به این پارامترها دسترسی داشته باشید.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
           services.AddControllersWithViews();
        }

       public void Configure(IApplicationBuilder app, IWebHostEnvironment
        env)
        {
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "employee",
                    pattern:"{controller=Employee}/{action=GetAllEmployee}/
                    {id?}");


                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
             });

        }
    }

}
```



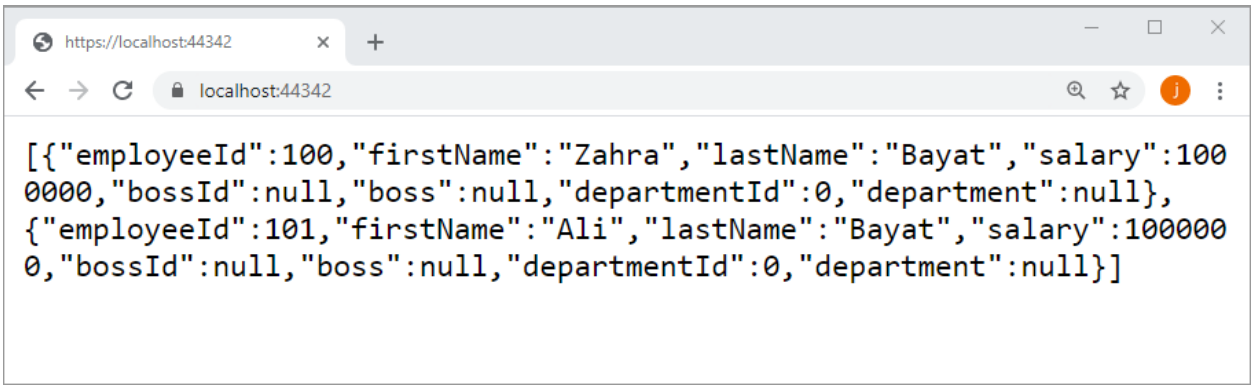


در مثال فوق، سیستم مسیریابی تلاش می‌کند تا یک URL را با اولین الگوی تعریف شده مطابقت دهد و در صورت عدم تطابق، با الگوی بعدی این روند ادامه می‌یابد.

توجه داشته باشید، مسیرهای خاص در ابتدای این لیست قرار گیرد و ترتیب مسیرها را هم در نظر بگیرید.

حالا Attributeهای اضافه شده به EmployeeController را بردارید و اپلیکیشن را اجرا نمایید.

همانطور که می‌بینید، اپلیکیشن به مسیر Employee / GetAllEmployee هدایت می‌شود.



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter4/Sample2

Constraint بر روی مسیرها

Routing در مورد نوع داده‌های که پارامترهای مسیر می‌گیرند، هیچ اطلاعی ندارد و تنها کاری که انجام می‌دهد این است که، پارامترهای مسیر را تطبیق دهد. بنابراین می‌توان گفت: با توجه به قالب پیش‌فرض `"{controller=Home}/{action=Index}/{id?}"`، آدرس‌های زیر همگی مطابقت دارند:

- **/Home/Edit/test**
- **/Home/Edit/123**
- **/1/2/3**

تمامی این آدرس‌ها با توجه به syntaxی که در الگو تعریف شده، کاملا معتبر هستند، اما همه‌ی شما می‌دانید برخی از این آدرس‌ها هیچ گونه خروجی نخواهند داشت.

پس چاره چیست؟؟

برای جلوگیری از این مشکل، می‌توان Constraintها را به یک قالب مسیر اضافه نمود.

وظیفه‌ی Constraint چیست؟

Constraintها، URLهایی که بلـیک قلالب مســـ یر مطابقت دلوند را محدود می‌کند. بنابراین شـــما از این پس می‌توانید، زمانی که یک Segment مشخص نشده بود، جهت Valid بودن یک پارامتر مسیر، مقادیر پیش‌فرض را تعیین کنید.

چطور Constraintها را اعمال نماییم؟

با اســـتفاده از دو نقطه می‌توانید یک پارامتر مســـیر را محدود کنید. **به عنوان مثال:** {id: int} محدودیت عدد بودن را به پارامتر id اضافه می‌کند و از این پس پارامتر id نمی‌تواند مقداری غیر از عدد دریافت کند.

شـما همچنین می‌توانید محدودیت‌های پیشـرفته‌تری را بررسـی کنید، **به عنوان مثال:** اضـافه کردن محدودیت حداقل مقدار یک عدد یا حداکثر طول یک رشته.

و در پایان اینکه، شما می‌توانید چندین محدودیت را با استفاده از کلون و علامت ؟ ترکیب کنید. در جدول پایین چند مثال ذکر شده است.

Type	Constraint
Int	{qty:int}
Guid	{id:guid}
Decimal	{cost:decimal}
length(value)	{name:length(6)}
min(value)	{age:min(18)}
optional int	{qty:int?}
optional int max(value)	{qty:int:max(10)?}

حالا زمانیکه Router، یک URL را با یک قالب مسـیر منطبق داد، باید ابتدا محدودیت‌ها را بررسـی کند تا همه پارامترها معتبر باشند. در صورت معتبر بودن پارامترها، Router تلاش می‌کند تا یک اکشن‌متد مناسب بیابد.

نکاتی درباره Constraintها:

۱) برای انعطاف‌پذیری بیشـــتر در اعمال محدودیت‌ها، می‌توان از Regex اســـتفاده نمایید. Regex محدودیتی است که یک Segment را با عباراتی منظم مطابقت می‌دهد.

```
app.UseEndpoints(endpoints =>
 {
     endpoints.MapControllerRoute(
         name: "employee",
        pattern:
            "{controller:regex(^E.*)=Employee}/{action=GetAllEmployee}
            /{id?}");
 });
```

عبارت regex

در مثال بالا، تنها URLهایی که Controller آن‌ها با حرف E آغاز می‌شوند، تطبیق داده خواهد شد.

۲) اگر برای یک بخش نیاز به محدودیت‌های چندگانه باشد، باید آنها را با یک دو نقطه به هم متصل کنید.

```
app.UseEndpoints(endpoints =>
  {
     endpoints.MapControllerRoute(
      name: "employee",
      pattern: "{controller=Employee}/{action=GetAllEmployee}
    /{id:alpha:minlength(3)?}");
  });
```

محدودیت چندگانه

۳) شما می‌توانید Attribute [Route] را هم به همین صورت محدود کنید. در مثال پایین این مورد را می‌بینید.

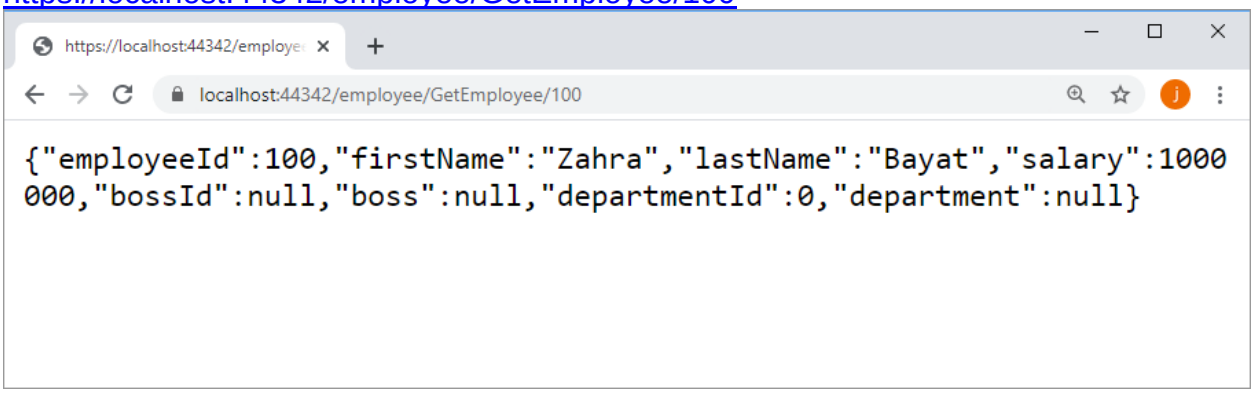
```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Routing;
using Microdev.ASPNETCore.Models;

namespace Microdev.ASPNETCore.Controllers
{
    [Route("[controller]")]
    public class EmployeeController: Controller
    {
        public List<Employee> Employees { get; private set; }

        public EmployeeController()
        {
            Employees = new List<Employee>
            {
                new Employee{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    Salary=1000000
                },
                new Employee{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    Salary=1000000
                },
            };

        }
      [Route("[action]/{employeeId:int}")]
```

employeeId باید عدد باشد.

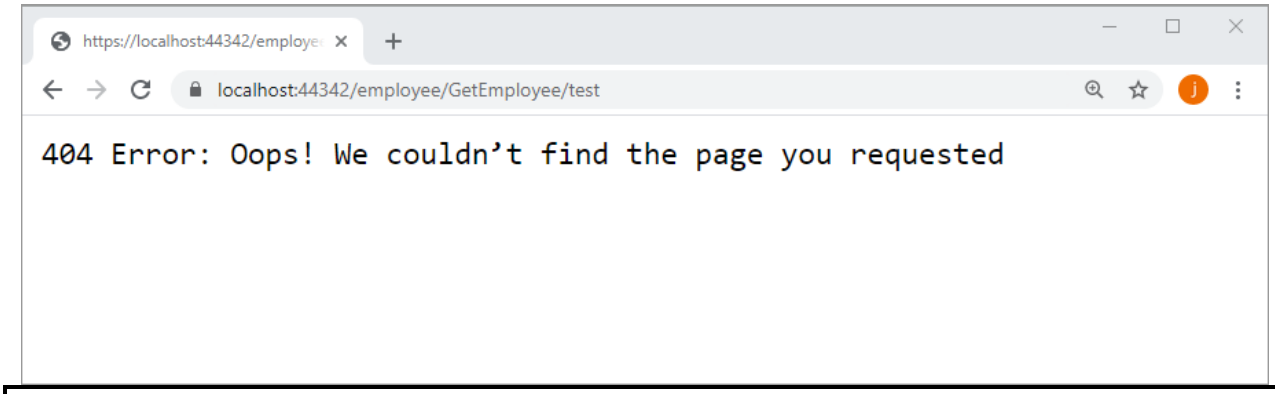
```
        public IActionResult GetEmployee(int employeeId)
        {
            var employee = Employees.FirstOrDefault(x => x.EmployeeId ==
            employeeId);
            return Json(employee);
        }

        public IActionResult GetAllEmployee()
        {
            return Json(Employees);
        }

        [HttpPost]
        public IActionResult CreateEmployee(Employee employee)
        {
            if (ModelState.IsValid)
            {
                // بیزینس شما

                return View(employee);
            }
            else
            {
                return BadRequest(ModelState);
            }


        }
    }

}
```

حالا اگر اپلیکیشن را اجرا کنید و پارامتر employeeId عدد باشد، نتیجه درست است و اگر عدد نباشد، نتیجه خطا برمی گردد.

https://localhost:44342/employee/GetEmployee/100



https://localhost:44342/employee/GetEmployee/test



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter4/Sample3

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Razor چیست؟**
- ✓ **چطور از Razor استفاده کنیم؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What is Routing in ASP.NET Core MVC?

Q2: Where is the route mapping code written in Asp.NET Core?

Q3: Can we map multiple URL's to the same action?

Q4: Explain attribute-based routing in MVC?

Q5: What is the advantage of defining route structures in the code?

Q6: How to enable Attribute Routing?

Q7: What are the components required to create a route in MVC?

Q8: Can we add constraints to the route?

Q9: How to apply Routing Constraints in ASP.net MVC?

Q10: How to enable Convention Routing?

Quiz

Q1: Which of the following is a default route pattern in MVC?

1. "/{action}/{controller}/{id}"
2. "{controller}/{id}"
3. "{controller}/{action}/{id}"
4. "{controller}/{action}”

Q2: Which of the following default class is used to configure all the routes in MVC?

1. FilterConfig
2. RegisterRouteConfig
3. RouteConfig
4. MVCRoutes

Q3: _____ is a key part of the MVC design pattern in ASP.NET Core

1. Model
2. RegisterRouteConfig
3. Routing
4. Controller

Q4: Which middleware will attempt to match a request’s path to a configured route.

1. ExceptionHandlerMiddleware
2. StaticMiddleware
3. WelcomePageMiddleware
4. RoutingMiddleware

Q5: Which method is equivalent to *UseMvc():*

1. UseExceptionHandler();
2. UseStatic();
3. UseWelcomePage();
4. UseRouting(); UseEndpoints();

Q6: The first segment of the URL maps to the name of the selected _____Model.

1. Action
2. View
3. Controller
4. Model

Q7: Attribute routing for _____controllers where possible.

1. Web API
2. Conventional

3. All
4. None of the above

Q8: Constraints can be defined in a route template for a given route parameter using:

1. (a colon)
2. optional mark (?)
3. regex
4. All of the above

Q9: Attribute routing is enabled when you call the _____method.

1. UseExceptionHandler();
2. UseStatic();
3. UseRouting(); UseEndpoints();
4. UseIdentity();

Q10: The_____based routes are defined globally for your application

1. Convention
2. Attribute
3. MVC
4. Option 1 and 2

Answers

1-Correct Answer: "{controller}/{action}/{id}"

2-Correct Answer: RouteConfig

3-Correct Answer: Routing

4-Correct Answer: Routing

5-Correct Answer: UseRouting(); UseEndpoints();

6-Correct Answer: Controller

7-Correct Answer: Web API

8-Correct Answer: (a colon)

9-Correct Answer: UseRouting(); UseEndpoints();

10-Correct Answer: Convention

خلاصه فصل

- ✓ **Routing فرایند Map کردن یک درخواست ورودی به یک اکشن‌متد است که در نتیجه یک Response تولید می‌شود.**
- ✓ **الگوهای مسیر ساختار URLهای مشخص شده در اپلیکیشن را تعریف می‌کنند.**
- ✓ **مسیرها را می‌توان یا با استفاده از Conventional Routing در سطح عمومی تعریف کرد یا با استفاده از Attribute Routing در بالای یک اکشن‌متد قرار داد.**
- ✓ **یک اپلیکیشن می‌تواند مسیرهای متفاوتی داشته باشد که در این صورت Router تلاش خواهد کرد اولین مسیری که با URL ورودی منطبق باشد را انتخاب کند.**
- ✓ **پارامترهای مسیر می‌توانند Constraintهایی داشته باشند که مقادیر ورودی را محدود کنند.**

فصل پنجم: رندر کردن HTML با استفاده از Razor view

آنچه خواهید آموخت:

- View چیست؟
- Razor چیست؟
- نحوه استفاده ازRazor
- روش‌های انتقال داده به View
- نوشتن عبارات با سینتکس Razor
- Layout چیست؟
- مزایای Layout
- چطور از Layout استفاده کنیم؟
- Section چیست؟
- Partial view چیست؟
- ایجاد یک Partial View
- استفاده از Partial Viewهای Strongly Type
- ViewStart چیست؟
- ViewImports چیست؟

View چیست؟

به طور کلی، کاربران دو نوع تعامل با اپلیکیشن شما دارند:

- **یا اطلاعاتی که اپلیکیشن نمایش می‌دهد را می‌خوانند.**
- **و یا داده‌هایی را به آن ارسال می‌کنند.**

مسئولیت نمایش داده‌های اپلیکیشن در ASP.NET Core MVC، برعهده Viewهاست و زبان Razor شامل ساختارهایی است که ایجاد هر دو نوع اپلیکیشن را آسان می‌کند. Viewها جهت تولید UI از سینتکس Razor استفاده می‌کنند.

Razor چیست؟

Razor یک View Engine است که اطلاعات Razor templateها را پردازش می‌کند.

Razor template چیست؟

Razor template ترکیبی از HTML و کدهای سی‌شارپ است، که با استفاده از HTML می‌توان مشخص نمود چه چیزی باید به مرورگر ارسال شود و کدهای سی‌شارپ هم برای تولید داده‌های داینامیک است.

> **نکته!!**
>
> **استفاده از سی‌شارپ به این معنی است که شما می‌توانید به صورت داینامیک HTML نهایی را تولید نمایید. برای مثال:**
>
> - **نمایش کاربر فعلی**
> - **پنهان کردن لینک‌هایی که کاربران فعلی به آن دسترسی ندارند.**
> - **تولید Button برای هر آیتم در یک لیست و... .**

چرا یادگیری Razor مهم است؟

در یک وب‌اپلیکیشن، تولید محتوای داینامیک امری ضروریست و Razor با استفاده از عبارات سی‌شارپ این هدف را محقق کرده و باعث سادگی کار با ASP.NET Core MVC شده است.

اهداف طراحی Razor عبارتند از: ایجاد کارآیی بیشـــتر و یادگیری ســـاده‌تر، که به طور قطع می‌توان گفت: به این اهداف رسیده است.

نحوه استفاده از Razor

برای نشان دادن نحوه کارکرد Razor، بیاید با هم پروژه اولیه کتاب را تکمیل کنیم:

۱) انتخاب یک View درون Controller

Controllerهای شـــما، همیشـــه باید کدهای HTML را با اســـتفاده از Viewها ایجاد کنند، بنابراین اکشن‌متدها برای ایجاد یک View از شیی به نام ViewResult استفاده می‌کنند.

متد View در کلاس پایه Controller قرار گرفته که به ســـادگی از یک View Model عبور کرده و یک اکشـــن‌متـد را انتخـاب می‌کنـد. این متـد یـک ViewResult برمی‌گردانـد و برای پیـدا کردن View از Conventionها استفاده می‌نماید.

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
 {
     public IActionResult Index()
     {
         var employeeNames = new List<string> { "Zahra", " Ali", " Sara" };

         return View(employeeNames);
     }

    public string Error(int id)
     {
         return $"{id} Error: Oops! We couldn't find the page you
          requested";
      }
 }
```

با ارث بری از کلاس پایه Controller می‌توانید از متدView استفاده کنید.

من برای ســادگی، داده‌های لیســت را هاردکد کردم و هیچ داده‌ی داینامیکی وجود ندارد.

متد View یک ViewResult برمی‌گرداند.

نکته!!

در این مثال متد View، از نام اکشن‌متد Index به عنوان نام View استفاده می‌کند.

با توجه به اینکه، نام کنترلر ما HomeController است و نام اکشن‌متد هم Index می‌باشد، به طور پیش‌فرض Razor engine دنبال مســیر Views / Home / Index.cshtml می‌گردد، اما با این حال، شـما می‌توانید به صـــراحت نام View مورد نظر خود را در ورودی متد View قرار دهید. **به عنوان مثال:** در اکشـــن‌متد Index

می‌توانید ورودی متد View را این گونه بنویسید: View ("EmployeeList")، حالا Razor engine مستقیما به فایل EmployeeList.cshtml نگاه می‌کند.

شما حتی می‌توانید، در ورودی متد View، مسیر کامل فایل View را نسبت به مسیر ریشه اپلیکیشن قرار دهید (مانند View ("Views / EmployeeList.cshtml")) تا از این پس هر زمان اپلیکیشن این اکشن‌متد را صدا زد، به دنبال مکان View / EmployeeList.chtml بگردد.

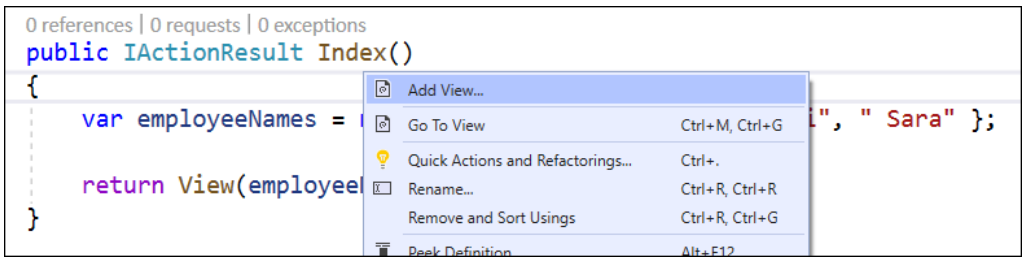
نکته!!

ممکن است وسوسه شوید و به صورت صریح مسیر فایل View را در همینجا ذکر کنید، اگر چنین تصمیمی دارید، هرگز این کار را انجام ندهید زیرا این رفتار به برنامه‌نویسان بعدی نیز، که به کد شما نگاه می‌کند انتقال می‌یابد و کدهای شما، دیگر حس خوبی را منتقل نخواهد کرد.

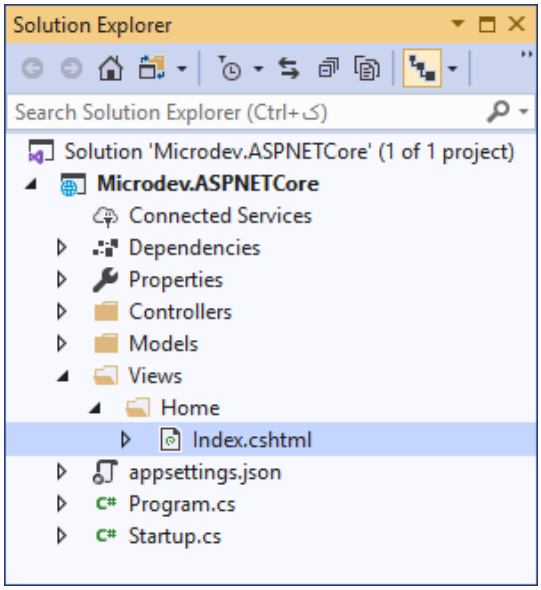
۲) ایجاد یک Razor view

در ASP.NET Core، هر بار که نیاز به نمایش HTML Response به کاربر باشد، باید از یک View استفاده کنید. به طور قراردادی، در ASP.NET Core، Viewها در مسیر Views و در یک Folderی همنام با نام Controller قرار می‌گیرند. نام فایل View هم به این شکل است: **نام اکشن‌متد با یک پسوند cshtml**.

حالا برای ایجاد یک View، می‌توانید بر روی اکشن‌متد خود راست کلیک نمایید و سپس ...Add View را انتخاب کنید. انجام این کار یک Folder با نام Views در مسیر پروژه ایجاد و سپس درون این Folder یک Folder دیگر با نام Home ایجاد و در پایان یک View با نام Index اضافه می‌کند.



حالا ساختار Solution را ببینید.



مانند کد پایین محتوای فایل **Index.cshtml** را تغییر دهید:

```
@model List<string>
<h1>EmployeeNames: </h1>
<ul>
    @for (int i = 0; i < Model.Count; i++)
    {
        <li>@i - @Model[i]</li>
    }
</ul>
```

@model یک لیست string را نشان می‌دهد.

برای نوشتن نام یک کارمند در خروجی HTML، از یک عبارت Razor استفاده می‌شود.

همانطور که می‌بینید، دستور @model در این View، نشان می‌دهد که من برای تولید HTML نهایی، یک لیست string ارائه داده‌ام. این بدین معنی است که دستور @model به Razor می‌گوید که انتظار دارم کدام مدل به این View داده شود.(این مدل از طریق پراپرتی Model قابل دسترس است.)

در این مثال ویژگی‌های مختلف Razor به وضوح نشان داده شده است. (ترکیبی از HTML و ساختار سی‌شارپ که برای ایجاد HTML داینامیک استفاده می‌شود.)

قبل از اجرای اپلیکیشن فایل Stutup.cs را مانند کد زیر تغییر دهید:

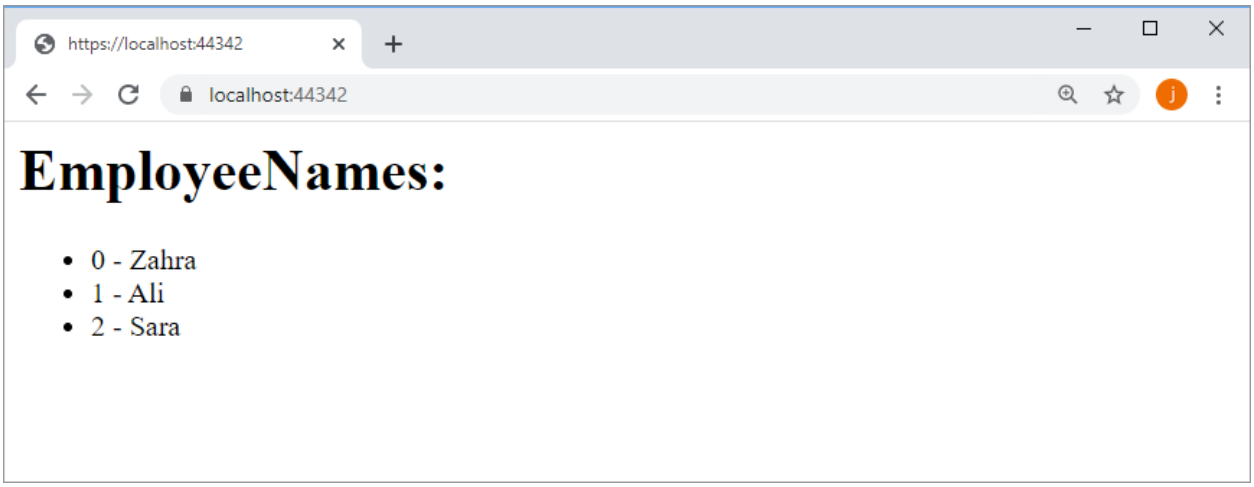
```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
           services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
           env)
        {
            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

         app.UseRouting();

         app.UseEndpoints(endpoints =>
         {
             endpoints.MapControllerRoute(
                 name: "default",
                 pattern: "{controller=Home}/{action=Index}/{id?}");

             endpoints.MapControllerRoute(
          name: "employee",
             pattern:"{controller=Employee}/{action=GetAllEmployee}/{id?}");
```

مسیر اول

مسیر دوم

```
            });
        }
    }
}
```

حالا اپلیکیشن را اجراکنید:



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter5/Sample1

روش‌های انتقال داده به View

در ASP.NET Core چندین روش برای انتقال داده‌ها از اکشن‌متد به View وجود دارد که در ادامه چند نمونه ذکر شده است:

۱) **View Model**: View Model یک Object با تعدادی Property‌ی است، که داده‌های موردنیاز View را فراهم می‌کند. با View Model می‌توانید مطمئن شوید که داده‌های مدل، همان چیزیست که انتظار دارید.

۲) **ViewData:** ViewData یک دیکشنری از String است که به اشیاء اشاره دارد و برای انتقال داده از Controller به View مورد استفاده قرار می‌گیرد.

۳) **ViewBag**: ViewBag یک Wrapper بر روی شـ ی ViewData اســت، که داینامیک بوده و به ســادگی می‌توان به Propertyهای آن اشــاره داشــت. بنابراین، اگر ترجیح می‌دهید یک دیکشــنری دینامیک داشته باشید، می‌توانید به جای ViewData از ViewBag استفاده کنید.

انتقال داده با View Model

بهترین روش انتقال اطلاعات از Controller به View، اســتفاده از یک View Model اســت. View Model یک کلاس سفارشی، جهت نگهداری داده‌های مورد نیاز View می‌باشد.

یک اکشن‌متد، یک View را انتخاب می‌کند و سپس یک شی ViewResult که شامل View Model است را به آن پاس می‌دهد. حالا هنگامی که ViewResult اجرا می‌شــود، View اطلاعات را در یک قالب Razor قرار داده و محتوا را نمایش می‌دهد.

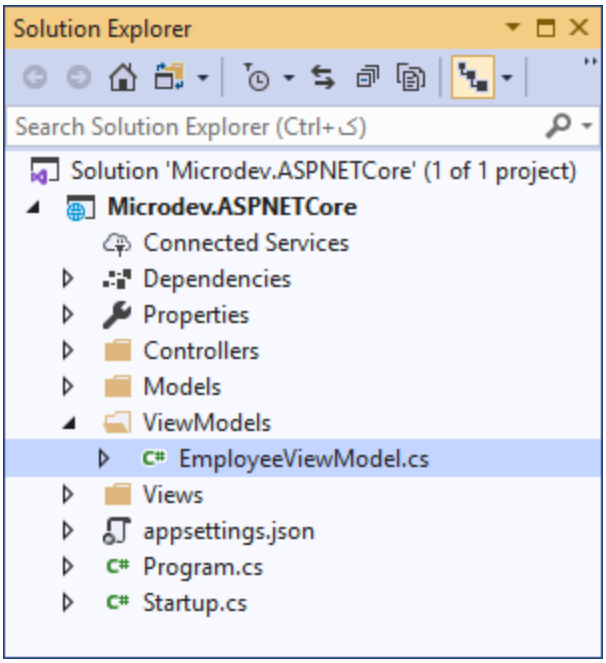
نکته!!

در View Model، Binding یکطرفه اسـت، این بدین معنیسـت که، View Model مقادیر را برای UI فراهم کرده و هنگامی که UI ساخته و Response ارسال شد، View Model از بین می‌رود.

ایجاد View Model

برای ایجاد یک View Model مراحل زیر را دنبال کنید:

۱) در مسیر پروژه، یک Folder به نام ViewModels ایجاد کنید.
۲) سپس درون این Folder یک کلاس به نام EmployeeViewModel اضافه نمایید.



محتوای کلاس **EmployeeViewModel**:

```
using System.Collections.Generic;
using Microdev.ASPNETCore.Models;
using System.ComponentModel.DataAnnotations;

namespace Microdev.ASPNETCore.ViewModels
{
    public class EmployeeViewModel
    {
        public int EmployeeId { get; set; }

        [Required]
        [MaxLength(64)]
        public string FirstName { get; set; }

        [Required]
        [MaxLength(64)]
        public string LastName { get; set; }

        [Required]
        public decimal Salary { get; set; }

        [Required]
        public string BossName { get; set; }

        [Required]
        [MaxLength(64)]
        public string DepartmentName { get; set; }
```

```
  }
}
```

حالا اکشن‌متد **Index** موجود در کنترلر **HomeController** را ویرایش کنید:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Controllers
{
    public class HomeController: Controller
    {
        public IActionResult Index()
        {
            List<EmployeeViewModel> viewModel = new List<EmployeeViewModel>
            {
                new EmployeeViewModel{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    DepartmentName = "Raveshmand",
                    Salary=10000000
                   },
                new EmployeeViewModel{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    DepartmentName = " Raveshmand ",
                    Salary=10000000

                },
            };
            return View(viewModel);

        }

        public string Error(int id)
        {
           return $"{id} Error: Oops! We couldn't find the page you
           requested";
        }

    }
}
```

ایجاد یک view model

این متد یک ViewResult ایجاد و ViewModel را به آن منتقل می‌کند.

نکته!!

از View Model به عنوان یک مکانیسم استاندارد جهت انتقال اطلاعات بین Controller و View، استفاده می‌شود.

حالا فایل Index.cshtm را مانند کدهای پایین ویرایش نمایید:

```
@using Microdev.ASPNETCore.ViewModels;
```

Namespace موردنظر را مشخص می کند.

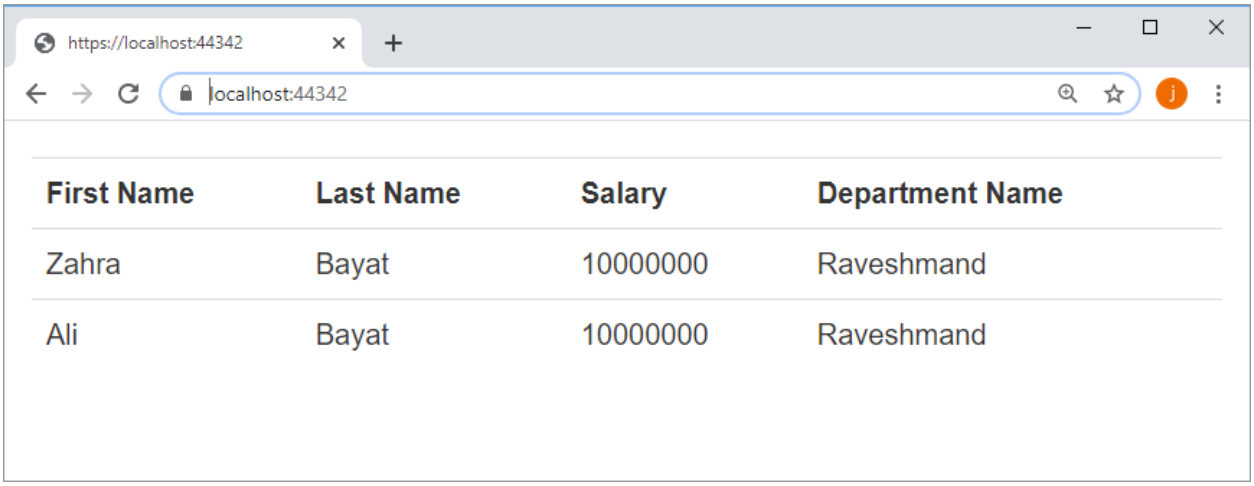
```
@model List<EmployeeViewModel>

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></scri
pt>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></s
cript>
<div class="Container">
<h3>Employee list: </h3>
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>

</div>
```

یکی از دستورالعمل‌های رایج، دستور using new Namespace@ است که آبجکت‌ها را در فضای‌نام تعریف شده، در دسترس قرار می‌دهد. هنگامی که این دستورالعمل را اضافه کردید، می‌توانید با استفاده از پراپرتی Model، به هر یک از داده‌های موجود در EmployeeModel دسترسی داشته باشید.

نکته!!

پراپرتی Model در هر نقطه از Viewبلیدبا حرف M بزرگ و عبارت @modelبا حرف m کوچک شروع شود.

حالا لطفا اپلیکیشن را اجرا کنید:



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter5/Sample2

انتقال داده با استفاده از ViewData

View Model برای انتقال داده بین Layoutها مناسب نیست، بنابراین یک رویکرد مرسوم برای این وضعیت، استفاده از ViewData است.

ViewData یک دیکشنری String است که اشاره به Object دارد و با استفاده از آن می‌توانید هرآنچه می‌خواهید، به یک View پاس دهید.

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Controllers
{
    public class HomeController: Controller
    {
        public IActionResult Index()
        {
            ViewData["Title"] = "Employee list:";
            List<EmployeeViewModel> viewModel = new List<EmployeeViewModel>
```

با استفاده از ViewData می‌توان داده را به به‌View پاس داد.

```
            {
                new EmployeeViewModel{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    DepartmentName = "Raveshmand",
                    Salary=10000000
                   },
                new EmployeeViewModel{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    DepartmentName = " Raveshmand ",
                    Salary=10000000

                },
            };

            return View(viewModel);

        }

        public string Error(int id)
        {
            return $"{id} Error: Oops! We couldn't find the page you
           requested";
        }

    }

}
```

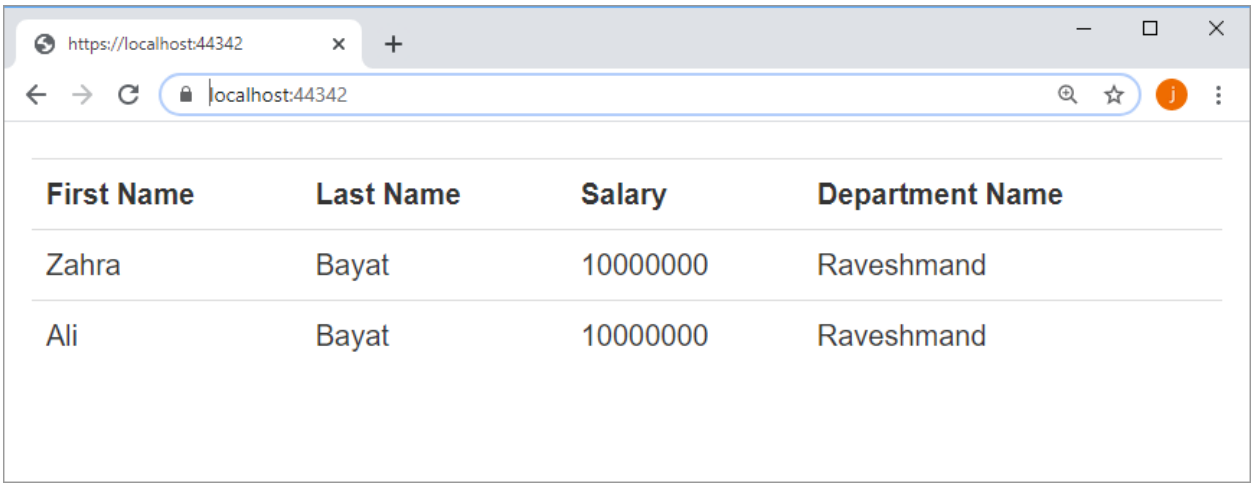
حالا در فایل **Index.cshtm** تغییرات پایین را اعمال نمایید.

```
@using Microdev.ASPNETCore.ViewModels;

@model List<EmployeeViewModel>

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></scri
pt>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></s
cript>

<div class="Container">
<h3> @ViewData["Title"]</h3>
```

و شما می توانید داده‌های خود را این طور بازیابی کنید:

```
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>

</div>
```

حالا لطفا اپلیکیشن را اجرا کنید:



نکاتی درباره ViewData:

- **شما می‌توانید مقادیر موجود در دیکشنری ViewData را از طریق خود View تنظیم کنید:**

```
@{
    ViewData["Title"] = "About";
}

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

داده‌ها می‌توانندبا اســتفاده از ViewDataبه View پاس داده شوند.

- **اگر عبارت سی‌شارپی که می‌خواهید اجرا کنید، چیزی است که نیاز به یک فاصله دارد، باید قبل از کدهای #C از علامت ()@ استفاده نمایید، تا Razor engine بداند کجا #C متوقف و کجا HTML شروع می‌شود.**

```
<p>The sum of 15 and 2 is: <i>@(15 + 2)</i></p>
```

- **هنگامی که کدهایتان را درون بلوک‌های کد قرار می‌دهید، باید #C معتبر باشـد، بنابراین در پایان دستور باید سمی کالن را اضافه کنید.**

```
@{
    ViewData["Title"] = "About";
}
```

انتقال داده با استفاده از ViewBag

روش دیگری برای دسترسی View به داده‌ها، استفاده از ViewBag است. ViewBag یک شی داینامیک است که به راحتی می‌توان به Propertyهای آن دسترسی داشت.

ViewBag این امکان را می‌دهد تـیک شـ ی داینامیک را از طریق کلاس کنترلر به View ارائه دهید و یکی از مهمترین مزیت‌های آن این است که ارسال چندین شی به View را آسان کرده است.

در مثال زیر، من یک Property در ViewBag با نام Message نامگذاری و مقداردهی کردم.

```
Public Iactionresult Test()
 {
    Viewbag.Message = "Hello";
    return View();
 }
```

در فایل Test.cshtm:

حالا برای خواندن داده‌ها در View، Propertyیی که در اکشن‌متد تنظیم شده را دریافت می‌کنم.

```
<p>The message is: @ViewBag.Message</p>
```

نوشتن عبارات با سینتکس Razor

✓ متغیرها

اضافه کردن متغیر در یک View بسیار ساده است، کافیست علامت@ را قبل از نام متغیر قرار دهید.

```
<P>It is now @DateTime.Now</P>
```

در این مثال، تگ HTML را باز کردیم و سپس یک فرمت تاریخ را با استفاده از علامت @ وارد نمودیم.

نکته!!

ما این نوع کد را یک Implicit Expression می نامیم.

✓ عبارات شرطی

از آنجا که Razor از زبان C# به عنوان زبان اسکریپت خود استفاده می‌کند، هر عبارت شرطی که در C# وجود دارد را می‌توان در Razor استفاده کرد.

```
@if (Model.Department!=null)
{
   <span>Depatment Name: @Model.Department.Name </span>
}
else
{
    <span>This Depatment is not valid</span>
}
```

✓ حلقه‌ها[۶]

حلقه‌ها یکی از رایج‌ترین موارد استفاده در سینتکس Razor است و Razor از تمامی ساختارهای حلقه در سی‌شارپ (از جمله for، foreach، do، while) پشتیبانی می‌کند. با حلقه‌ها می‌توانید بخش‌هایی از UI را پنهان یا HTML را برای هر آیتم در یک لیست ایجاد کنید.

استفاده از حلقه‌ها در قالب Razor تقریبا با C# یکسان است، شما تنها باید از علامت @ استفاده کنید.

```
@using Microdev.ASPNETCore.ViewModels;

@model List<EmployeeViewModel>

<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></scri
pt>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></s
cript>
<div class="Container">
```

[۶] Loops

```
<h3> @ViewData["Title"]</h3>

    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>

</div>
```

نحوه استفاده از حلقه foreach در سینتکس Razor

✓ بلوک‌های کد

بلوک‌های کد، بخش‌هایی از View هستند، که تنها شامل کدهای سی‌شارپ بدون هیچ‌گونه markupی می‌باشند.

```
@{
var title = "Code Blocks";
}
<h1>@title</h1>
```

بلوک های کد با {...}@ شروع می‌شوند و برای نوشتن هر تعداد خط کد، هیچ‌گونه محدودیتی وجود ندارد.

نکته!!

در داخل یک بلوک کد، باید تمامی قوانین زبان برنامه‌نویسی را دنبال کرد. در نظر داشته باشید که کد شما تنها باید در منطق View اضافه شود و شما نباید محاسبات و تغییرات مدل را در زمان View در دست بگیرید.

✓ کامنت[7]

هر زبان برنامه‌نویسی نیاز به یک روش برای اضافه کردن کامنت‌ها دارد. در Razor، شما از syntax زیر برای ایجاد یک کامنت استفاده می‌کنید.

[7] Comment

```
@* This is my comment. *@
```

هر کد یا عبارتی که بین دو علامت @*...*@ قرا بگیرد به عنوان کامنت درنظر گرفته می‌شود.

نکته!!

شما می‌توانید برای قرار دادن کامنت‌های خود، از بلوک‌های کد استفاده کنید.

```
@{
    // this is a comment inside of a code block
    /*
    * This is a multi-line comment inside a code block
    */
}
```

Layout چیست؟

هر HTML document دارای تعداد مشخصی المان است. (به طور مثال تگ‌های: < html >< head> و < body >) همچنین در یک اپلیکیشن، اغلب بخش‌هایی(مانند Header و Footer) وجود دارد که در هر صفحه از برنامه شما تکرار می‌شوند. بنابراین این تکرار می‌تواند موضوع نگهداری کد برای آینده را دشوار کند.

خوشبختانه Razor view engine از مفهوم Section پشتیبانی می‌کند، این مفهوم به شما اجازه می‌دهد تا المان‌های رایج را تنها در یک مکان(به نام Layout) ارائه نمایید.

Layout چیست؟ Layout یک صفحه‌ی HTML است که کدهای مشترک تمامی صفحات در آنجا قرار می‌گیرد. این صفحه در ارتباط با Razor Viewهای معمولی Render می‌شود و به تنهایی نمی‌توان آن را استفاده نمود.

مزایای Layout

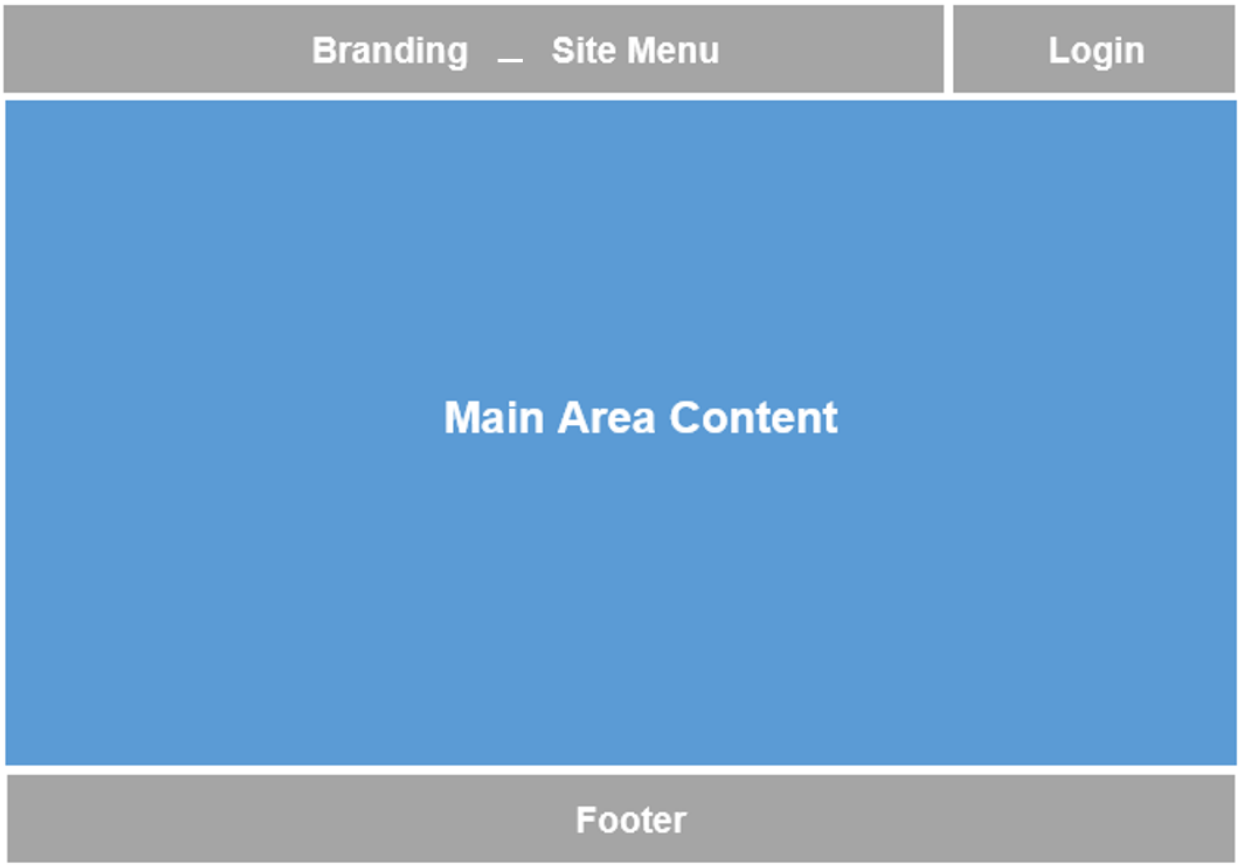
Layout با استخراج کدهای عمومی صفحات، تکثیر کد در اپلیکیشن را کاهش می‌دهد. این عمل باعث می‌شود:

- **تغییرات ساده‌تر شود.**
- **Viewهای شما به راحتی مدیریت و نگهداری شود.**
- **و به طور کلی یک حرکت فوق العاده است!**

نکته!!

استفاده دائم از layoutها می‌تواند در کاهش یکپارچگی در یک صفحه بسیار مفید باشد.

بیایید خیلی سریع نگاهی به Layoutها بیندازیم:



چطور از Layout استفاده کنیم؟

یک فایل Layout شبیه یک صفحه‌ی معمولی Razor است،که حتما باید تابع RenderBody() را صدا بزند. این تابع به Template engine می‌گوید که در چه مکان‌هایی Child Viewها قرار بگیرند.

به صورت قراردادی Layoutها در مسیر Views / Shared قرار می‌گیرند و معمولا فایل پایه Layout اپلیکیشن با نام _Layout.cshtml نامگذاری شود.

حالا شما درون فولدر Views یک فولدر به نام Shared ایجاد نمایید و سپس درون این فولدر فایلی با نام _Layout.cshtml(همراه با محتوای پایین) اضافه نمایید:

_Layout.cshtml:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.c
    ss">
    <script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></
    script>
    <script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"
    ></script>
    <title>@ViewData["Title"]</title>
</head>
<body>
    @RenderBody()
</body>
</html>
```

متد RenderBody در Layout الزامیست.

همانطور که می‌بینید، فایل Layout شامل المنت‌های موردنیاز هر صفحه(مانند <html> و <head>)می‌باشد. حالا Viewها با تنظیم پراپرتی Layout در داخل‌یک بلوک کد Razor، می‌توانند فایل Layout را مشـــخص کنند.

Index.cshtml:

```
@using Microdev.ASPNETCore.ViewModels;

@model List<EmployeeViewModel>

@{
    Layout = "_Layout";
}
<div class="Container">
    <h3> @ViewData["Title"]</h3>

    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
```

تنظیم Layout برای View

```
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>
</div>
```

نکته!!

همانطور که در کدهای بالا می‌بینید، دیگر نیاز به وجود اسکریپت‌های Bootstrapt و تگ‌های تکراری (مثل ..,<Head> ,<HTML>)در فایل Index.cshtml نیست، زیرا این تگ‌ها در فایل Layout.chtml_ وجود دارد.

بعد از اجرای اپلیکیشن، تمام محتوای View در داخل Layout و در جایی که متد RenderBody فراخوانی می‌شود رندر خواهد شد.

نکته!!

در Razor، فایل View قبل از فایل Layout رندر می‌شود، این بدین معنی است که شما می‌توانید مقادیری مانند عنوان صفحه را از طریق پراپرتی‌های View Data تنظیم و سپس از این مقادیر در Layout استفاده نمایید.

Section چیست؟

Layout مکانیست که در آن بتولنید محتوای View را در هنگام فراخوانی RenderBody@ رندر کنید. اما هنگامی که شما در اپلیکیشن خود شروع به استفاده از layoutها می‌کنید، یک نیازمندی مشترک این است که بتوانید بخش‌هایی از مطالب خود را (**مثل:** Header, Footer و...) در Child Viewها هندل کنید.

برای حل این مشکل، Razor view engine از مفهوم Sectionها پشتیبانی می‌کند. Section به شما امکان می‌دهد تا محتوای خود را در جاهای مختلف یک Layout قرار دهید.

Section یک روش سازماندهی برای مکان‌هایست که عناصر View باید در یک Layout قرار گیرند.

نحوه استفاده از Section:

- Sectionها با استفاده از section@ تعریف می‌شوند. این کلمه کلیدی را می‌توان در هر نقطه‌ای از View قرار داد(بالا، پایین و یا هر جایی که مناسب است).
- و در Layout با صدا زدن ()RenderSection@ این Sectionها رندر می‌شوند.

Sectionها می‌توانند required یا optional باشند:

- اگر required باشند، پس یک View باید @section را تعریف کند.
- و اگر optional باشـــند، می‌توان section را از Viewحذف نمود، تا Layout آن را پر کند، در این صورت Section های Skip شده در رندر HTML ظاهر نمی‌شوند.

برای نشان دادن این که چگونه Sectionها کار می‌کنند:

۱) فایل _Layout.cshtml را باز و کدهای پایین را درون آن قرار دهید.

_Layout.cshtml:

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.c
    ss">
    <script
    src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></
    script>
    <script
    src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"
    ></script>
    <title>@ViewBag.Title</title>

</head>
<body class="panel-body">
<nav class="navbar navbar-inverse">
    <div class="container-fluid">
        <div class="navbar-header">
            <a class="navbar-brand" href="#">Microdev</a>
        </div>
        <ul class="nav navbar-nav">
            <li ><a href="#">Home</a></li>
            <li><a href="#">Employee</a></li>
            <li><a href="#">Department</a></li>
        </ul>
    </div>
</nav>

    @RenderBody()

    @RenderSection("Footer", required: false)
```

با فراخوانی این متد تمام محتوای View به غیر از محتوای Sectionها رندر می‌شوند.

این متد رندر Sectionی به نام Footer را انجام می‌دهد. از آنجایی که این Section به صورت optional تعریف شده، اگر در View همچین Sectionی وجود نداشته باشد، این متد اجرا نخواهد شد.

```
</body>
</html>
```

هنگامی که Razor، شـــروع به تجزیه‌ی Layout می‌کند، محتویات Section در View توســـط متد RenderSection رندر می‌شود.

نکته!!

ـ یک View تنها می‌تواند Sectionهایی که در Layout ذکر شـــده‌اند را تعریف کند. اگر شـــما تلاش کنید Sectionهایی را در View تعریف کنید، که در RenderSection مربوط به Layout وجود ندارد، آن وقت یک Exception اجرا می‌شود.

۲) حالا به فایل Index.cshtml باید Sectionی به نام Footer اضافه کنید:

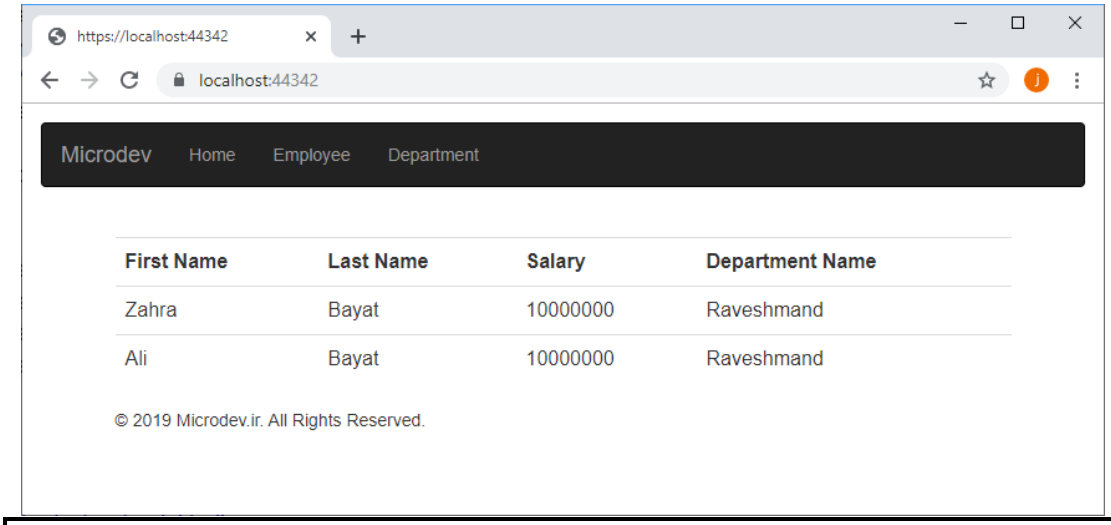
Index.cshtml

```
@using Microdev.ASPNETCore.ViewModels;
@model List<EmployeeViewModel>

@{
    Layout = "_Layout";
}

<div class="Container">
  <h3>@ViewData["Title"] </h3>
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>
    @section Footer {
    <div class="bg-dark text-white">
        © 2019 Microdev.ir. All Rights Reserved.
    </div>
```

تمام محتوای درون آکولاد قسمتی از Section – Footer هستند و به عنوان محتوای Body در نظر گرفته نمی‌شود.

```
    }
</div>
```

حالا اپلیکیشن را اجرا کنید:



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter5/Sample3

Partial view چیست؟

گاهی در اپلیکیشـــن، نیاز به Copy/Paste کردن قطعات تکراری Razor و یا HTML دارید. همان‌طور که می‌دانید، این کار باعث ازدیاد کدهای تکراری می‌شود و در آینده نگهداری کد را سخت خواهد کرد.

پس چاره چیســت؟ راه حل این مشـکل، اسـتفاده از Partial Viewهاسـت. Partial Viewها بخش ی از یک View هستند و می‌توان گفت یک روش عالی برای به اشتراک گذاشتن کد بین Viewهای مختلف می‌باشند.

با Partial Viewها می‌توان یک View بزرگتر را به قطعات کوچکتر تبدیل کرد، با این کار پیچیدگی درـیک View بزرگتر کاهش می‌یابد و می‌توان بخشی از یک View را درون View دیگر مورد استفاد قرار داد.

شما می‌توانید Partialها را به عنوان یک Child در یک View در نظر بگیرید که توسط Viewها رندر می‌شود.

ایجاد یک Partial View

Partial Viewـیک View معمولی با فرمت فایل cshtml اســـت، که دقیقا برای قرارگیری در View، از همان استراتژی Viewها استفاده می‌کند.

ساده‌ترین روش ایجاد یک Partial View، افزودن یک View معمولی با استفاده از الگوی MVC View Page item است. برای همین منظور، من یک فایل با نام _FooterPartial.cshtml به فولدر Views / Home اضافه نمودم.

_FooterPartial.cshtml:

```
<div> © 2019 Microdev.ir. All Rights Reserved.</div>
```

نکته!!

به طور معمول Partial Viewها هم مانند Layoutها، با یک Underline نامگذاری می‌شوند.

استفاده از Partial View

برای استفاده از یک Partial View درون یک View، باید از متدی به نام Html.PartialAsync استفاده نمایید، البته متدهای دیگر، مانند Html.Partial و Html.RenderPartial هم وجود دارد که البته توصیه نمی‌شود.

حالا بیایید فایل Index.cshtml را ویرایش کنیم:

```
@using Microdev.ASPNETCore.ViewModels;

@model List<EmployeeViewModel>
@{
   Layout = "_Layout";
 }


<div class="Container">
  <h3> @ViewData["Title"]</h3>

    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
```

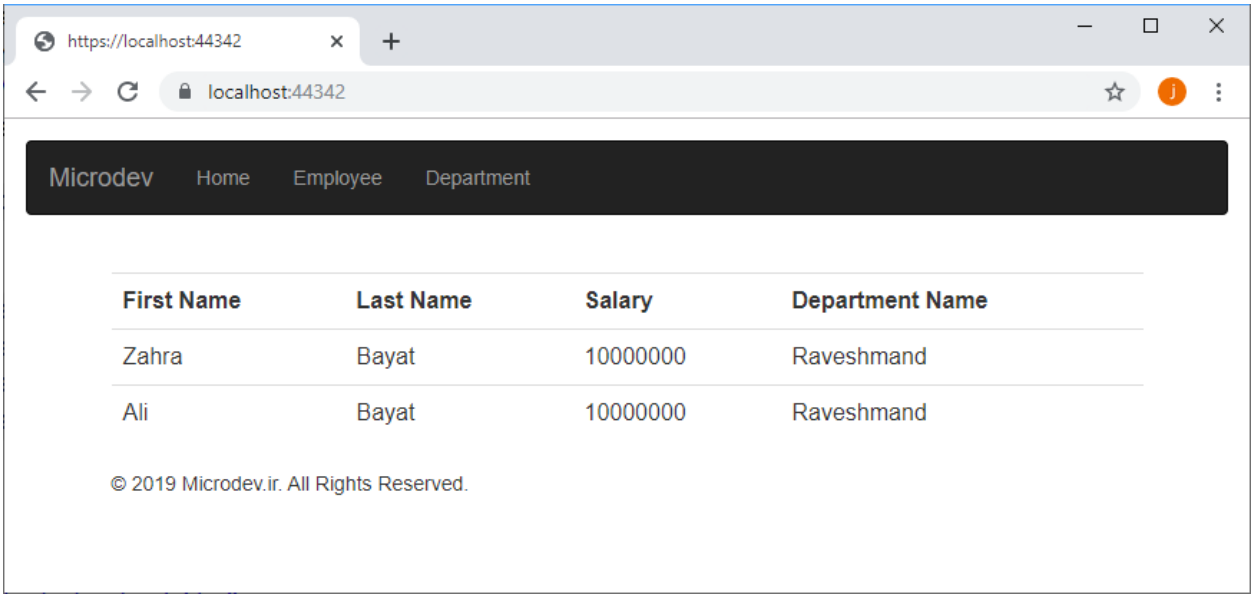
```
                <td>@item.DepartmentName</td>
            </tr>
        }
    </table>
@await Html.PartialAsync("_FooterPartial")
 </div>
```

یک Partial View می‌تواند با استفاده از @await Html.PartialAsync() رندر شود.

نکته!!

با روش بالا دیگر نیاز به نوشتن Section→ Footer در هر View ندارید.

لطفا اپلیکیشن را اجرا کنید:



استفاده از Strongly Typeهای Partial View

گاهی نیاز است Partial Viewها در زمان Render شدن از آبجکت‌های View Model استفاده کنند. در این‌گونه موارد باید از Strongly Typeهای Partial View استفاده نماییم.

در این Partial Viewها، کدهای Razor شبیه کدهای یک View استاندارد است، با این تفاوت که Partial Viewها معمولا به جای اینکه به عنوان Result یک اکشن‌متد باشند، مستقیما درون یک View دیگر صدا زده می‌شوند.

برای نشان دادن این ویژگی، درون فولدر Views / Home یک View با نام _EmployeePartial.cshtml ایجاد و سپس کدهای پایین را در آن اضافه نمایید.

```
@using Microdev.ASPNETCore.ViewModels;
```

```
@model EmployeeViewModel

<table class="table">
    <tr>
        <td>@Model.FirstName</td>
        <td>@Model.LastName</td>
        <td>@Model.Salary</td>
        <td>@Model.DepartmentName</td>
    </tr>

</table>
```

در این Partial View قرار است اطلاعات کارمند در یک جدول نمایش داده شود.

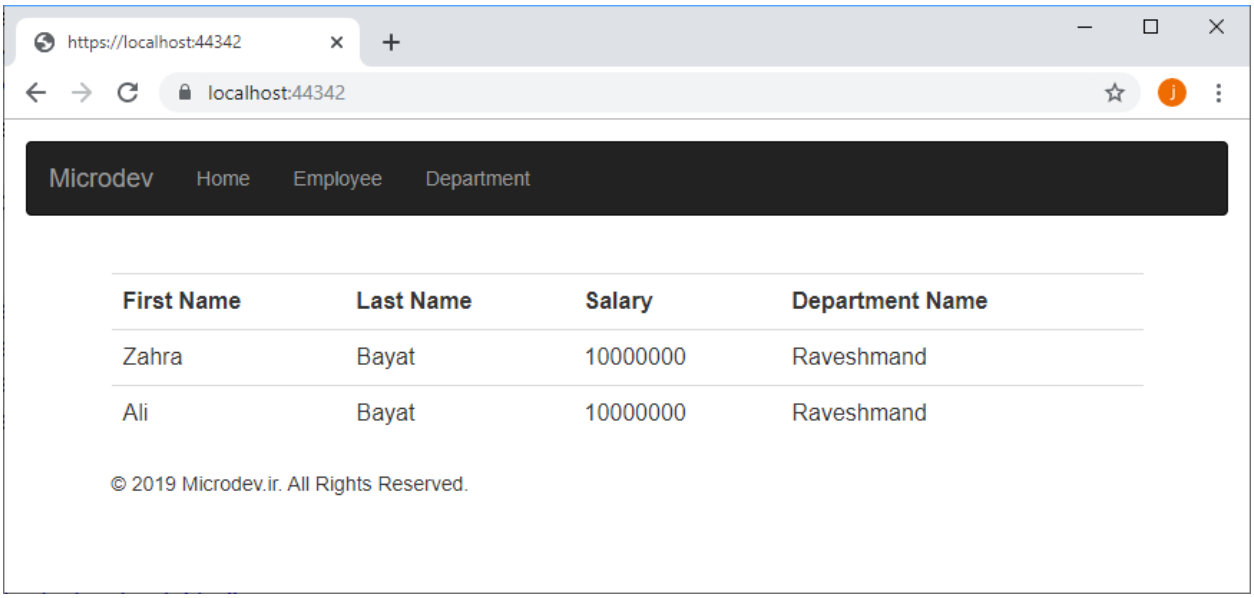
حالا برای استفاده از این Partial View، من فایل Index.cshtml را همانند کد پایین آپدیت نمودم.

ویرایش فایل Index.cshtml :

```
@using Microdev.ASPNETCore.ViewModels;
@model List<EmployeeViewModel>
@{
    Layout = "_Layout";
}

<div class="Container">
    <h3> @ViewData["Title"]</h3>
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>

        <tr>
            @foreach (var item in Model)
            {
                @await Html.PartialAsync("_EmployeePartial", item)
            }
        </tr>

    </table>

    @await Html.PartialAsync("_FooterPartial")

</div>
```

تفاوت این مثال با قبلی این است که من یک آرگومان اضافی به متد PartialAsync فرستادم تا View Model را برای EmployeePartial آماده کند.

برای دیدن نتیجه‌ی Strongly Typed Partial View، اپلیکیشن را اجرا کنید.



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter5/Sample4

ViewStart چیست؟

با توجه به ماهیت Viewها، شما مجبورید که مرتبا برخی کدهای خاص را تکرار کنید. **به طور مثال:** اگر نیاز به تغییر نام فایل Layout داشته باشید، باید هر Viewیی که به آن اشاره می‌کند را بیابید و تغییرات موردنظر را اعمال نمایید.

همانطور که می‌دانید این فرایند باعث بروز خطا می‌شود و موضوع نگهداری کد را به خطر می‌اندازد. اما به نظر شما **راه حل این موضوع چیست؟**

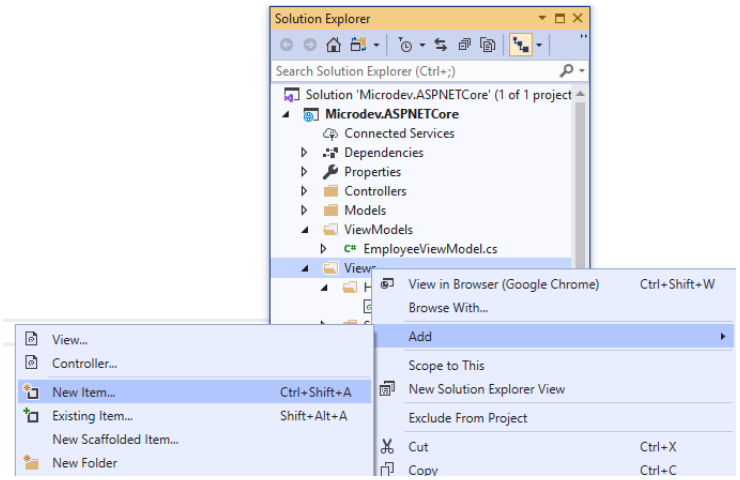
اگر تمام Viewها از یک Layout استفاده کنند، می‌توان این مشکل را با افزودن فایل _ViewStart.cshtml حل نمود.

ViewStart چیست؟ ViewStart فایلیست که در آن، کدهای مشترک Viewها قرار می‌گیرد (مانند نام تکراری Layout که در تمام Viewها قرار دارد) و بعد از اجرای اپلیکیشن، MVC به این فایل نگاه و این کدها را در ابتدای هر View اجرا می‌کند.

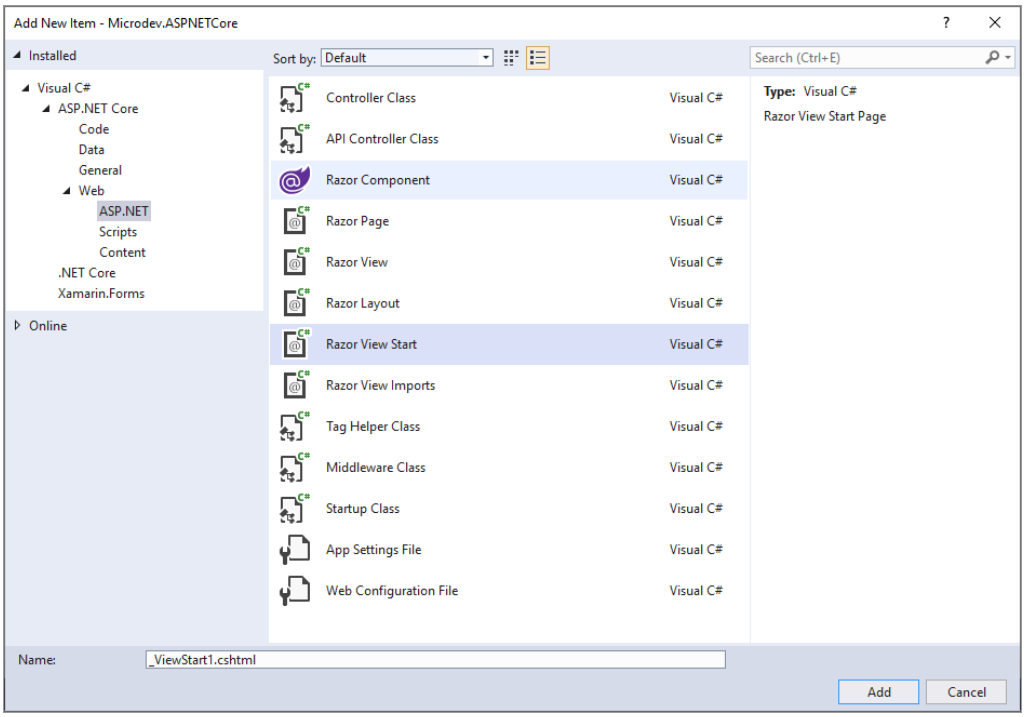
ایجاد فایل ViewStart

برای ایجاد یک فایل ViewStart:

- **بر روی فولدر Views راســت کلیک کنید و از منوی باز شــدهNew Item➤ Add را انتخاب کنید.**



- **حالا همانند تصویر زیر، Razor View Start را انتخاب و بر روی Add کلیک کنید.**



نکته!!

ویژوال استودیو به صورت خودکار نام فایل را به _ViewStart.cshtml تنظیم می‌کند و هر کدی در این فایل قرار بگیرد، قبل از View اجرا می‌شود. بنابراین نام این فایل را تغییر ندهید.

این فایل می‌تواند حاوی هر کد Razorی باشد، اما معمولا جهت تنظیم Layout، برای تمام Viewهای اپلیکیشن، مورد استفاده قرار می‌گیرد.

محتوای فایل ViewStart همانند کد پایین است بنابراین دیگر نیاز نیست که عبارت Layout را در هیچ Viewیی قرار دهید.

```
@{
    Layout = "_Layout";
}
```

نکته!!

در فایل Index.cshtml دیگر نیازی به عبارت Layout نیست بنابراین این عبارت را در این فایل پاک نمایید.

این Assignment ساده اجازه می‌دهد، تا Razor بداند که ما در Template خود، از فایلی به نام _Layout.cshtml استفاده می‌کنیم.

نکته!!

_ViewStart.cshtml تنها برای Viewها اجرا می‌شود و برای Layoutها و یا Partial Viewها اجرا نخواهد شد.

ViewImports چیست؟

_ViewImports.cshtml یک ویژگی جدید از ASP.NET Core MVC است، که به شما امکان می‌دهد تا دستوراتی که برای همه‌ی Viewها موردنیاز است را، مشخص نمایید.

به طور مثال: مطمئنا در پروژه شما Namespaceهایی وجود دارد که در خیلی از Viewها تکرار می‌شود. برای جلوگیری از اضافه کردن این عبارات به هر View، می‌توانید از اهرم _ViewImports استفاده کنید.

ایجاد فایل ViewImports

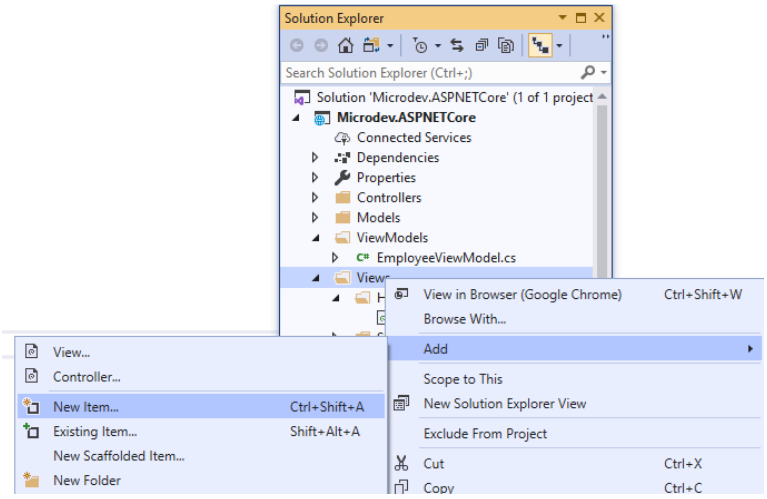
فایل _ViewImports.cshtml را می‌توان در هر فولدری قرار داد. در این صـــورت این فایل تنها برای تمام Viewهای آن Folder اعمال می‌شود. اما معمولا، این فایل را در ریشـه فولدر Views قرار می‌دهند تا به تمامی Viewهای اپلیکیشن اعمال شود.

نکته!!

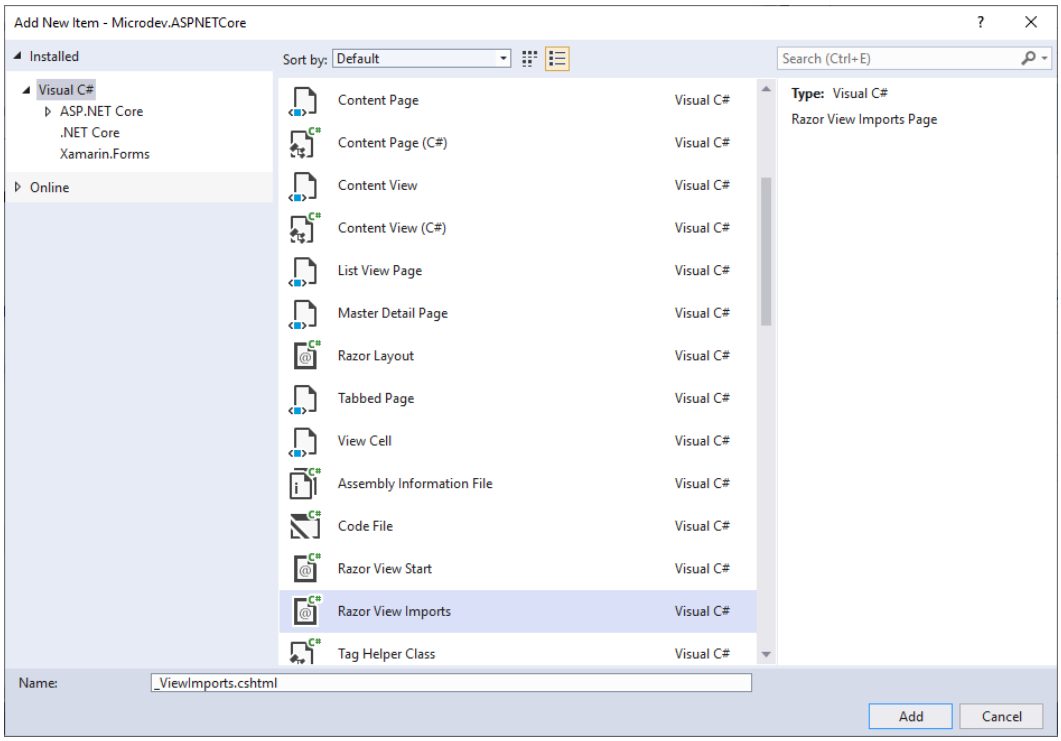
توجه داشته باشید که در این فایل تنها باید دستورات Razor نوشته شود.

برای ایجاد یک فایل ViewImports:

- **بر روی فولدر Views راسـت کلیک کنید و از منوی باز شـده New Item ➤ Add را انتخاب نمایید.**



- **حالا همانند تصویر زیر، Razor View Imports را انتخاب و بر روی Add کلیک کنید.**



- سپس Namespace زیر را در فایل ViewImports اضافه و در پایان این عبارت را از فایل Index.cshtml حذف کنید.

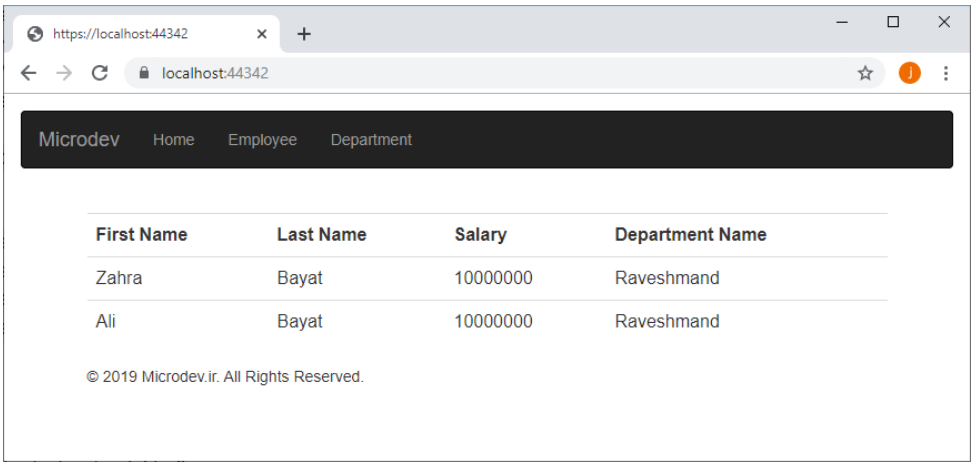
```
@{
   @using Microdev.ASPNETCore.ViewModels;
}
```

حذف Namespace از Index.cshtml:

```
@model List<EmployeeViewModel>

<div class="Container">
    <h3> @ViewData["Title"]</h3>
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>

        <tr>
            @foreach (var item in Model)
            {
                @await Html.PartialAsync("_EmployeePartial", item)
```

```
            }
        </tr>

    </table>

    @await Html.PartialAsync("_FooterPartial")

</div>
```

برای دیدن نتیجه کار لطفا اپلیکیشن را اجرا کنید:



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter5/Sample5

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Tag Helper چیست؟**
- ✓ **چطور از Tag Helper استفاده کنیم؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What is Razor View Engine?

Q2: What is the difference between ViewBag and ViewData in MVC?

Q3: What are HTML Helpers in MVC?

Q4: What is Layout in MVC?

Q5: Explain Sections in MVC?

Q6: Can you explain RenderBody in MVC?

Q7: What is ViewStart Page in MVC?

Q8: What are Code Blocks in Views?

Q9: Why to use Html.Partial in MVC?

Q10: What is PartialView in MVC?

Quiz

Q1: Which of the following is a type of view in MVC?

1. Partial view
2. Executable view
3. Data view
4. Designer view

Q2: What are partialviews in MVC?

1. It’s the resource file for a view
2. View that has strongly-type models
3. Reusable view
4. All of the above

Q3: HtmlHelper class _____.

1. Generates html elements
2. Generates html view
3. Generates html help file
4. Generates model data

Q4: Which of the following view contains common parts of UI?

1. Partial view
2. Html View
3. Layout view
4. Razor view

Q5: Which of the following methods are used to render partial view?

1. Html.Partial()
2. Html.RenderPartial()
3. Html.RenderAction()
4. All of the above

Q6: How to transfer data from controller to view?

1. Using model object
2. Using ViewBag
3. Using ViewData
4. All of the above

Q7: _____is a dictionary of strings pointing to objects.

1. View model
2. ViewData
3. ViewBag
4. HttpContext

Q8: It adds a section where form validation errors will be displayed.

1. Html.ValidationSummary
2. Html.BeginForm
3. Html.Validation
4. All of the above

Q9: A _____ in Razor is a template that includes common code.

1. Layout
2. View
3. PartialView
4. None

Q10: Every layout must call the _____ function.

1. @RenderPartialView ()
2. @RenderBody()
3. @RenderSection()
4. @RenderView()

Answer

1-Correct Answer: Partial view

2-Correct Answer: All of the above

3-Correct Answer: Generates html elements

4-Correct Answer: Layout view

5-Correct Answer: All of the above

6-Correct Answer: All of the above

7-Correct Answer: ViewData

8-Correct Answer: Html.ValidationSummary

9-Correct Answer: Layout

10-Correct Answer: @RenderBody()

خلاصه فصل

- ✓ Razor زبانیست که به شما امکان می‌دهد تا با استفاده از ترکیبی از HTML و سی‌شارپ، HTML داینامیک تولید کنید.
- ✓ کنترلرها می‌توانندبا استفاده ازیک View Model، داده‌ها رابه Viewپاس داده و برای دسترسی به Propertyهای این View Model، Viewبایدبا استفاده از @model نوع Model را اعلام کند.
- ✓ کنترلرها می‌توانند با استفاده از دیکشنری ViewData، یک لیست key-value را به View پاس دهند.
- ✓ عبارات Razor مقادیر سی‌شارپ را با استفاده از @ یا ()@ به خروجی HTML رندر می‌کند.
- ✓ عبارات سی‌شارپ با استفاده از {}@ در Razor تعریف می‌شود.
- ✓ سی‌شارپ در بلوک‌های کد Razor، باید عبارات کاملی باشد، بنابراین حتما سمی‌کالن را فراموش نکنید.
- ✓ از حلقه‌ها و شرط‌ها جهت ایجاد HTML داینامیک استفاده می‌شود.
- ✓ می‌توانید HTML مشترک در چندین View مختلف را، در یک Layout قرار دهید. Layout محتوای View را با استفاده از متد RenderBody فراخوانی می‌کند.
- ✓ _ViewStart.cshtml برای اجرای کد مشترک Viewها استفاده می‌شود و همیشه قبل از اجرای هر View فراخوانی خواهد شد.
- ✓ _ViewImports.cshtml یک ویژگی جدید از ASP.NET Core MVC است، که‌به شما امکان می‌دهد تا دستوراتی که برای همه‌ی Viewها موردنیاز است را، مشخص کنید.

فصل ششم: Tag Helperها چیست؟

آنچه خواهید آموخت:

- Tag Helperچیست؟
- فعال کردن Tag Helper در اپلیکیشن
- استفاده ازTag Helperها
- ایجاد یک Tag Helper سفارشی

Tag Helperچیست؟

همانطور که دیدید یکی از جنبه‌های بسیار مهم وب اپلیکیشن‌ها، نمایش داده‌های داینامیک بود، اما در این بین کاربر اپلیکیشن شما هم، باید بتواند داده‌هایی را به اپلیکیشن ارسال کند. برای حل این مشکل، Formها یک راه‌حل کلیدیست و ASP.NET Core برای رسیدن به این راه حل یک ویژگی به نام Tag Helper را به ارمغان آورده است.

Tag Helperها اجزای Razor هستند، که می‌توانید آنها را برای سفارشی کردن HTMLهای اپلیکیشن، استفاده کنید. **برای مثال:** Tag Helperها را می‌توان به یک عنصر HTML مانند تگ < input > اضافه نمایید.

ویژگی‌های Tag Helper شبیه به HTML Helper است، با این تفاوت که Tag Helperها نسبت به Helper HTMLها یک Syntax ساده‌تری دارند و فوکوس آن‌ها بر روی سی‌شارپ است. می‌توان به قطعیت گفت: Tag Helperها جانشین واقعیه HTML Helperها هستند.

در پایین دو مثال برای ایجاد یک لینک به اکشن متد Index در کنترلر Home نوشته شده که یکی با HTML Helper و دیگری با یک Helper Tag است.

```
1) @Html.ActionLink("Go to home page", "Index", "Home", null, new { @class = "h4" })
2) <a asp-controller="Home" asp-action="Index" class="h4">Go to home page</a>
```

همانطور که می‌بینید، هر دو مثال معادل هم هستند، اما به طور قطع، مثال Tag Helper ساده‌تر و قابل‌فهم‌تر است.

در مواردی که نیازی به اضافه کردن Style به یک لینک ندارید، HTML Helper می‌تواند مختصرتر باشد اما متاسفانه در مواردی که نیاز به سفارشی کردن خروجی HTML دارید، سینتکس HTML Helper کمتر قابل فهم است و حتی شاید کمی گیج کننده هم باشد.

برای کار با فرم‌ها، Tag Helperها بسیار کاربردی هستند. شما می‌توانید برای تولید HTML داینامیک(براساس Propertyهای Model)، ایجاد Id و نام Attributeها، تنظیم مقدار المنت‌ها به مقدار Propertyهای Model از Tag Helperها استفاده کنید. این ویژگی به میزان قابل توجهی مقدار کد HTML را کاهش می‌دهد.

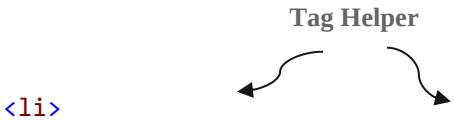
نکته!!

Tag Helperها برای اســتفاده از عبارات لامبدا، نیاز به دســترسـ ی به داده‌های مدل شــما ندارند، بنابراین شما هم نیازی به تعریف نخواهید داشت.

نتیجه گیری: به طور خلاصه Tag Helperها ساده‌ترین مکانیزم ساخت فرم هستند و به جای نوشتن HTMLهای تودرتو، شما را مجبور به تمرکز بر طرح کلی اپلیکیشن می‌کنند.

فعال کردن Tag Helper در اپلیکیشن

Tag Helperها می‌توانند المنت HTMLیی که به آن متصل شده را، با اضافه کردن عباراتی مانند asp- تغییر دهند.




```
<li>
    <a asp-controller="Home" asp-action="Index">Home</a>
</li>
```

نکته!!

قبل از اینکه View رندر شــود، ASP.NET Core این Tag Helperها را با HTML attributeهای واقعی جایگزین می‌کند.

Tag Helperها باید قبل از نوشــتن هر کدی رجیســتر شــوند. بنابراین، شــما باید یک عبارت را در فایل _ViewImports.cshtml اضافه نمایید تا به MVC بگوید کلاس‌های Tag Helper را جستجو کند.

لطفا در فولدر Views، فایل _ViewImports.cshtml را با محتوای زیر آپدیت نمایید:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

عبارت addTagHelper، برای فعال کردن Tag Helperهای داخلی ASP.NET Core اســت اما در ادامه ایجاد یک Tag Helper سفارشی هم توضیح داده خواهد شد.

نکته!!

این قابلیت با پکیج Microsoft.AspNetCore.Mvc همراه اســت و در تمام Templateهای پروژه گنجانده شده است.

استفاده از Tag Helperها

Tag Helperها Attributeهایی هستند که به المنت‌های HTML، اضافه و باعث تغییر در عملکرد این المنت‌ها می‌شوند. Tag Helperها کدنویسی شما را ساده‌تر می‌کنند و می‌توان برای موارد زیر از آن‌ها استفاده نمود:

- **پر کردن خودکار مقادیر از View Model.**
- **انتخاب نوع ورودی صحیح برای نمایش اطلاعات.**
- **نمایش هر گونه خطای اعتبارسنجی.**

ASP.NET Core MVC همراه با بسـیاری از Tag Helperها آمده اسـت؛ اکثر آنها همان Helper Razor HTML هسـتند که برای ویرایش فرم‌ها اسـتفاده می‌شـدند، اما مواردی از Tag Helperها هم هسـتند که برای رندر کردن المنت‌های مختلف HTML بر اساس environment و script fallback یا فایل‌های CSS بوجود آمدند.

من می‌خواهم در اینجا تعدادی از Tag Helperهای رایج و نحوه استفاده از آنها را معرفی کنم.

Environment Tag Helper

مطمئنا همه شـما دوسـت دارید بر اسـاس محیط فعلی، یک رفتار را بطور خودکار فعال یا غیرفعال کنید. **برای مثال:** شما ممکن است بخواهید اپلیکیشن در محیط Staging و Production از فایل CSS minified و در محیط Development از نسخه‌ی non-minified استفاده کند.

Environment tag helper یک راه آسان برای انجام این کار است، که به شما اجازه می دهد بلوک‌های کد را در محیط‌های مختلف مشخص کنید.

```
<environment names="Development">
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet" href="~/css/site.min.css" />
</environment>
```

این لینک فقط در محیط‌های Staging یا Production اجرا می‌شود.

این لینک فقط در محیط Development اجرا می‌شود.

نکته!!

Environment Tag Helper بیشتر به همراه Link و Script Tag Helperها استفاده می‌شود.

Link Tag Helper و Script Tag Helper

Script و Link، دو Tag Helperبسیار کاربردی در مجموعه‌ی Tag Helperها هستند که رفتاری شبیه به هم دارند. هر کدام از این دو Tag Helper، دو Propery دارند که این امکان را فراهم می‌کند تا اسکریپت‌های شما در دو مکان مشخص شود. (**به طور مثال**: یکی Local و دیگری CDN)

مزیت این کار این است که، اگر یکی از مسیرها به مشکل برخورد کند، دیگری جایگزین خواهد شد. به طور مثال اگر CDN از کار بیفتد، اسکریپت از مکان Local خوانده می‌شود.

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet"
     href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.
     css"
    asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
    asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
    asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true"
/>
</environment>
```

توجه داشته‌باشید، در مثال‌بالا environment از متغیر ASPNET_ENVIRONMENT خولنده می‌شود و اگر این متغیر تنظیم نشده باشد، ASP.NET Core فرض می‌کند که در محیط Production هستید.

همانطور که در کد بالا می‌بینید، LinkTag Helper شامل Propertyهای، asp-fallback-test-class، asp-fallback-test-property و asp-fallback-test-value است. این Propertyها به شما امکان می‌دهند تا بررسی کنید که یک کلاس CSS وجود دارد یا خیر؟ اگر وجود داشت، باید مقدار موردانتظار برای یک Propertyی مشخص شده را داشته باشد. در ادامه با هم این Propertyها را بررسی می‌کنیم:

- **href**: آدرس مشخص شده جهت لینک به CSS مرتبط.
- **asp-fallback-href**: در صورتی که URL مشخص شده در href از کار بیفتد، این آدرس جایگزین خواهد شد.
- **asp-fallback-test-class**: بررسی می‌کند که آیا کلاس CSS مشخص شده، وجود دارد یا خیر؟

- **asp-fallback-test-property و asp-fallback-test-value**: تایید می‌کنند که نام و مقدار CSS Property با مقدار مورد انتظار تنظیم شده است یا خیر.

نکته!!

Script Tag Helper هم از Propertyهای asp-fallback-src و asp-fallback-test استفاده می‌نماید و کارکردش شبیه Link است.

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-
3.3.1.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-
tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
</script>
```

Form Tag Helper

اولین چیزی که برای ساختن فرم HTML خود نیاز دارید، عنصر <form> است.

```
<form asp-action="Index" asp-controller="Home">
```

این Tag helper باعث می‌شود تا Attributeهای اکشن و کنترلر به HTML نهایی اضافه شود. تنظیم asp-action و asp-controller این امکان را به شما می‌دهند تا اکشن‌متد و کنترلر موردنظر خود را در زمان ارسال فرم مشخص کنید.

Label Tag Helper

در اپلیکیشن، هر فیلد <input> نیاز به یک label مرتبط دارد. شما می‌توانید نام و سایر Attributeهای این Labelها را به صورت دستی تنظیم نمایید، اما روشی ساده‌تر برای انجام این کار، استفاده از Label Tag Helper است.

Label Tag Helper، عنوان و سایر Attributeهای تگ <label> را بر اساس Propertyهای View Model تنظیم می‌نماید. **برای مثال:** asp-for در کد پایین Label را به Property ← FirstName بایند می‌کند و اگر در View Model، یک نام فارسی برای این Property مشخص شده باشد، به صورت خودکار این تغییرات در زمان نمایش به این Label اعمال خواهد شد.

```
<label asp-for="FirstName"></label>
```

ایجاد یک Tag Helper سفارشی

Tag Helperیک کلاس معمولیست(با هر نامی که دوست دارید) که باید از کلاس پایه TagHelper ارثبری و در متد Process یا ProcessAsync، رفتارهایش را تعریف کند.

بنابراین برای ایجاد یک Tag Helper سفارشی، نیاز است:

۱) **یک کلاس ایجاد کنید که از کلاس پایه TagHelper ارثبری کند.**

۲) **Attribute یا Tagی که می‌خواهید با آن ارتباط برقرار کنید را مشخص نمایید.**

۳) **برای اضافه کردن محتوای سفارشی، متد Process یا ProcessAsync را override کنید.**

- `public virtual void Process(TagHelperContext context, TagHelperOutput output)`
- `public virtual Task ProcessAsync(TagHelperContext context, TagHelperOutput output)`

این متدها دارای دو آرگومان ورودی هستند:

۱. **context**: شامل اطلاعاتی در مورد، نحوه اجرای فعلی context است.

۲. **output**: حاوی یک مدل از تگ HTML و محتوای آن است، که باید توسط Tag Helper تغییر کند.

در متد Process، Tag Helper قادر است تا TagHelperContext فعلی را بازبینی کند و برخی از HTMLها را تولید یا به نوعی TagHelperOutput را تغییر دهد.

نکته!!

یک قرارداد برای ایجاد Tag Helper، این است که فایل Tag Helper سفارشی، درون فولدر TagHelpers اضافه شود و به صورت پیش‌فرض در نام این فایل، تگ HTML موردنظر هم قرار گیرد.

مثال:

در پروژه Microdev.ASPNETCore، فولدری به نام TagHelpers ایجاد و سپس درون این فولدر کلاسی به نام CustomBottonTagHelper.cs اضافه نمایید.

عنصر هدف این کلاس، المنت Button است و قرار است یک Attribute به نام microdev-custom-button را به این المنت اضافه کند.

```
using Microsoft.AspNetCore.Razor.TagHelpers;
```

```
namespace Microdev.ASPNETCore.TagHelpers
{
    [HtmlTargetElement("button", Attributes = "microdev-custom-button")]
    public class CustomBottonTagHelper : TagHelper
    {

        [HtmlAttributeName("microdev-custom-button")]
        public string BottonName { get; set; }
        public override void Process(TagHelperContext context, TagHelperOutput
        output)
        {
            output.Attributes.SetAttribute("type", "submit");
            output.Attributes.SetAttribute("value", $"Microdev{BottonName}");
            output.Attributes.SetAttribute("name", $"Microdev{BottonName}");
        }
    }

}
```

تنظیم Attributes تضمین می‌کند که Tag-Helper توسط Attribute ← microdev-custom-button انجام می‌شود.

قرار دادن این Attribute در بالای Property ها، این امکان را می دهد تا مقدار Razor markup را تنظیم کنیم.

Razor engine با فراخوانی این متد Tag Helper را اجرا می کند.

همانطور که در مثال بالا می‌بینید:

- **ما کلاسی با نام CustomBottonTagHelper ایجاد کردیم که از کلاس پایه‌ی TagHelper ارث‌بری می‌کند.**
- **تنظیم پراپرتی Attributes در HtmlTargetElement (که در بالای این کلاس نوشته شده) تضمین می‌کند که Tag-Helper توسط Attribute ← microdev-custom-button انجام می‌شود.**
- **قرار دادن Attribute ← HtmlAttributeName در بالای Property ← BottonName این امکان را می‌دهد تا مقدار Razor markup تنظیم شود.**
- **حالابلید متد Process را override نمایید،تا Razor engine بتولند Tag Helper را اجرا کند.**
- **درپلیان، دربدنه این متد، فرایند تغییرنام Button بعد از کلیک بر روی آن نوشته شده است.**

قبل از اینکه بتوانید از امکان TagHelper سفارشی در Viewهای خود استفاده کنید، ابتدا باید با استفاده از دستور addTagHelper، به پروژه خود یک رفرنس دهید. برای اعمال این دستور، کلاس _ViewImports.cshtml را باز و سپس عبارت `@addTagHelper *, Microdev.ASPNETCore` را به آن اضافه نمایید.

```
@using Microdev.ASPNETCore.ViewModels;
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, Microdev.ASPNETCore
```

در حال حاضر این Tag Helper، بی فایده است. برای تکمیل این کار، شما باید محتوای المنت را بخوانید و یک Attribute جدید ایجاد کنید. بنابراین بیایید با هم، درکنترلر EmployeeController متد CreateEmployee را اضافه نماییم:

EmployeeController:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController: Controller
    {
        public List<Employee> Employees { get; private set; }

        public EmployeeController()
        {
            Employees = new List<Employee>
            {
                new Employee{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    Salary=10000000
                },
                new Employee{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    Salary=10000000
                },
            };

        }
        public IActionResult CreateEmployee()
        {
            return View(new EmployeeViewModel());
        }
        public IActionResult GetEmployee(int employeeId)
        {
```

افزودن متد CreateEmployee

```
            var employee=  Employees.FirstOrDefault(x => x.EmployeeId ==
           employeeId);
               return Json(employee);
           }

           public IActionResult GetAllEmployee()
           {
               return Json(Employees);
           }

       }
   }
```

> **نکته!!**
>
> **قبل از اضافه نمودن این Tag Helper به View حتما پروژه را Build نمایید.**

سپس باید یک View با نام CreateEmployee.cshtml در مسیر Views/Employee ایجاد نمایید:

CreateEmployee.cshtml:

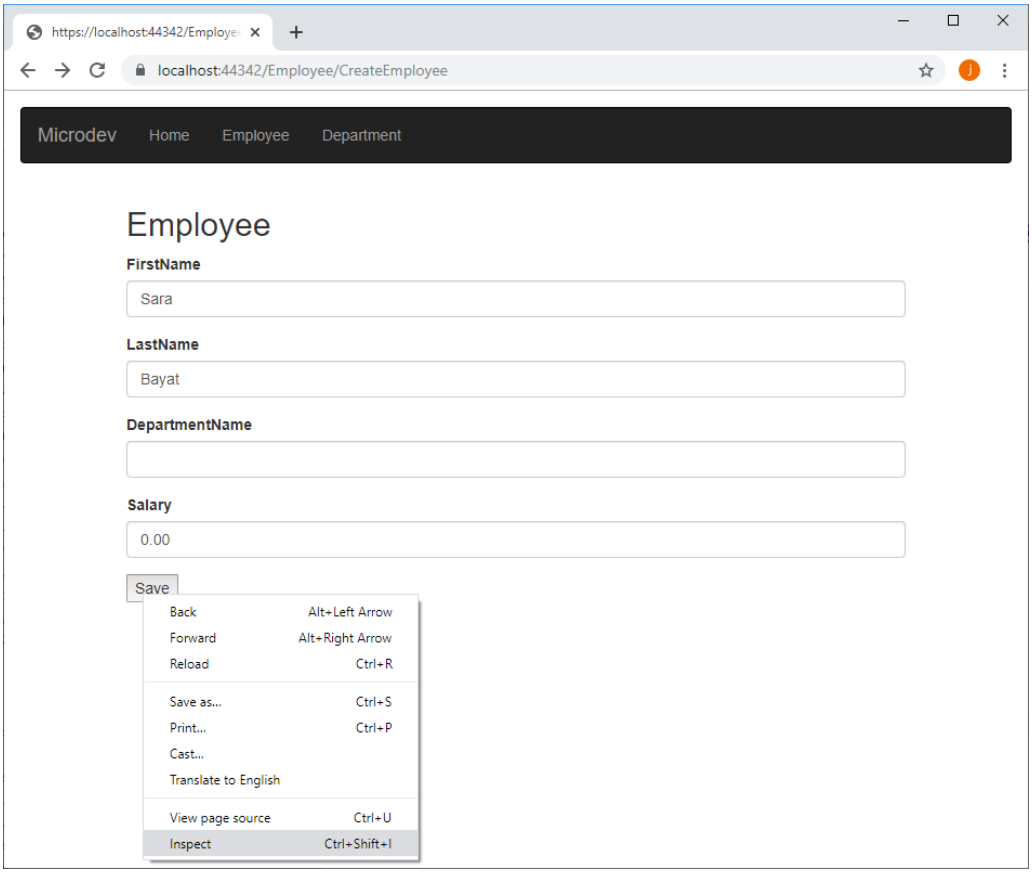
```
@model EmployeeViewModel
<div class="Container">
    <h2>Employee </h2>
    <form asp-action="SaveEmployee" asp-controller="Home">
        <div class="form-group">
            <label asp-for="FirstName"></label>
            <input class="form-control" asp-for="FirstName" />
            <span asp-validation-for="FirstName"></span>
        </div>
        <div class="form-group">
            <label asp-for="LastName"></label>
            <input class="form-control" asp-for="LastName" />
            <span asp-validation-for="LastName"></span>
        </div>
        <div class="form-group">
            <label asp-for="DepartmentName"></label>
            <input class="form-control" asp-for="DepartmentName" />
            <span asp-validation-for="DepartmentName"></span>
        </div>
        <div class="form-group">
            <label asp-for="Salary"></label>
            <input class="form-control" asp-for="Salary" />
            <span asp-validation-for="Salary"></span>
        </div>
                                استفاده از Tag Helper سفارشی
        <button microdev-custom-button="Save">Save</button>
    </form>
```

```
</div>
```

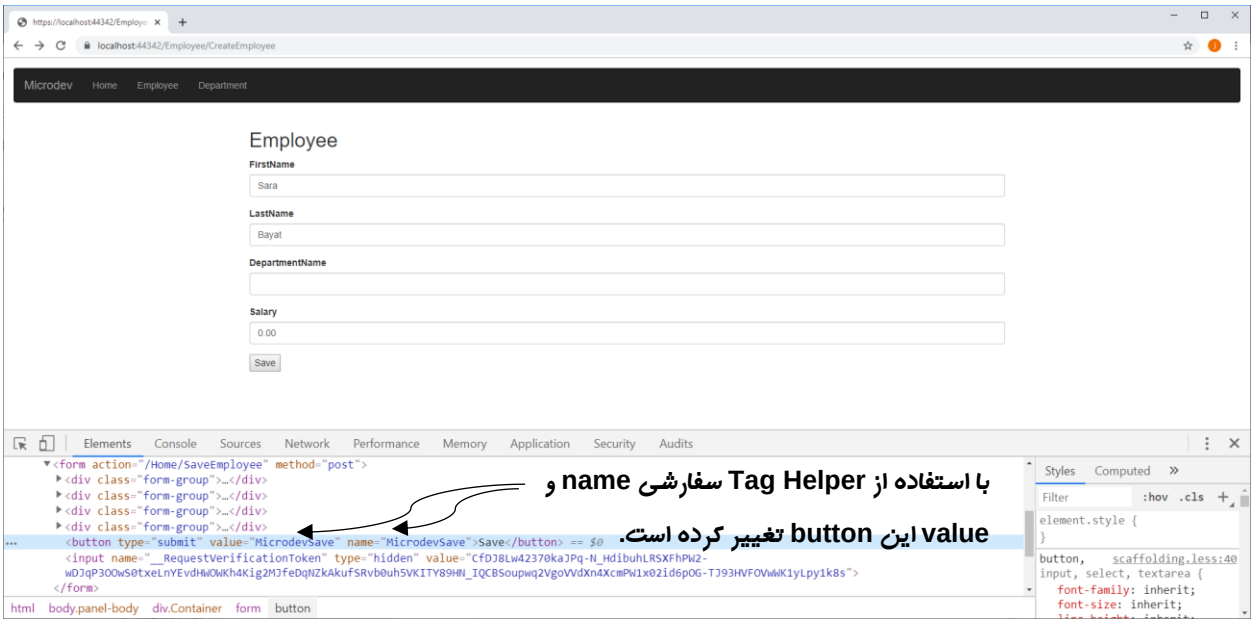
لطفا اپلیکیشن را اجرا کنید و وارد آدرس پایین شوید.

https://localhost:44342/Employee/CreateEmployee

حالا بر روی Save راست کلیک نمایید و گزینه‌ی Inspect را انتخاب کنید.



همانطور که در تصویر پایین می‌بینید، نام و مقدار این Button با MicrodevSave مقداردهی شده است.



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter6/Sample1

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **DI چیست و چرا مهم است؟**
- ✓ **چطور در ASP.NET Core می‌توان Dependency Injection را پیکربندی نمود؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What are Tag Helpers?

Q2: What are the ways of use Tag Helpers?

Q3: How to define custom Tag Helpers?

Q4: What are Form Tag Helpers?

Q5: What are Environment Tag Helpers?

Quiz

Q1: Tag Helpers can be added to _____.

1. A standard HTML element
1. A C# code
2. Data view
3. Designer view

Q2: What is @addTagHelper?

1. The @addTagHelper statement enables the built-in Razor View.
2. The @addTagHelper statement enables the built-in tag helpers.
3. The @addTagHelper statement enables the built-in HTML Helper.
4. All of the above

Q3: The _____tag helper provides an easy way to render different sections of HTML depending on the current environment.

1. Form
2. Link
3. Label
4. Environment

Q4: Which of tag helper is most often used with the link and script tag helpers.

1. Label
2. Form
3. Environment
4. All of the above

Q5: _____Tag Helpers expose two bounded properties, which allow you to specify the URL for the local copy of the script, and fallback test.

1. The Link
2. The Script
3. The Environment
4. Both 1 and 2

Q6: The _____Tag Helper uses the bounded properties asp-fallback-src and asp-fallback-test.

1. Link
2. Script
3. Environment
4. Both 1 and 2

Q7: Which of tag helper allowing you to test that a CSS class?

1. Environment
2. Form
3. Link
4. Script

Q8: Which of tag helper is used to generate the caption?

1. Label
2. Form
3. Link
4. Script

Q9: A Tag Helper is a class that inherits from_____.

1. TagHelper
2. Process
3. TagHelperContext
4. Both 1 and 3

Q10: Which of method define Tag Helpers behavior?

1. ProcessAsync
2. Process
3. TagHelperContext
4. Both 1 and 2

Answer

1-Correct Answer: A standard HTML element

2-Correct Answer: The @addTagHelper statement enables the built-in tag helpers.

3-Correct Answer: Environment

4-Correct Answer: Environment

5-Correct Answer: Both 1 and 2

6-Correct Answer: Script

7-Correct Answer: Link

8-Correct Answer: Label

9-Correct Answer: TagHelper

10-Correct Answer: Both 1 and 2

خلاصه فصل

- **Tag Helperها به شـما امکان می‌دهند تا مدل خود را به تگ‌های HTML متصـل و HTML داینامیک را راحت‌تر تولید نمایید.**
- **Tag Helperها می‌توانند یا با اسـتفاده از Attributeها به HTML متصـل شـوند و یا اینکه المنت‌های مستقل باشند.**
- **Tag Helperها می‌توانند المنت‌ها را سفارشی و Attributeهایی را به آن‌ها اضافه نمایند. این کار می‌تواند کدنویسی را به میزان زیادی کاهش دهد.**
- **می‌توانید Attributeهای asp-action و asp-controller را به تگ <form> اضـافه کنید تا اکشن درون URL با استفاده از مسیریابی تنظیم شود.**
- **می‌توانید با اسـتفاده از asp-for، Label Tag Helper را به تگ <label> اتصـال دهید. این Tag Helper باعث می‌شود تا بتوانید Lable را به یک Propertyی مدل خود متصل نمایید و هر آنچه را که در DataAnnotation [Display] این Property نوشـته شـده، در این Label نمایش دهید.**
- **Environment Tag Helper به شما این امکان را می‌دهد تا براساس محیط فعلی اپلیکیشن HTML خود را با شرایط مختلف Render نمایید.**

فصل هفتم: تزریق وابستگی چیست؟

آنچه خواهید آموخت

- DI چیست؟
- اهداف و مزایای DI چیست؟
- تزریق وابستگی در ASP.NET Core
- استفاده از تزریق وابستگی
- مراحل ایجاد کدهای loosely coupled
- طول عمر سرویس چیست؟
- پیاده‌سازی‌های مختلف از یک سرویس

DI چیست؟

داشتن وابستگی بین اجزای یک برنامه، امری اجتناب ناپذیر است. معمولا یک اپلیکیشن از کلاس‌های بسیاری ساخته شده که هر یک از آن‌ها به کلاس‌های دیگری وابستگی دارند و همانطور که می‌دانید اگر در هر کلاس به صورت دستی اشیاء را ایجاد کنیم، این اشیاء می‌توانند غیر قابل کنترل شوند.

پس چاره چیست؟

اینجاست که تزریق وابستگی وارد بازی می‌شود و می‌تواند به کاهش این زنجیره وابستگی کمک کند.

DI یک الگوی طراحی است که اجازه می‌دهد نمونه‌هایی[۸] از اشیا را در زمان اجرا به اشیاء دیگر منتقل کنید. به این ترتیب، به راحتی می‌توان گراف بسیار پیچیده‌ای از اشیا را ایجاد نمود.

اهداف و مزایای DI چیست؟

نرم‌افزارهای موفق باید قادر به تغییر باشند بنابراین شما نیاز به اضافه کردن Featureهای جدید و گسترش Featureهای موجود دارید. همچنین Componentهای Tightly Coupled، تقریبا اپلیکیشن را برای تست و نگهداری غیرممکن می‌کنند. **به عنوان مثال:** اگر شما Instanceهای یک کلاس را با استفاده از عملگر new ایجاد کنید، کدهای خود را به کلاس‌های دیگر لینک داده و باعث ایجاد Coupling شده‌اید.

هدف از اکثر متدهای برنامه‌نویسی این است که، تا حد امکان نرم‌افزارهایی با کارایی[۹] بالا ارائه دهند. یکی از جنبه این کار این است که کد قابل نگهداری باشد.

همانطور که می‌دانید، یک روش عالی برای نگهداری نرم‌افزار، نوشتن کدهای Loose Coupling است و DI یک تکنیک برای فراهم کردن این قابلیت می‌باشد.

نکته!!

Coupling یک مفهوم مهم در برنامه‌نویسی شی‌گراست و اشاره به این دارد که عملکرد یک کلاس چطور وابسته به کلاس‌های دیگر می‌باشد. Loose coupling باعث می‌شود کد قابل گسترش، و قابل نگهداری باشد و به شما این امکان را می‌دهد تا نرم افزار را شبیه قطعات الکترونیکی بسازید و با این روش انعطاف‌پذیری و کارآمدی نرم‌افزار را بالا ببرید.

---

[۸] Instances

[۹] Efficiently

نتیجه‌گیری: هدف تزریق وابستگی این است که به شما کمک کند تا کدهای Loosely Coupled و Testable بنویسید.

مزایای DI عبارتند از:

- **یک مزیت کلیدی این رویکرد این است که Unit Testing را بسیار ساده‌تر کرده است.**
- **بدون نیاز به نوشتن کد مجدد، می‌توان Serviceها را با یکدیگر تعویض کرد.**
- **کد را می‌توان گسترش داد و قابل استفاده‌ی مجدد نمود.**
- **کدها را می‌توان به صورت موازی توسعه داد.**
- **انعطاف‌پذیری برای انتخاب اینکه دقیقا چطور کامپوننت‌ها را در اپلیکیشن ترکیب کنید، یکی از مهمترین نقاط قوت DI است.**

تزریق وابستگی[۱۰] در ASP.NET Core

تزریق وابستگی (DI) یک رویکرد جایگزین جهت ایجاد کامپوننت‌های Loosely Coupled است که با پلتفرم ASP.NET Core تجمیع شده و به صورت خودکار توسط MVC استفاده می‌شود. این بدین معنیست که کنترلرها و سایر کامپوننت‌ها، نیازی به دانستن Typeهای مورد نیاز خود ندارند.

به طور خلاصه، DI یک Design Pattern قدرتمند است که Flexibility وTestability را به ASP.NET Core MVC اضافه نموده است. این Design Pattern به شما امکان می‌دهد تا Instance کلاس‌ها را از طریق Configuration انجام دهید و با این روش از Dependency Coupling استفاده نمایید.

با DI مسئولیت ایجاد و مدیریت Instance کلاس‌ها به یک Container واگذار می‌شود. هر کلاس اعلام می‌کند که به چه کلاس‌های دیگری وابستگی دارد، سپس Container می‌تواند آن وابستگی‌ها را در زمان اجرا فراهم و در صورت لزوم آن‌ها را پاس دهد.

الگوی Dependency Injection فرمی از IoC[۱۱] است، بنابراین احتمالا پیاده‌سازی‌های DI را به عنوان IoC Container هم بشنوید. IoC Container به شما امکان می‌دهد که Interfaceها را با پیاده‌سازی مرتبط به خودش رجیسترکنید. سپس Container می‌تواند یک Instance مرتبط را برای یک درخواست Interface فراهم کند.

---

[۱۰] Dependency Injection

[۱۱] Inversion Of Control

ASP.NET Core یک IoC container توکار دارد که البته از IoC Containerهای دیگر(مانند Autofac)هم پشتیبانی می‌کند.

نکته!!

چه از IoC container داخلی استفاده کنید، چه از یک کانتینر ثالث استفاده نمایید، رجیستر کردن کلاس‌ها باید در Startup.cs انجام شود.

استفاده از تزریق وابستگی

تزریق وابستگی (DI)یک مفهوم بسیار ساده و با اهمیت است. ASP.NET Core برای پیکربندی Componentهای داخلی خود و سرویس‌های سفارشی شما از DI استفاده می‌کند. به همین منظور برای اینکه بتوانید از این کامپوننت‌ها و سرویس‌ها در زمان اجرا استفاده کنید، DI container باید از کلاس‌های مورد نیاز شما آگاهی داشته باشد. بنابراین برای استفاده از تزریق وابستگی‌ها به چهار کلاس نیاز دارید:

۱) **Interface**: برای تعریف سرویس خارجی. **به عنوان مثال: IEmployeeService.cs**.
۲) **Concrete Class**: برای پیاده‌سازی Interface بالا. **به عنوان مثال: EmployeeService.cs**.
۳) **کلاس Startup.cs**: برای پیکربندی سرویس.
۴) **Consumer Class**: برای استفاده از سرویس رجیستر شده. **به عنوان مثال: EmployeeController.cs**.

مراحل ایجاد کدهای loosely coupled

بیایید با هم نمونه‌ای از نحوه استفاده DI را، در یک برنامه ASP.NET Core MVC ببینیم.

۱) تعریف Interface

ابتدا باید یک Interface جهت مشخص کردن سرویسی که تنها کلاس Consumer به آن وابسته است تعریف و متدهای موردنظرتان را در آن قرار دهید.

بنابراین Folderی به نام Servicesایجاد و سپس در آن یک Interface با نام IEmpoyeeService اضافه نمایید.

IEmployeeService.cs:

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

```
using Microdev.ASPNETCore.ViewModels;
namespace Microdev.ASPNETCore.Services
{
    public interface IEmployeeService
    {
        IEnumerable<EmployeeViewModel> GetAllEmployee();
        EmployeeViewModel GetEmployee(int employeeId);
        EmployeeViewModel CreateEmployee();
    }

}
```

۲) پیاده سازی اینترفیس در یک Concrete Class

در اصطلاحات DI، ما اغلب در مورد سرویس‌ها صحبت می‌کنیم. یک سرویس معمولاً یک Abstraction است که سرویس ما را ارائه می‌دهد.

پس از تعریف Interface، شما باید کلاسی که قرار است کار اصلی را انجام دهد ایجاد و سپس Interface تعریف شده بالا را در آن پیاده‌سازی کنید.

بنابراین در فولدر Services یک کلاس به نام EmpoyeeService اضافه کنید.

EmpoyeeService.cs

پیاده سازی اینترفیس

```
using System.Collections.Generic;
using System.Linq;
using Microdev.ASPNETCore.ViewModels;
namespace Microdev.ASPNETCore.Services
{
  public class EmployeeService : IEmployeeService
  {
    public IEnumerable<EmployeeViewModel> Employees { get; }
     public EmployeeService()
     {
         Employees = new List<EmployeeViewModel>
         {
             new EmployeeViewModel{
                 EmployeeId = 100,
                 FirstName = "Zahra",
                 LastName = "Bayat",
                 DepartmentName = "Raveshmand",
                 Salary=1000000
                },
             new EmployeeViewModel{
                 EmployeeId = 101,
                 FirstName = "Ali",
                 LastName = "Bayat",
                 DepartmentName = " Raveshmand",
```

```
                    Salary=1000000
                },
            };
    }

    public IEnumerable<EmployeeViewModel> GetAllEmployee()
    {
        return Employees;
    }
    public EmployeeViewModel GetEmployee(int employeeId)
    {
        return Employees.FirstOrDefault(x => x.EmployeeId == employeeId);
    }

    public EmployeeViewModel CreateEmployee()
    {
        return new EmployeeViewModel ();
    }
 }
}
```

این متد لیست تمام کارمندان را برمی‌گرداند

این متد کارمند موردنظر را براساس Id می‌یابد و سپس برمی‌گرداند.

این متد یک Instance از EmployeeViewModel را جهت ایجاد یک کارمند برمی‌گرداند.

۳) پیکربندی DI در ASP.NET Core

اکنون که اهمیت استفاده از DI را درک کردید، باید بدانید که چطور آن را پیکربندی نمایید.

با شروع اپلیکیشن، سرویس‌ها در Container رجیستر می‌شوند، بنابراین بعد از ایجاد IEmployeeService و EmployeeService، مرحله بعدی پیکربندی IoC Container است. IoC Container باید بداند چه کلاس‌هایی برای هر Interface استفاده شوند و طول عمر هر کلاس چقدر است. پس از انجام این کار، IoC Container می‌تواند بطور خودکار Instanceهایی از کلاس‌ها را ایجاد و در زمان اجرا تزریق کند.

همانطور که قبلاً نیز اشاره شد، ASP.NET Core یک Container داخلی دارد که شما می‌توانید با استفاده از متد ConfigureService کلاس Startup، از آن بهرمند شوید.

شما قبلاً با متد ConfigureServices آشنا شدید، بنابراین اکنون باید EmployeeService را در DI container رجیستر کنید.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IEmployeeService, EmployeeService >();
    services.AddControllersWithViews();
}
```

رجیستر کردن سرویس EmployeeService

نکته!!

Namespace پایین را در کلاس Startup اضافه کنید:

```
using Microdev.ASPNETCore.Services;
```

وظیفه Map کردن Interfaceهای مربوط به هر کلاس در ASP.NET Core، برعهده‌ی کامپوننتی به نام Service Provider است. کلاس IServiceCollection شامل متدهای بسیاری است که نحوه مدیریت Instanceها را ساده‌تر و کار رجیستر کردن Typeها در IoC container را مشخص می‌کند.

در عبارات فوق، متد AddTransient سه قطعه اطلاعات را به DI container ارائه می‌دهد:

۱) **نوع سرویس:** این پارامتر مشخص می‌کند، از چه کلاس یا Interfaceی به عنوان Dependency استفاده می‌شود. معمولا در اینجا یک Interface و گاهی هم یک کلاس معمولی استفاده می‌شود.
۲) **نوع پیاده‌سازی:** کلاسی است که Container برای تحقق وابستگی باید آن را ایجاد کند.
۳) **طول عمر:** طول عمر مدت زمانی که یک Instance از سرویس، باید استفاده شود را مشخص می‌کند. مقدار این پارامتر یکی از سه مقادیر Transient, Singleton, Scoped می‌باشد.

نکته!!

اگر رجیستر کردن سرویس ی که مورد نیاز فریم ورک یا اپلیکیشن می‌باشد را فراموش نمایید، در زمان اجرا InvalidOperationException دریافت خواهید کرد.

طول عمر سرویس چیست؟

طول عمر یک سرویس، مدت زمان زنده ماندن یک Instance از سرویس، قبل از ایجاد یک نمونه جدید دیگر است.

در نظر داشته باشید، شما باید در طول رجیستر شدن سرویس DI، طول عمر آن را تعیین کنید. بنابراین، هر زمان که DI Container یک سرویس خاص رجیستر شده را درخواست کند، یکی از دو اتفاق پایین رخ می‌دهد:

۱) **Instance جدیدی از سرویس ایجاد و بازگرداند می‌شود.**
۲) **یک Instance موجود از سرویس بازگشت داده خواهد شد.**

شما می‌توانید هنگام رجیستر شدن سرویس در ASP.NET Core ، سه طول عمر مختلف را تعیین کنید:

- **Transient :** با این چرخه عمر، هر بار که یک سرویس درخواست شود، یک **Instance** جدید ایجاد می‌شود. این چرخه‌عمر ممکن است باعث تخصیص و تفکیک مکرر حافظه شود و در نتیجه در صورت استفاده زیاد، می‌تواند تأثیر منفی بگذارد.

```
services.AddTransient<TService,TImplementation>().
```

> **نکته!!**
>
> این چرخه عمر برای سرویس‌های **lightweight** و **stateless** که هیچ وضعیتی را در خود نگه نمی‌دارند و سریع **Instantiate** می‌شوند مناسب است.

- **Scoped** : سرویس‌هایی با طول عمر **Scoped**، تنها یکبار در طی هر درخواست ایجاد می‌شوند.

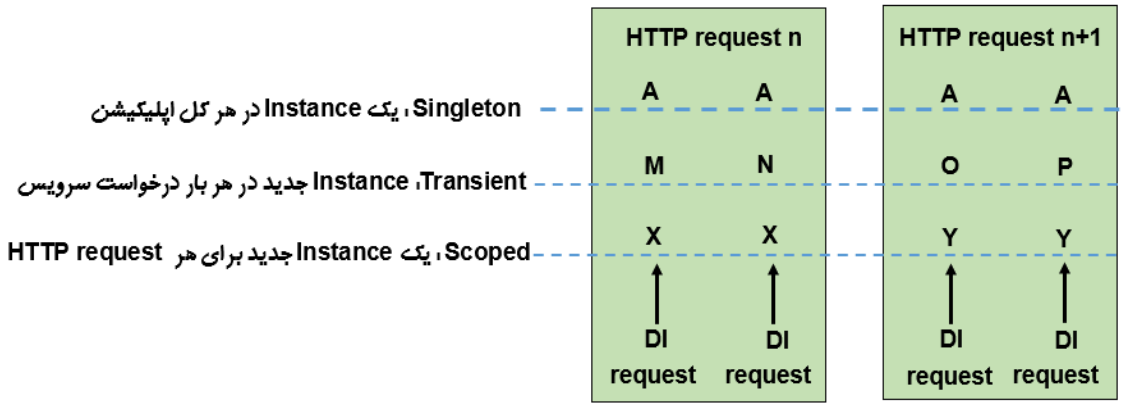
```
services.AddScoped<TService,TImplementation>().
```

تفاوت Scoped با Transient: در **Transient** اگر در یک **Request**، یک سرویس دو بار درخواست شود، هر بار **Instance** جدید ایجاد خواهد شد، اما در **Scoped** بابت هر **Request** تنها یک بار از سرویس **Instance** ایجاد خواهد شد و تا پایان **Request** هیچ **Instance** جدید دیگری ایجاد نخواهد شد.

- **Singleton**: در این چرخه عمر، اولین‌باری که سرویس درخواست می‌شود، یک **Instance** ایجاد شده و همیشه همان **Instance** به سرویس‌ها تزریق می‌شود. یا به عبارتی، شما در طول کل حیات اپلیکیشن تنها یک نمونه از سرویس را دریافت خواهید کرد.

```
services.AddSingleton<TService,TImplementation>().
```

> **نکته!!**
>
> الگوی **singleton** برای اشیایی که با اهمیت هستند و یا وضعیت مناسبی ندارند مفید است.

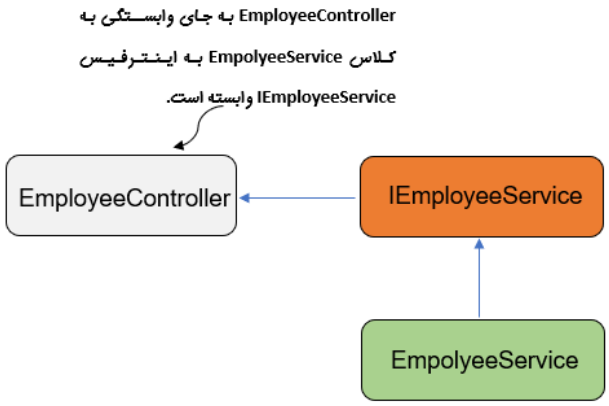


۴) تزریق سرویس در Controller

پس از رجیستر شدن، DI container می‌تواند سرویس را به هر جایی تزریق کند. متداول‌ترین روش برای تزریق وابستگی‌ها، استفاده از تکنیکی به نام Constructor Injection است. در این روش آبجکت‌ها به عنوان پارامتر، به سازنده کلاس پاس داده شده و از این طریق تزریق می‌شوند.

با الگوی Constructor Injection، تمام وابستگی‌ها هنگام ایجاد کلاس ساخته می‌شوند. این الگو، تنها موردی است که توسط ASP.NET Core IoC Container پشتیبانی می‌شود.

نحوه استفاده از الگوی Constructor Injection بدین صورت است که کلاس مصرف کننده باید یک سازنده public تعریف کند و سپس سرویس را از ورودی این سازنده دریافت نماید.



حال برای استفاده از این الگو باید EmployeeController را کمی تغییر دهیم. مهمترین تغییر، اضافه کردن یک سازنده و یک متغیر Private برای نگه داشتن رفرنس وابستگی خارجی است.

و قدم بعدی تغییر متدهای GetAllEmployee، GetEmployee و CreatEmployee می‌باشد.

EmployeeController:

```
using Microdev.ASPNETCore.Services;
using Microsoft.AspNetCore.Mvc;

namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController: Controller
    {
        private readonly IEmployeeService _service;

        public EmployeeController(IEmployeeService  service)
        {
            _service = service;
        }
        public IActionResult CreateEmployee()
        {
            var model = _service.CreateEmployee();
            return View(model);
        }

        public IActionResult GetEmployee(int employeeId)
        {
          var model =_service.GetEmployee(employeeId);
            return View(model);
        }

        public IActionResult GetAllEmployee()
        {
            var model = _service.GetAllEmployee();
            return View(model);
        }

    }

}
```

تزریق وابستگی از طریق سازنده

ریختن Instanceی از سرویس درون یک فیلد Readonly

حالا می‌توانید Instance جدید از EmployeeViewModel را از سرویس برگردانید.

با این سرویس می‌توانید کارمند موردنظر خود را با استفاده از سرویس بیابید.

حالا می‌توانید لیست کارمندان را با استفاده از سرویس برگردانید.

نکته اصلی این موضوع این است که، کدهای درون EmployeeController ، به نحوه اجرای وابستگی اهمیت نمی‌دهند. حالا کد اپلیکیشن از پیاده‌سازی مستقل شده است.

در مثال بالا، روند اصلی Dependency Injection بدین صورت است:

۱) **MVC یک Request به اکشن‌متد GetAllEmployee در کنترلر Employee دریافت می‌کند.**

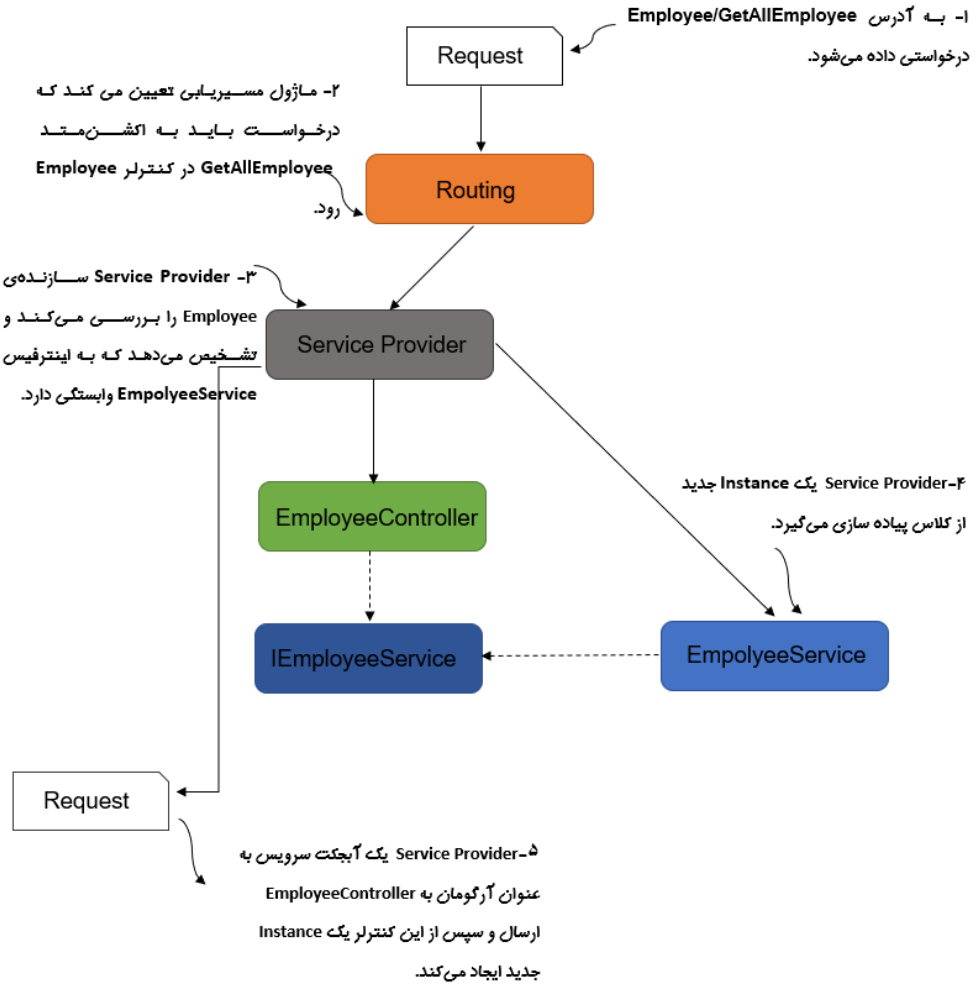
۲) **MVC، نمونه جدیدی از کلاس EmployeeController را از کامپوننت Service Provider ASP.NET Core درخواست می‌کند.**

۳) Service Provider سازنده کلاس EmployeeController را بررسی می‌کند و می‌گوید این کلاس یک وابستگی به اینترفیس IEmployeeService دارد.

۴) سپس Service Provider کلاسی که اینترفیس IEmployeeService را پیاده‌سازی کرده، می‌یابد و یک Instance جدید از این کلاس (EmployeeService) ایجاد می‌کند.

۵) سپس این Instance جدید را به عنوان آرگومان به سازنده EmployeeController پاس می‌دهد.

۶) در پایان، Service Provider شیء EmployeeController ایجاد شده را به MVC باز می‌گرداند، تا از آن برای رسیدگی به درخواست HTTP ورودی استفاده کند.



نکته!!

برای تزریق وابستگی‌ها، گزینه دیگری به نام **Property Injection** وجود دارد است که یک پراپرتی **public**با یک **Attribute** تزئین شـــده و نشـــان می‌دهد که این پراپرتی‌باید در زمان اجرا توســـط **Container** تنظیم شود. **Property Injection** از **Construction Injection** کمتر استفاده می‌شود و توسط تمام **IoC container**ها هم پشتیبانی نمی‌شود.

قبـل از اجرای برنـامـه، لطفـاً فـایـل **GetAllEmployee.cshtml** و **GetEmployee.cshtml** را در **View/Employee** اضافه کنید.

GetAllEmployee.cshtml:

```
@model List<EmployeeViewModel>

<div class="Container">
    <h3> Employee List:</h3>
    <table class="table">
        <tr>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Salary</th>
            <th>Department Name</th>
        </tr>


        @foreach (var item in Model)
        {
            <tr>
                <td>@item.FirstName</td>
                <td>@item.LastName</td>
                <td>@item.Salary</td>
                <td>@item.DepartmentName</td>
            </tr>
        }

    </table>

</div>
```

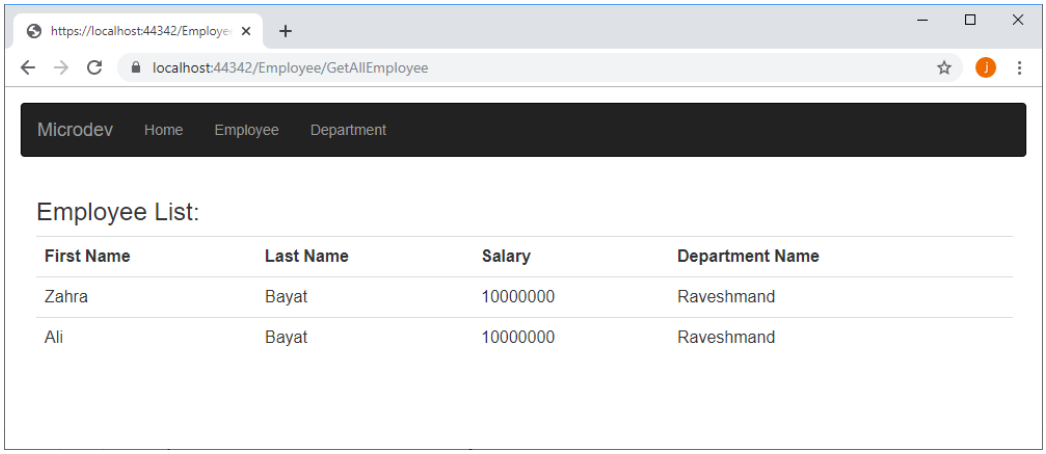
GetEmployee.cshtml:

```
@model EmployeeViewModel

<div class="Container">
    <h3> @ViewData["Title"]</h3>
    <table class="table">
        <tr>
```

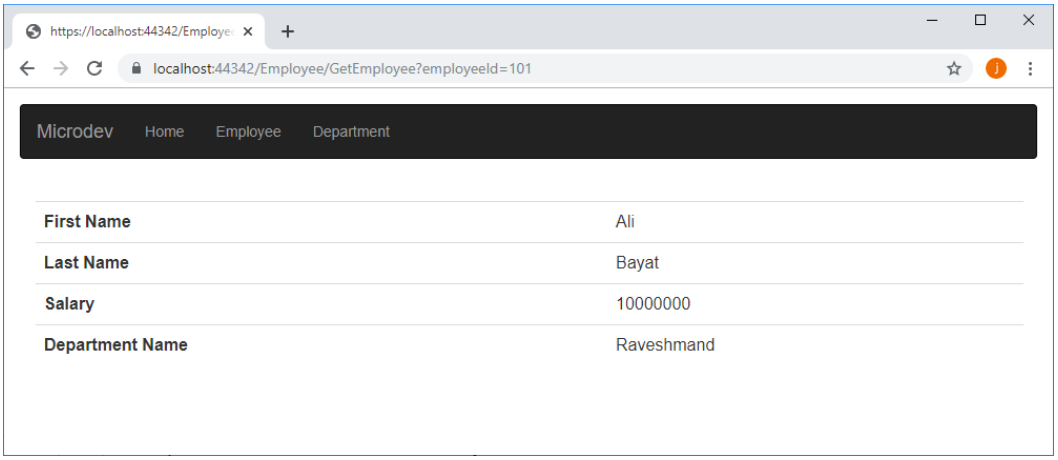
```
            <th>First Name</th>
            <td>@Model.FirstName</td>

        </tr>
        <tr>
            <th>Last Name</th>
            <td>@Model.LastName</td>
        </tr>
        <tr>
            <th>Salary</th>
            <td>@Model.Salary</td>
        </tr>
        <tr>
            <th>Department Name</th>
            <td>@Model.DepartmentName</td>
        </tr>

    </table>


</div>
```

حالا اپلیکیشن را اجرا و وارد مسیرهای پایین شوید:

https://localhost:44342/Employee/GetAllEmployee



https://localhost:44342/Employee/GetEmployee?employeeId=101

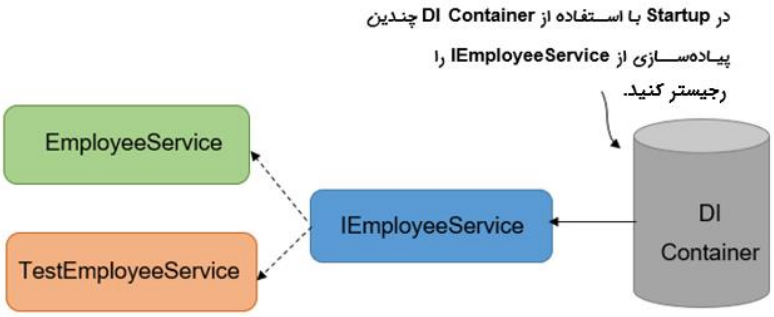


مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter7/Sample1

پیاده‌سازی‌های مختلف از یک سرویس

یکی از مزایای استفاده از اینترفیس‌ها این است که می‌توانید از یک سرویس چندین پیاده‌سازی داشته باشید. **به عنوان مثال:** تصـــور کنید که می‌خواهیدیک ورژن عمومی از IEmployeeService ایجاد کنید تا بتوانید داده‌ها را در یک محیط Test یا Production، در حالت‌های مختلف بازیابی کنید.



بیایید یک مثال ساده را در نظر بگیریم. شما قبلاً اینترفیسی با نام IEmployeeService.cs داشتید که توسط کلاس EmployeeService.cs پیاده‌سازی شده بود:

IEmployeeSevice.cs:

```
using System.Collections.Generic;
using Microdev.ASPNETCore.ViewModels;
namespace Microdev.ASPNETCore.Services
{
    public interface IEmployeeService
    {
```

```
        IEnumerable<EmployeeViewModel> GetAllEmployee();
        EmployeeViewModel GetEmployee(int employeeId);
        EmployeeViewModel CreateEmployee();
    }

}
```

EmployeeSevice.cs

```
public class EmployeeService : IEmployeeService
using System.Collections.Generic;
using System.Linq;
using Microdev.ASPNETCore.ViewModels;
namespace Microdev.ASPNETCore.Services
{
  public class EmployeeService : IEmployeeService
  {
    public IEnumerable<EmployeeViewModel> Employees { get; }
     public EmployeeService()
     {
           Employees = new List<EmployeeViewModel>
             {
                 new EmployeeViewModel{
                     EmployeeId = 100,
                     FirstName = "Zahra",
                     LastName = "Bayta",
                     DepartmentName = "Raveshmand",
                     Salary=1000000
                    },
                 new EmployeeViewModel{
                     EmployeeId = 101,
                     FirstName = "Ali",
                     LastName = "Bayat",
                     DepartmentName = "Raveshmand",
                     Salary=1000000
                 },
             };
     }

   public IEnumerable<EmployeeViewModel> GetAllEmployee()
   {
       return Employees;
   }

   public EmployeeViewModel GetEmployee(int employeeId)
    {
       return Employees.FirstOrDefault(x => x.EmployeeId == employeeId);
    }

    public EmployeeViewModel CreateEmployee()
```

```
    {
        return new EmployeeViewModel ();
    }
  }
}
```

حالا بیایید یک پیاده‌سازی دیگر از این اینترفیس داشته باشیم. بنابراین در فولدر Services کلاس دیگری با نام TestEmployeeService ایجاد و سپس همانند کد پایین یک پیاده‌سازی جدید از این اینترفیس قرار دهید:

TestEmployeeService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Services
{

     public class TestEmployeeService : IEmployeeService
    {
        public IEnumerable<EmployeeViewModel> Employees { get; }
        public TestEmployeeService()
        {
            Employees = new List<EmployeeViewModel>
            {
                new EmployeeViewModel{
                    EmployeeId = 100,
                    FirstName = "Zahra",
                    LastName = "Bayat",
                    DepartmentName = "Raveshmand",
                    Salary=1000000
                   },
                new EmployeeViewModel{
                    EmployeeId = 101,
                    FirstName = "Ali",
                    LastName = "Bayat",
                    DepartmentName = "Raveshmand",
                    Salary=3000000
                },
                 new EmployeeViewModel{
                    EmployeeId = 102,
                    FirstName = "Sara",
                    LastName = "Sadeghi",
                    DepartmentName = "Raveshmand",
                    Salary=2500000
```

```
                },
                new EmployeeViewModel{
                    EmployeeId = 103,
                    FirstName = "Amin",
                    LastName = "Eshaghi",
                    DepartmentName = "Raveshmand",
                    Salary=5000000
                },
            };
        }

        public IEnumerable<EmployeeViewModel> GetAllEmployee()
        {
            return Employees.Where(x=>x.Salary>2500000).ToList();
        }

        public EmployeeViewModel GetEmployee(int employeeId)
        {
            return Employees.FirstOrDefault(x => x.EmployeeId == employeeId);
        }

        public EmployeeViewModel CreateEmployee()
        {
            return new EmployeeViewModel {DepartmentName= "Raveshmand" };
        }
    }

}
```

یک سؤال؟؟

اکنون چگونه این پیاده‌سازی‌ها را در Container رجیستر کنیم؟

اول از همه، برای شرط بازگشت یک سرویس، بهتر است یک enum خاص تعریف شود. پس در فولدر Models یک کلاس به نام EnviromentServiceType اضافه نمایید:

EnviromentServiceType.cs:

```
namespace Microdev.ASPNETCore.Models
{
    public enum EnviromentServiceType
    {
        ProductionEmployeeService,
        TestEmployeeService
    }

}
```

این Enum باید دو مقدار جهت چک کردن حالت Production و Testداشته باشد.

حالا باید هر دو سرویس ارث‌بری شده از اینترفیس IEmployeeService را رجیستر کنیم:

ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<EmployeeService>();
    services.AddTransient<TestEmployeeService>();
    services.AddControllersWithViews();

}
```

ریجستر شدن دو سرویس EmployeeService , TestEmployeeService برای اجرا در محیط‌های مختلف.

و درنهایت باید بازگشت این سرویس‌ها همراه با شرط باشد:

ConfigureServices:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<EmployeeService>();
  services.AddTransient<TestEmployeeService>();
   services.AddTransient<Func<EnviromentServiceType,
  IEmployeeService>>(serviceProvider => key =>
  {
    switch (key)
    {
      case EnviromentServiceType.ProductionEmployeeService:
        return serviceProvider.GetRequiredService<EmployeeService>();
      case EnviromentServiceType.TestEmployeeService:
        return serviceProvider.GetRequiredService<TestEmployeeService>();
      default:
          throw new NotImplementedException($"Service of type {key} is not
    implemented.");
    }
  });
  services.AddControllersWithViews();
}
```

اعمال شرط برای اجرا شدن سرویس‌های EmployeeService و TestEmployeeService در محیط‌های مختلف.

خوب، حالا چگونه می‌توانید این پیاده‌سازی‌ها را به EmployeeController خود تزریق کنید؟

بیایید EmployeeController را هم کمی تغییر دهیم.

EmployeeController.cs:

```
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.Services;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
using System;

namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController: Controller
    {
        readonly Func<EnviromentServiceType, IEmployeeService> _service;

        public EmployeeController(Func<EnviromentServiceType,
        IEmployeeService> enviromentServiceType)
        {
            _service = enviromentServiceType;
        }

        public IActionResult CreateEmployee()
        {
            var service =
_service(EnviromentServiceType.TestEmployeeService);
            var model = service.CreateEmployee();
            return View(model);
        }
        public IActionResult GetEmployee(int employeeId)
        {
           var service = _service(EnviromentServiceType.TestEmployeeService);
            var model =service.GetEmployee(employeeId);
            return View(model);
        }

        public IActionResult GetAllEmployee()
        {
           var service = _service(EnviromentServiceType.TestEmployeeService);
            var model = service.GetAllEmployee();
            return View(model);


        }

    }
}
```

هم EmployeeService

و هم TestEmployeeService

هربار به سازنده تزریق می‌شوند.

این سه اکشن‌متد، سرویس

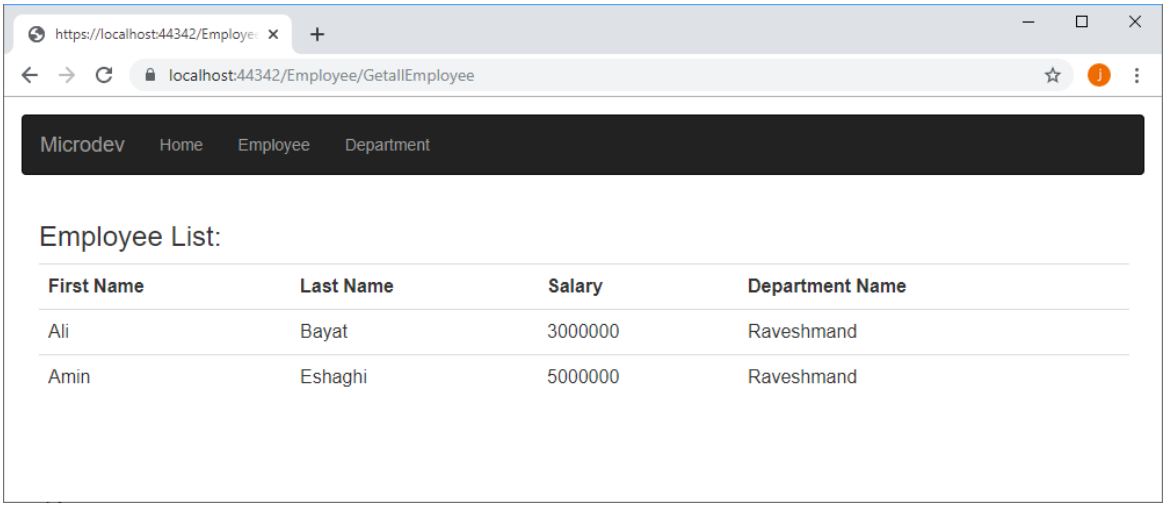
TestEmployeeService را از DI

Container درخواست می‌کنند و سپس

این سرویس [illegible]

اپلیکیشـــن را اجرا کنید و ببینید که اینبار اطلاعات از ســـرویس TestEmployeeService واکشـــی می‌شود.

https://localhost:44342/Employee/GetallEmployee



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter1/Sample2

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Entity Framework Core چیست؟**
- ✓ **چطور داده را با Entity Framework Core ذخیره کنیم؟**
- ✓ **Web API چیست و چه زمان باید از آن استفاده کرد؟**

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What is IoC container?

Q2: What is dependency injection?

Q3: Which of the pattern can be used to implement IoC?

Q4: What are the types of Dependency Injections?

Q5: What IoC containers for .NET application?

Q6: What is the Autofac container?

Q7: How to register a type with Autofac container?

Q8: Which of the lifetime manager is used to create singleton object?

Q9: What ConfigureServices() method does in Startup.cs?

Q10: How to define the lifetime of a service during DI service registration?

Quiz

Q1: DI stands for _____.

1. Dependency Injection
2. Dependency Inversion
3. Dependency Interface
4. All of the above

Q2: IoC stands for _____.

1. Inversion of Class
2. Invert Object Class
3. Inversion of Control
4. Inversion of Concept

Q3: IoC and DI are aimed to achieve _____.

1. Tight coupling
2. Loose coupling
3. Unit testing
4. Performance

Q4: IoC is a _____.

1. Design Pattern
2. Design Principle
3. Framework
4. None of the above

Q5: Dependency Injection is a _____.

1. Design Principle
2. Design Pattern
3. Code snippet
4. All of the above

Q6: What are the types of Dependency Injections?

1. Constructor Injection
2. Method Injection
3. Property Injection
4. All of the above

Q7: IoC Container is the _____.

1. Framework
2. Design Pattern
3. Design Principle

4. Third-party framework

Q8: Which of the following lifetime is definition: one instance is created per application?

1. Transient
2. Singleton
3. Scoped
4. None of the above

Q9: The _____ method in Startup class is used to registering services with IoC container.

1. ConfigureService
2. Configure
3. Main
4. All of the above

Answer

1-Correct Answer: Dependency Inversion

2-Correct Answer: Inversion of Control

3-Correct Answer: Loose coupling

4-Correct Answer: Design Principle

5-Correct Answer: Design Pattern

6-Correct Answer: All of the above

7-Correct Answer: Framework

8-Correct Answer: Singleton

9-Correct Answer ConfigureService

خلاصه فصل

- ✓ تزریق وابستگی (DI) یک عنصر مهم معماری در طراحی ASP.NET Core است.
- ✓ شما باید تمام وابستگی‌های فریم‌ورک را در Startup اضافه نمایید.
- ✓ IoC container مسئول ایجاد Instanceهایی از سرویس است. DI می‌دلند که چطور Instanceای از سرویس و تمام وابستگی‌های آن را بسازد و به سازنده‌ها پاس دهد.
- ✓ شما می‌تولنید در متد ConfigureService، با فراخوانی اکستنشن متدهایی که بر روی IServiceCollection قرار دارد، سرویس‌ها را درون Container رجیستر نمایید.
- ✓ برای رجیستر شدن یک سرویس به سه چیز نیاز دارید: نوع سرویس، نوع پیاده‌سازی و طول عمر.
- ✓ نوع سرویس مشخص می‌کند که کدام کلاس یا اینترفیس به عنوان وابستگی خواسته شود.
- ✓ نوع پیاده‌سازی مشخص می‌کند که Container چه کلاسی را باید برای درخواست وابستگی ایجاد نمایید.
- ✓ طول عمر، مدت زمانی است که باید یک Instanceای از سرویس استفاده شود.
- ✓ شما طول عمر سرویس را هنگام رجیستر سرویس تعریف می‌کنید.
- ✓ طول عمر سرویس سه حالت دارد: Transient، Scoped و Singletone.
- ✓ Transient : هر بار که یک سرویس درخواست می‌شود، یک Instance جدید ایجاد شود. این طول‌عمر ممکن است باعث تخصیص و تفکیک مکرر حافظه شود و در نتیجه در صورت استفاده زیاد، می‌تواند تأثیر منفی بگذارد.
- ✓ Scoped : سرویس‌هایی با طول عمر Scoped، تنها یکبار در طی هر درخواست ایجاد می‌شوند.

فصل هشتم: ایجاد WebAPI در ASP.NET Core

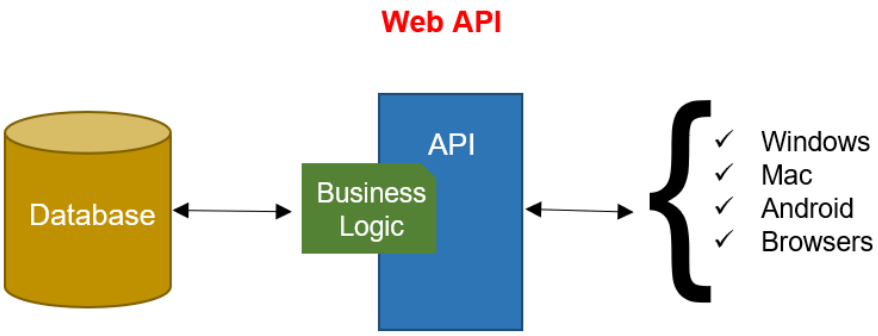
آنچه خواهید آموخت:

- Web API چیست و چه زمانی باید از آن استفاده کنید؟
- REST چیست و HTTP چگونه کار می‌کند؟
- ایجاد اولین اپلیکیشن Web API
- Entity Framework Core و Dapper
- Data Seeding چیست؟
- پیاده‌سازی CQRS
- تست APIها با استفاده از PowerShell

Web API چیست و چه زمانی باید از آن استفاده کنید؟

Web API تعدادی متد است، جهت دسترسی یا تغییر داده‌ها در یک سرور، که معمولا استفاده کنندگان نهایی این متدها، برنامه‌نویسان SPA و موبایل هستند.

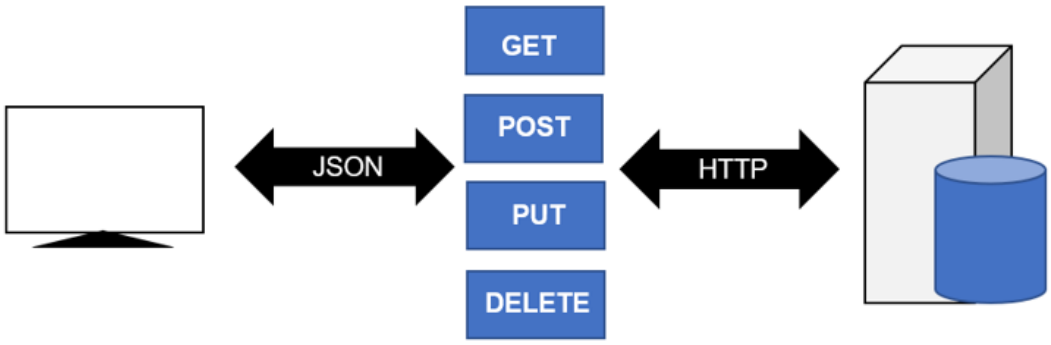
در دنیای مدرن امروزی APIها نقش اساسی در یکپارچگی سیستم‌ها دارند، زیرا دسترسی به اپلیکیشن را در سیستم‌های مختلف فراهم می‌کنند. بنابراین بسیار مهم است که طراحی API توسط یک برنامه‌نویس حرفه‌ای انجام شود.



REST چیست و HTTP چگونه کار می‌کند؟

Web APIها، از طریق HTTP فراخوانی می‌شوند. HTTP پروتکلی مبتنی بر متن است که در آن کلاینت برای ارتباط با سرور:

- **یک TCP Connection را باز می‌کند.**
- **درخواستی را به سرور ارسال نموده.**
- **سپس پاسخی را از سرور دریافت کرده.**
- **و در نهایت Connection را می‌بندد.**



کلاینت یک Request ارسال می‌کند. **HTTP Methodها** **سرور به کلاینت یک Response برمی‌گرداند.**

REST یک سبک معماری است که برقراری ارتباط بین کلاینت و سرور را از طریق پروتکل HTTP آسان نموده است.

نکته!!

HTTP از عملیات CRUD در سرور پشتیبانی می‌کند.

سرور شامل منطق بیزینسی می‌باشد که شما برای آن، کدنویسی می‌کنید و کلاینت هم می‌تواند یک اپلیکیشن موبایل، Angular یا حتی APIهای دیگر باشد.

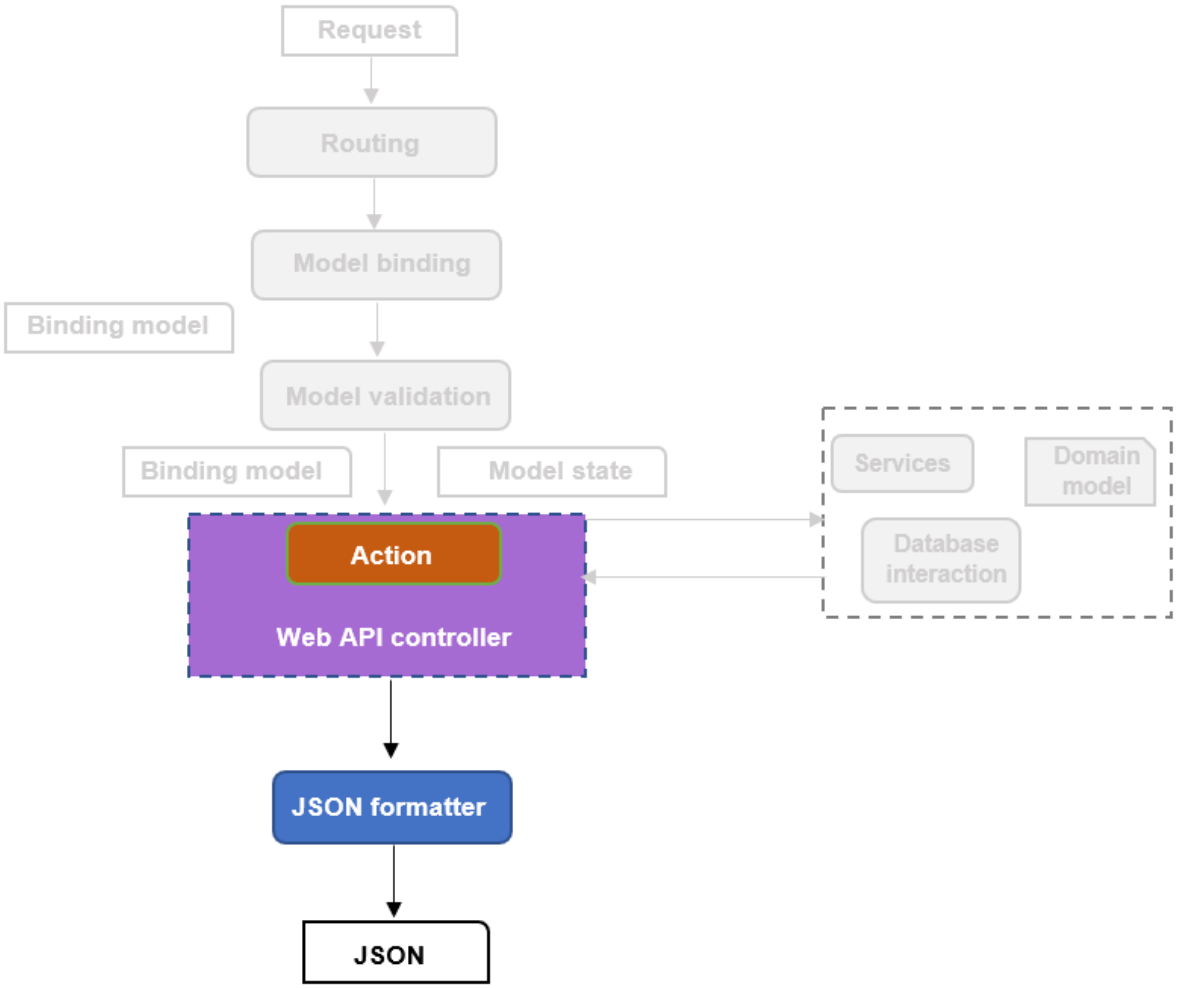
متداول‌ترین متدهای HTTP که در REST استفاده می‌شود:

- ✓ **GET: برای واکشی داده‌ها بدون تغییر داده‌های سرور.**
- ✓ **POST: برای ایجاد یک Resource جدید.**
- ✓ **DELETE: برای حذف یک منبع Resource.**
- ✓ **PUT: برای به روزرسانی داده های سرور.**

Controller و اکشن‌متدها

Web APIهای ASP.NET Core همانند وب‌اپلیکیشن‌های سنتی[۱۲]، از دیزاین پترن MVC پیروی می‌کنند. در اپلیکیشن‌های Web API برای دسته‌بندی منطقی اپلیکیشن از Controllerها استفاده می‌شود. همانطور که می‌دانید، هر Controller جهت تشکیل زیرساخت قسمتی از اپلیکیشن، دارای تعدادی اکشن‌متد است که هر اکشن‌متد می‌تواند با استفاده از IActionResult داده‌هایی را جهت تولید Response، برگرداند.

[۱۲] Traditional

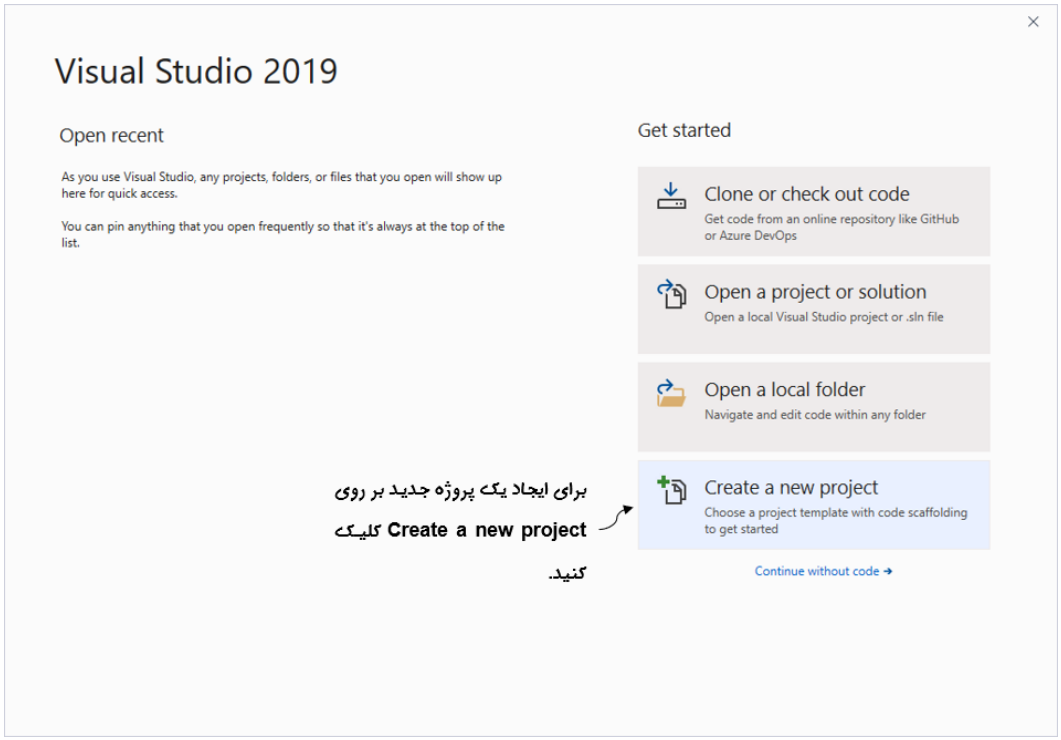


ایجاد اولین اپلیکیشن Web API

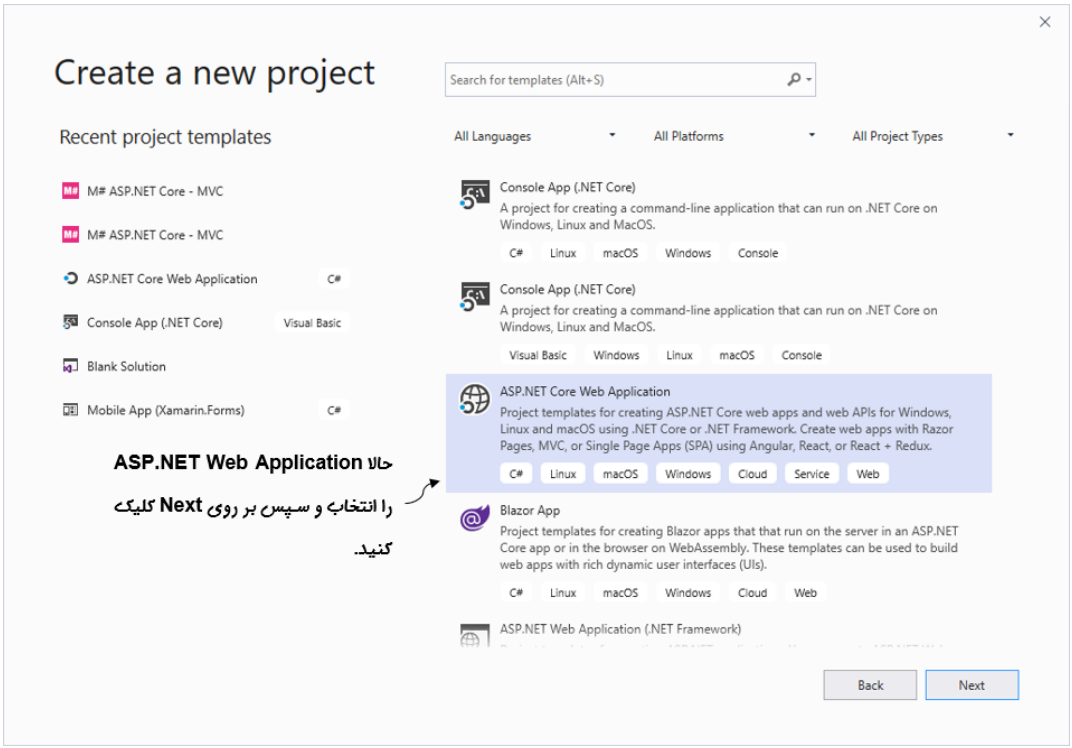
تا اینجا همه چیز خیلی عالی بود، بیایید با هم اولین اپلیکیشن Web API را ایجاد کنیم.

در Visual Studio 2019 برای ایجاد یک پروژه‌ی Web API، باید مراحل زیر را دنبال کنید:

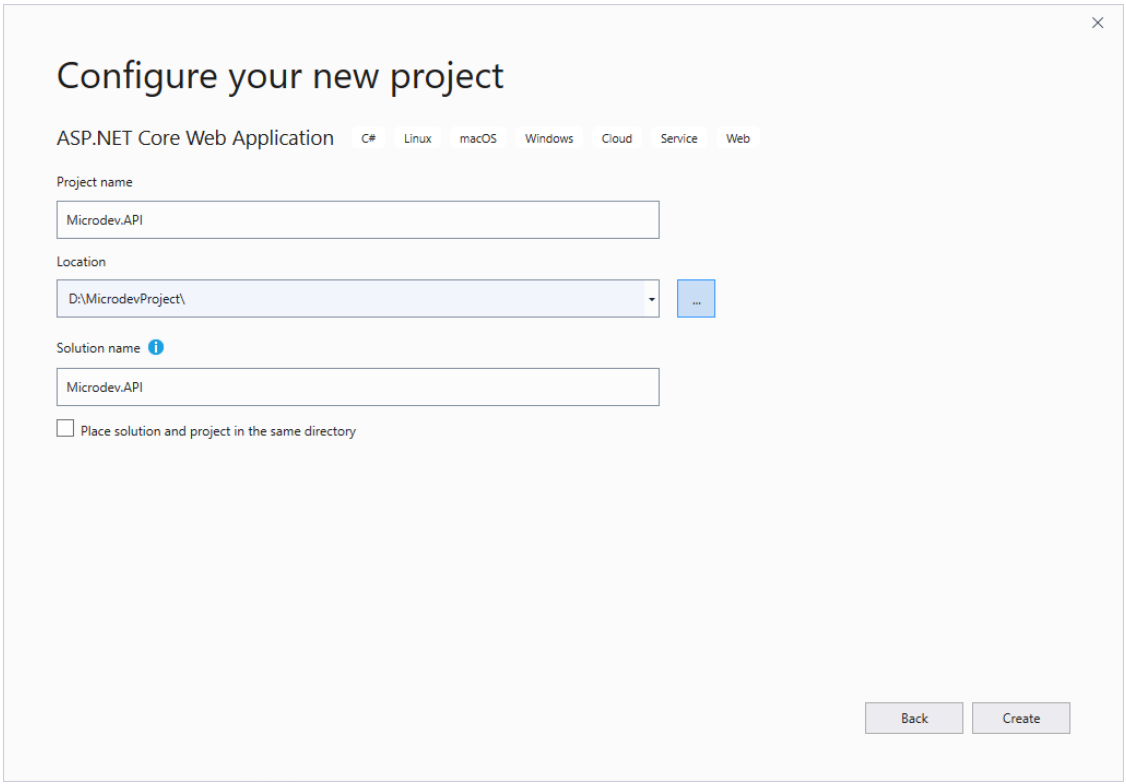
- **ویژوال استدیو ۲۰۱۹ را باز کنید و بر روی Create a new project کلیک کنید.**



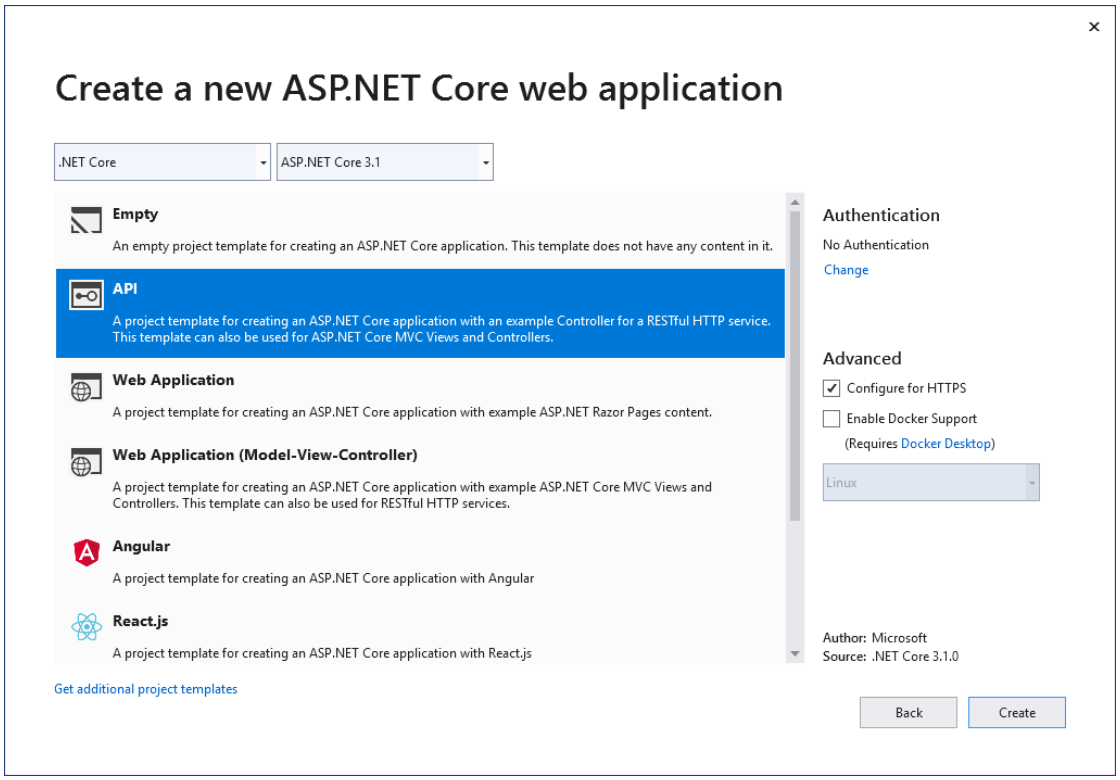
- در کادر بعدی **ASP.NET Core Web Application** را انتخاب و بر روی **Next** کلیک نمایید.



- حالا نام پروژه را **Microdev.API** بگذارید و سپس مکان ذخیره‌سازی و نام **Solution** را وارد و بر روی **Create** کلیک نمایید.

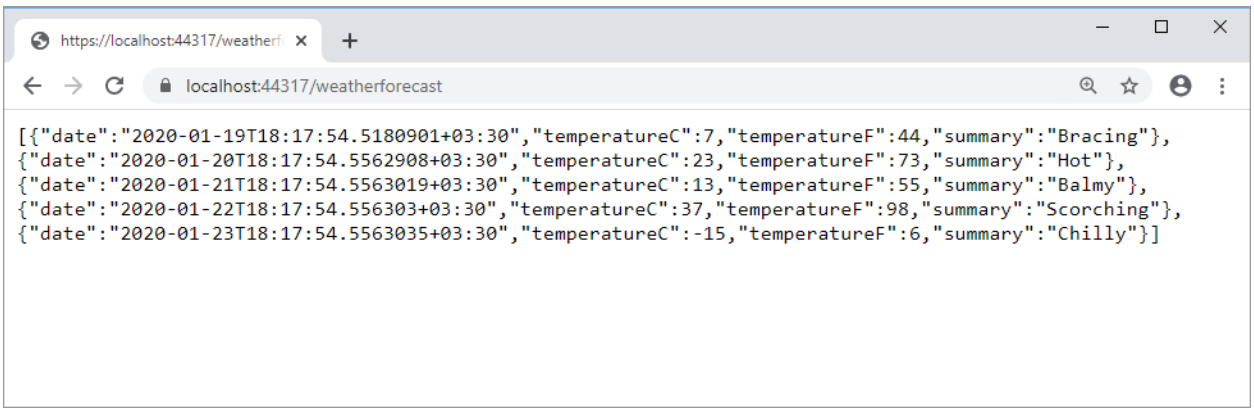


- در کادر بعدی **API** و **Asp Net Core 3.1** را انتخاب و بر روی **Create** کلیک کنید.



حالا در این پروژه جدید ASP.Net Core،یک کنترلر به نام WeatherForecastController در فولدر Controllers وجود دارد.

برای اجرای برنامه، F5 را فشار دهید.



همانطور که می‌بینید، ASP.NET Core به طور پیش‌فرض مدل API بازگشتی را در قالب یک JSON نمایش می‌دهد. شما در این صفحه بجای دیدن یک HTML UI، تنها داده‌های موردنیاز را می‌بینید.

در این مثال زمانیکه یک Request به آدرس /weatherforecast می‌فرستید Web API یک لیست String برمی‌گرداند.

JSON شبیه Anonymous Objectهای سی‌شارپ است، با این تفاوت که در JSON باید عملگر مساوی را با Colon جایگزین کنید و نام Propertyها را با علامت دابل کوتیشن بپیچید.

C# anonymous object:

```
var departments = new [] {
    new
    {
        Id=1,
        Name="Programming"
    },
    new
    {
        Id=2,
        Name="Fasico"
    },
    new
    {
        Id=3,
        Name="BFC"
    },
};
```

شیء JSON زیر آرایه‌ای از داده‌ها را نمایش می‌دهد:

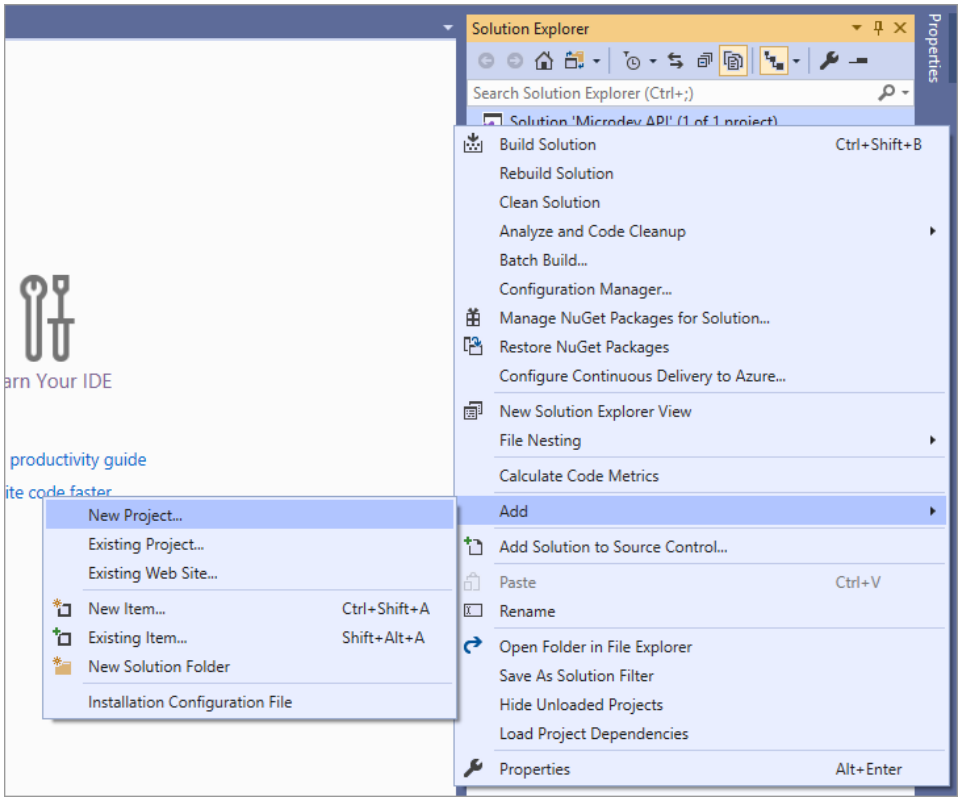
```
[
    {
        "id": 1,
        "name": "Programming"
    },
    {
        "id": 2,
        "name": "Fasico"
    },
    {
        "id": 3,
        "name": "BFC"
    }
]
```

افزودن Domain Modelها

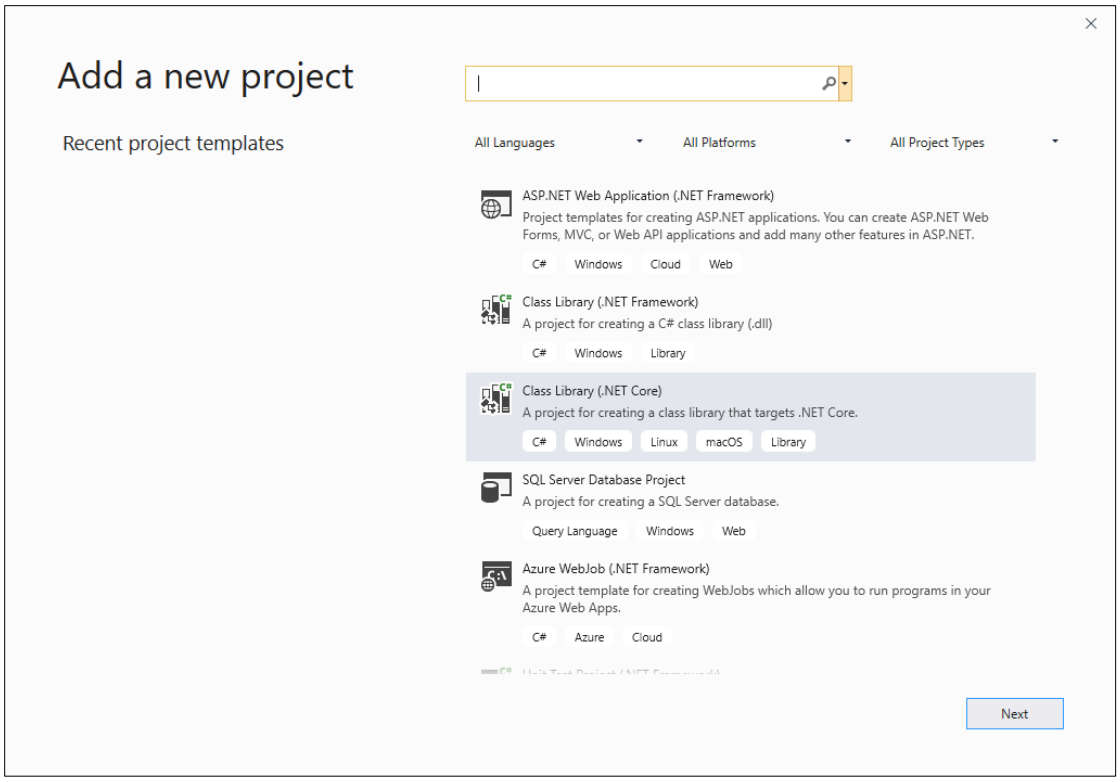
در این بخش قرار است Entity Classهایی ایجاد کنید که به جداول دیتابیس(MicrodevDB) مپ شوند و سپس EF Migrationها، از این Entitiyها، جدول‌هایی را آماده کنند.

برای اضافه کردن Domain Modelها به پروژه، مراحل زیر را دنبال نمایید:

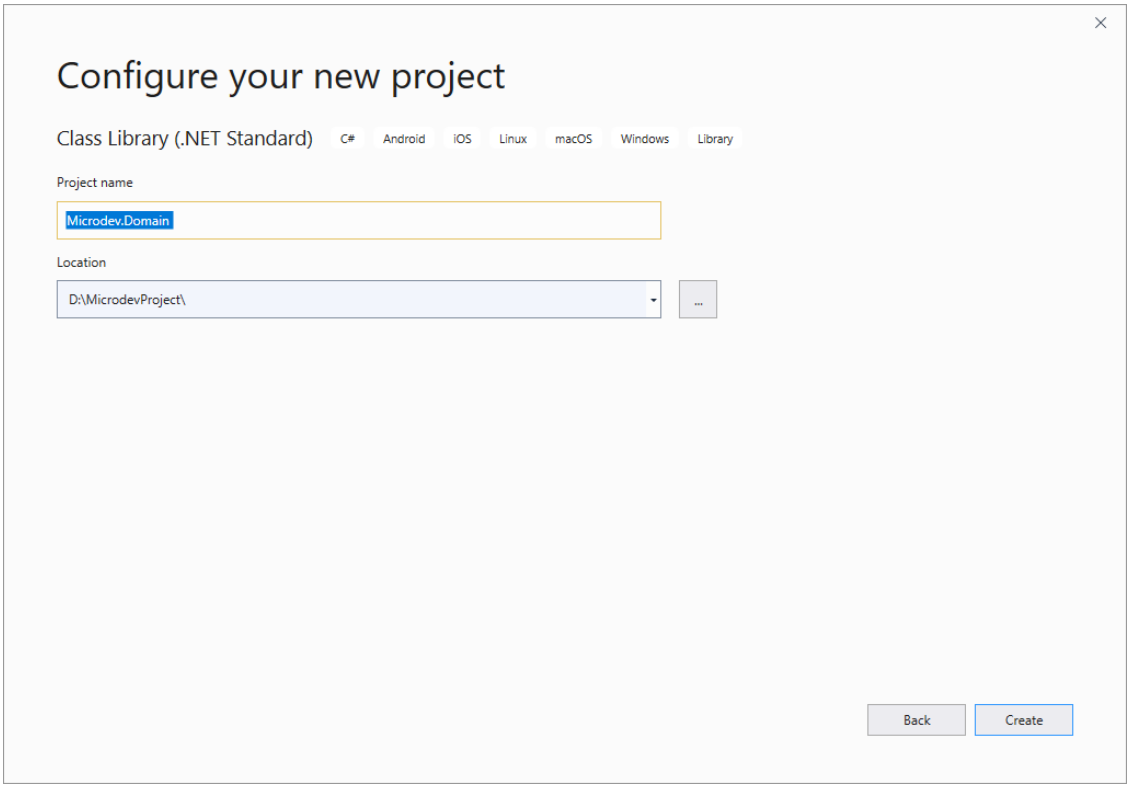
۱) **بر روی Solution راست کلیک کنید و سپس New Project < Add را انتخاب نمایید.**



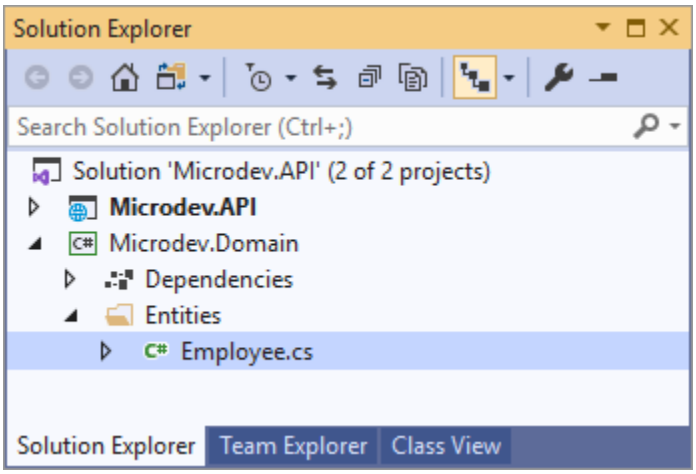
۲) **در کادر باز شده Class Library را انتخاب و سپس بر روی Next کلیک کنید.**



۳) نام پروژه را **Microdev.Domain** بگذارید و سپس بر روی **Create** کلیک نمایید.



۴) حالا بر روی پروژه Microdev.Domain راست کلیک کنید، Add > New Folder را انتخاب و نام فولدر را Entities بگذارید. در پایان هم در فولدر Entities یک کلاس به نام Employee اضافه نمایید.



حالا کد پایین را درون این کلاس اضافه کنید.

```
using System;

namespace Microdev.Domain.Entities
{
    public class Employee
    {
        public Employee(string firstName,string lastName, int? bossId, decimal
        salary)
        {
            FirstName = firstName ?? throw new
            ArgumentNullException(nameof(firstName));
             LastName = lastName ?? throw new
             ArgumentNullException(nameof(lastName));
             BossId = bossId;
             Salary = salary;
        }

        public Employee( string firstName, string lastName, int? bossId,
         decimal salary, int departmentId) : this(firstName, lastName, bossId,
         salary)
         {
             DepartmentId = departmentId;
         }

        public Employee(int id, string firstName, string lastName, int?
        bossId, decimal salary, int departmentId):this(firstName, lastName,
        bossId,salary,departmentId)
```

```
        {
            Id = id;
        }

        public int Id { get;private set; }
        public string FirstName { get;private set; }
        public string LastName { get;private set; }
        public int? BossId { get;private set; }
        public decimal Salary { get;private set; }
        public int DepartmentId { get;private set; }
    }


}
```

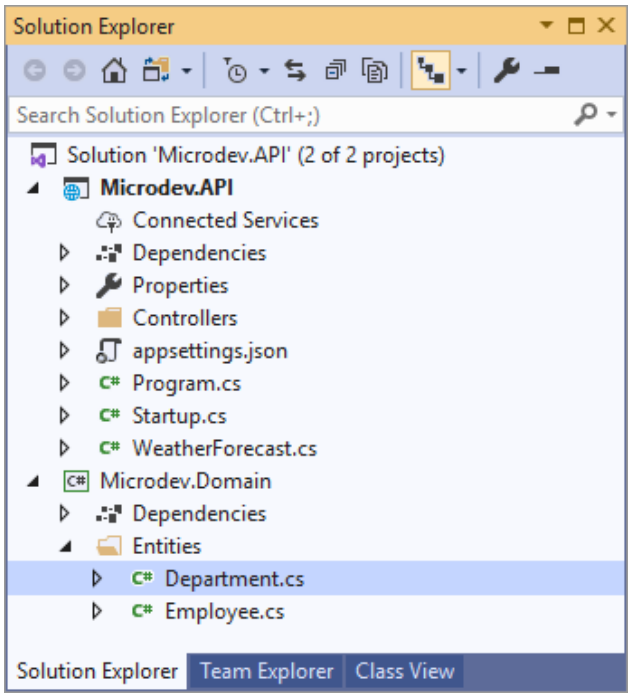
یک‌بار دیگر:

در فولدر **Entities** یک کلاس با نام **Department** ایجاد و سپس کدهای پایین را به آن اضافه نمایید.

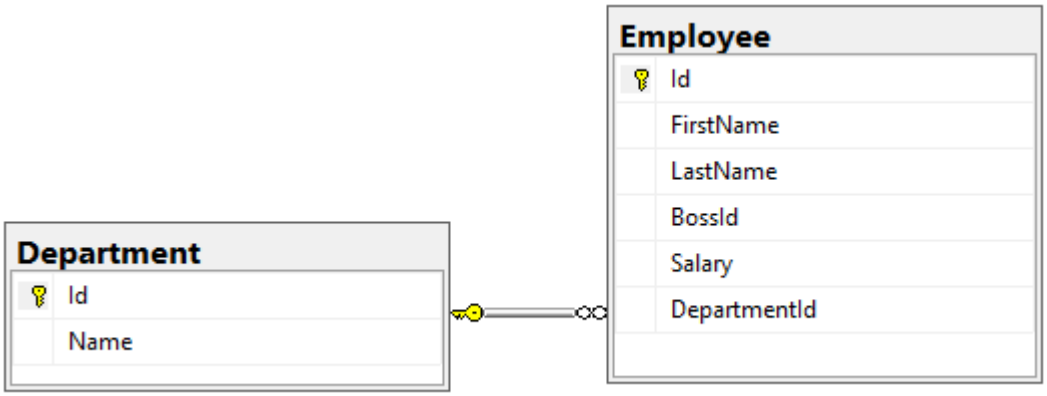
```
using System.Collections.Generic;

namespace Microdev.Domain.Entities
{
    public class Department
    {
        public Department(string name)
        {
            _employees = new List<Employee>();
            Name = name;
        }
        public Department(int id, string name):this(name)
        {
            Id = id;
        }

        public int Id { get;private set; }
        public string Name { get; private set; }
        private readonly List<Employee> _employees;
        public IReadOnlyCollection<Employee> Employees => _employees;
        public void UpdateName(string name)
        {
            //Domain rule
            Name = name;
            //Raise domain event
        }

        public void AddEmployee(string firstName,string lastName, int?
       bossId, decimal salary)
        {
            // Domain rules for adding the Employee to the Department
```

```
            var employee = new Employee(firstName,lastName, bossId, salary);

            _employees.Add(employee);
        }
    }

}
```

ساختار **Solution** :



در نهایت کلاس‌های **Department** و **Employee** ساختار پایین را در دیتابیس تولید خواهند کرد:

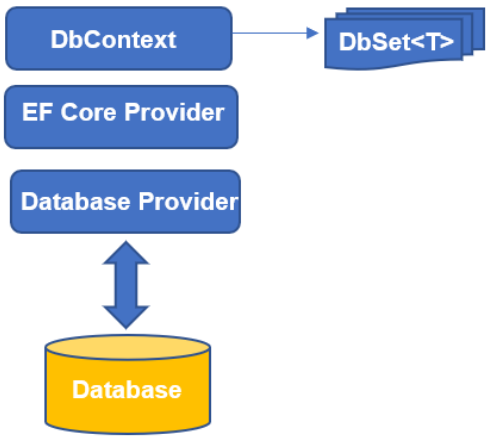


در کدهای بالا:

- **برای ذخیره اطلاعات دپارتمان در دیتابیس، یک کلاس Department تعریف شده است که طبق قرارداد، از پراپرتی Id به عنوان کلید اصلی استفاده می‌شود.**
- **یک Navigation Property به نام Employees (به منظور لود کارمندان مرتبط با هر دپارتمان) در کلاس Department تعریف شده است. با بودن این Navigation Property دیگر نیاز به کوئری مستقیم در کلاس Employees نیست.**
- **کلاس Employee هم شامل اطلاعات مربوط به کارمندان است.**

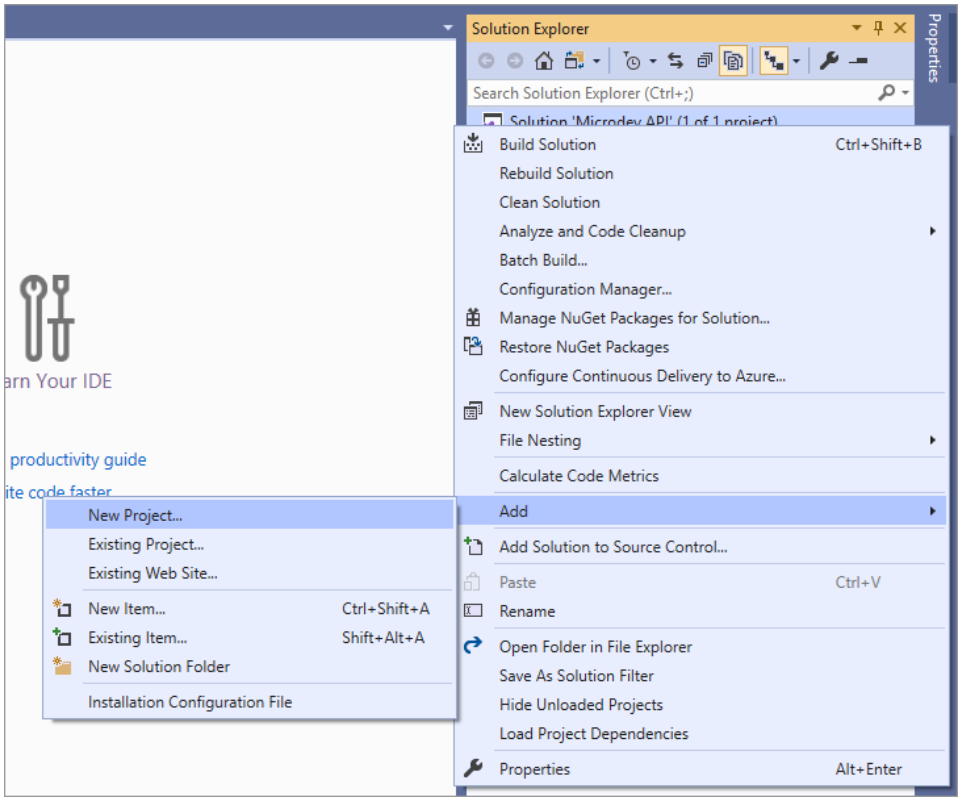
Entity Framework Core و Dapper

Entity Framework Core یک ORM است که به شما امکان می‌دهد تا با دیتابیس ارتباط برقرار کنید. Entity Framework Core کلاس‌های شما را به جداول، Propertyهای کلاس‌ها را به ستون‌های جداول و Instanceهای کلاس‌ها را به ردیف‌های جداول Map می‌کند.

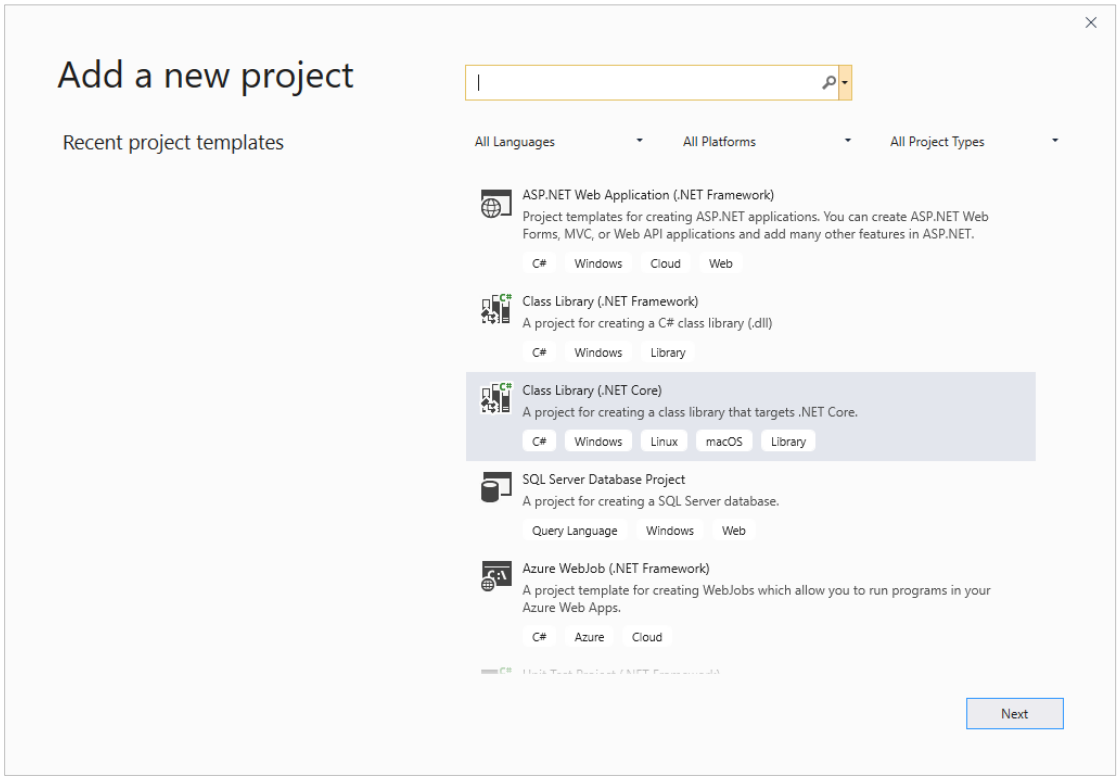


بیایید با هم یک پروژه به نام Microdev.Infrastructure به Solution اضافه کنیم:

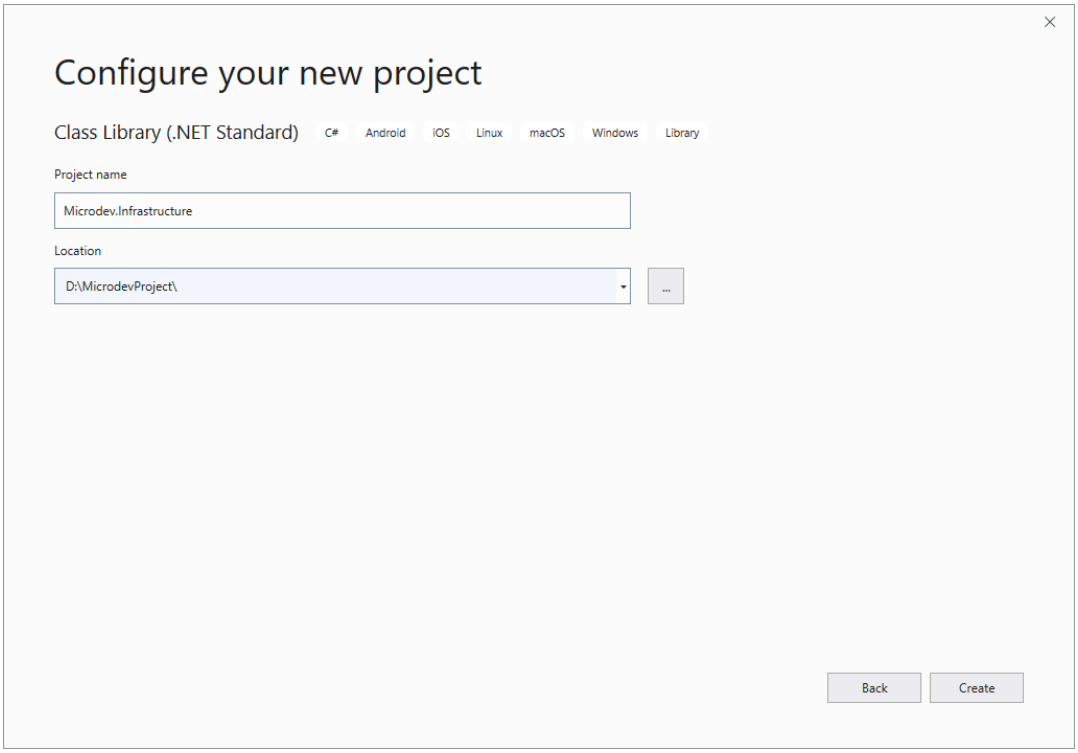
۱) **بر روی Solution راست کلیک کنید و سپس Add > New Project را انتخاب نمایید.**



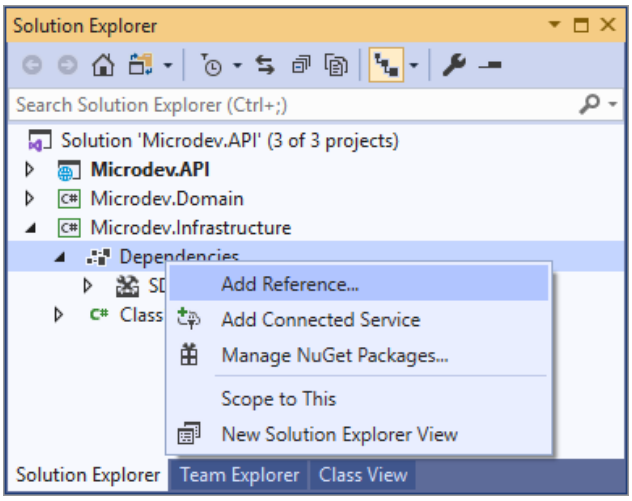
۲) در کادر باز شده **Class Library** را انتخاب و سپس **Next** را کلیک کنید.



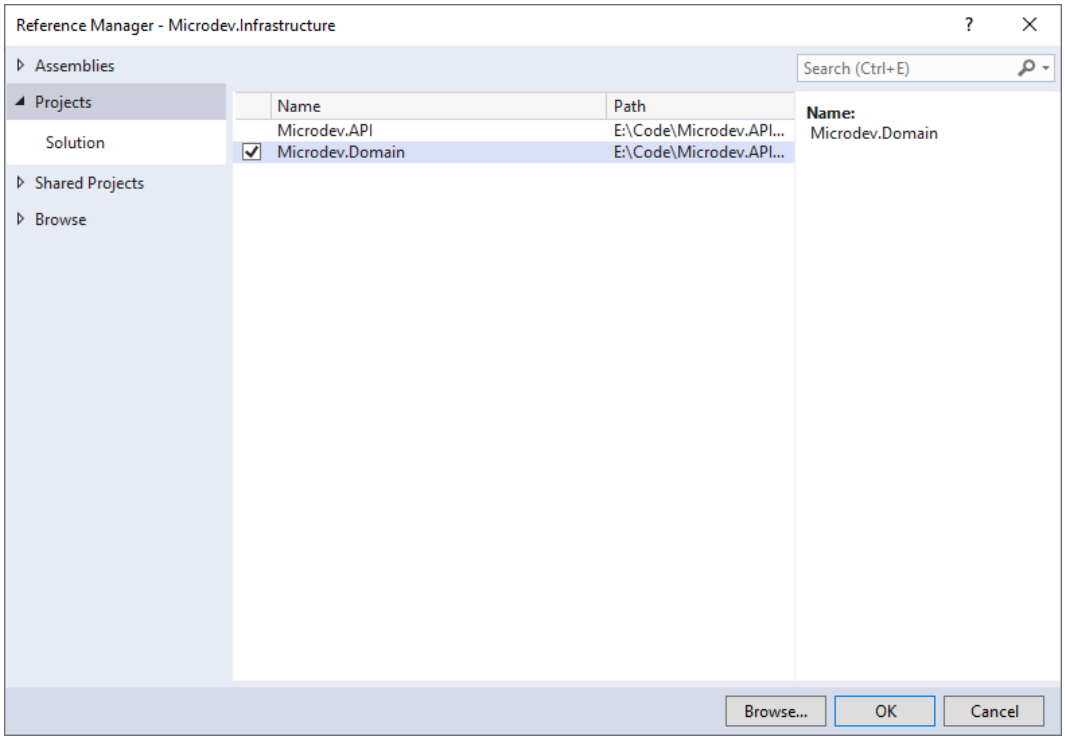
۳) نام پروژه را **Microdev.Infrastructure** بگذارید و بر روی **Create** کلیک کنید.



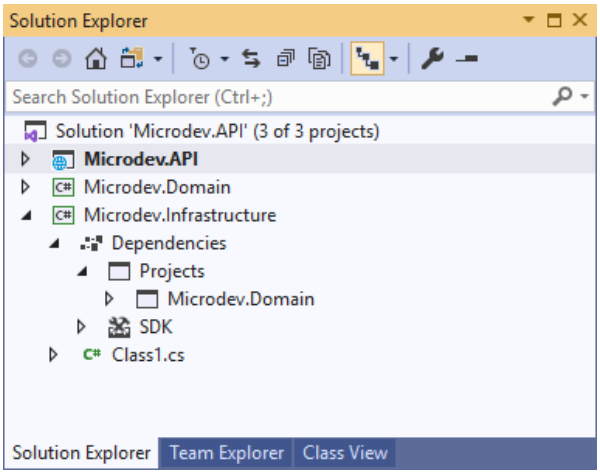
۴) در مرحله بعدی یکبار **Solution** را **Build** نمایید و سپس در پروژه **Microdev.Infrastructure** بر روی **Dependencies** راست کلیک و **Add Reference** را انتخاب کنید.



۵) حالا در کادر **Reference Manager**، پروژه **Microdev.Domain** را انتخاب و **OK** کنید.



ساختار Solution:

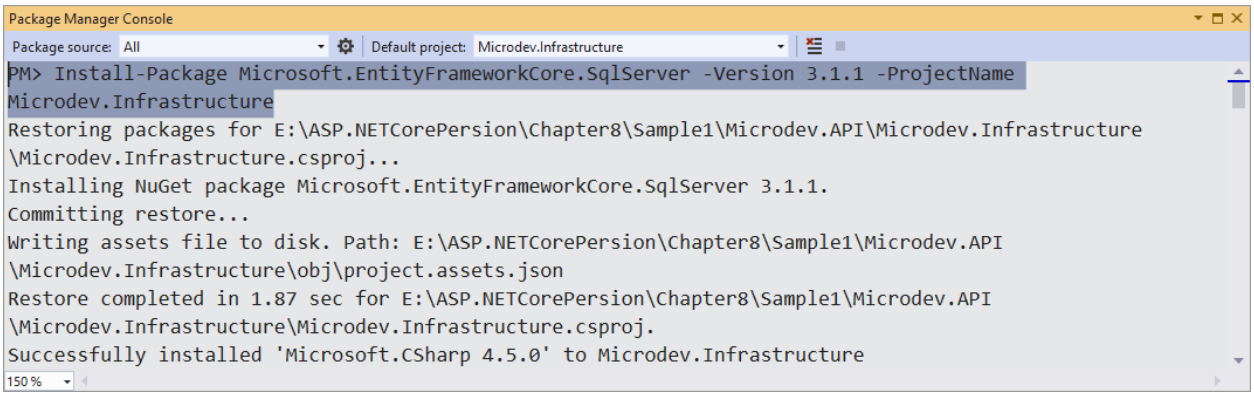


برای استفاده از EF Core یک NuGet package وجود دارد که قبل از هر کاری، باید برای دسترسی به دیتابیس موردنظرمان نصب شود. در اینجا ما می‌خواهیم به پایگاه داده SQL Server دسترسی داشته باشیم، بنابراین، باید بسته NuGet Microsoft.EntityFrameworkCore.SqlServer را نصب کنیم.

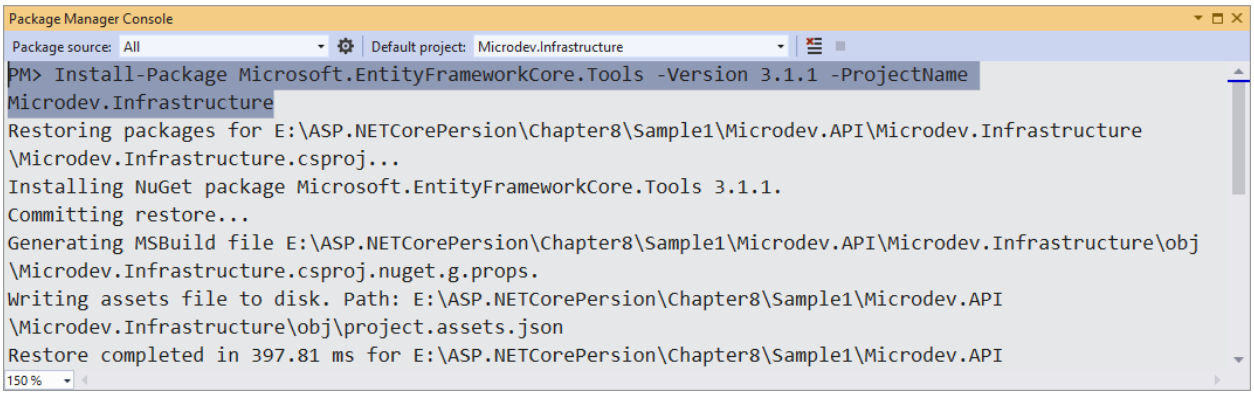
برای نصب این NuGet package:

Package Manager را از مسیر Tools> NuGet Package Manager> Console Manager باز کنید و سپس روبروی دستور PM> دستورات پایین را به ترتیب اجرا نمایید:

Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 3.1.1 -ProjectName Microdev.Infrastructure



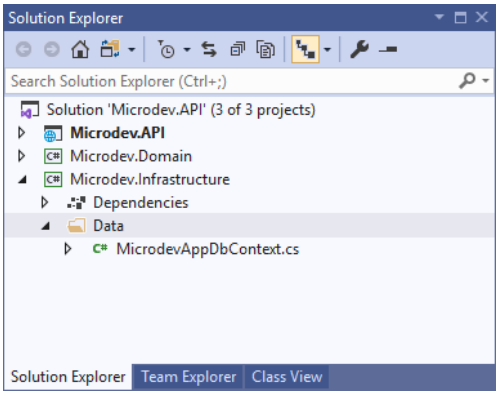
Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.1.1 -ProjectName Microdev.Infrastructure



اپلیکیشن برای فراخوانی دیتابیس از DbContext استفاده می‌کند. بنابراین برای ایجاد یک DbContext باید یک کلاس داشته باشیم که از کلاس پایه DbContext ارثبری کند. سپس در این کلاس یک پراپرتی از نوع DbSet<Department> و یک پراپرتی از نوع DbSet<Employee> قرا دهیم تا EF Core بتواند کلاس Department را بیاید و عملیات Map کردن را انجام دهد.

مراحل افزودن DBContext به پروژه:

- **بر روی پروژه Microdev.Infrastructure راست کلیک کنید و از مسیر Add > New Folder یک فولدر با نام Data ایجاد نمایید.**
- **درون این فولدر یک کلاس با نام MicrodevAppDbContext ایجاد نمایید.**



در کد پایین کلاس MicrodevAppDbContext نوشته شده که می‌توانید آن را درون اپلیکیشن خود کپی نمایید.

```
using Microdev.Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace Microdev.Infrastructure.Data
{
    public class MicrodevAppDbContext : DbContext
    {
        public MicrodevAppDbContext(DbContextOptions options) : base(options)
        {
        }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Employee> Employees { get; set; }
    }

}
```

نکته!!

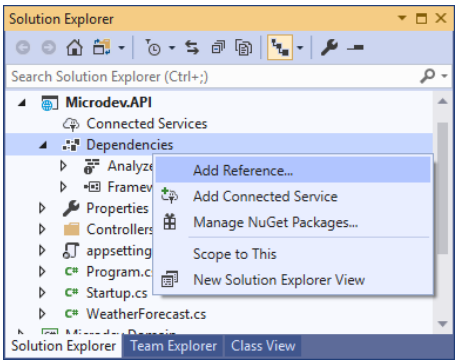
نام جداول با نام پراپرتی‌های DbSet<T> موجود در کلاس MicrodevAppDbContext هم‌نام هستند.

رجیستر DbContext

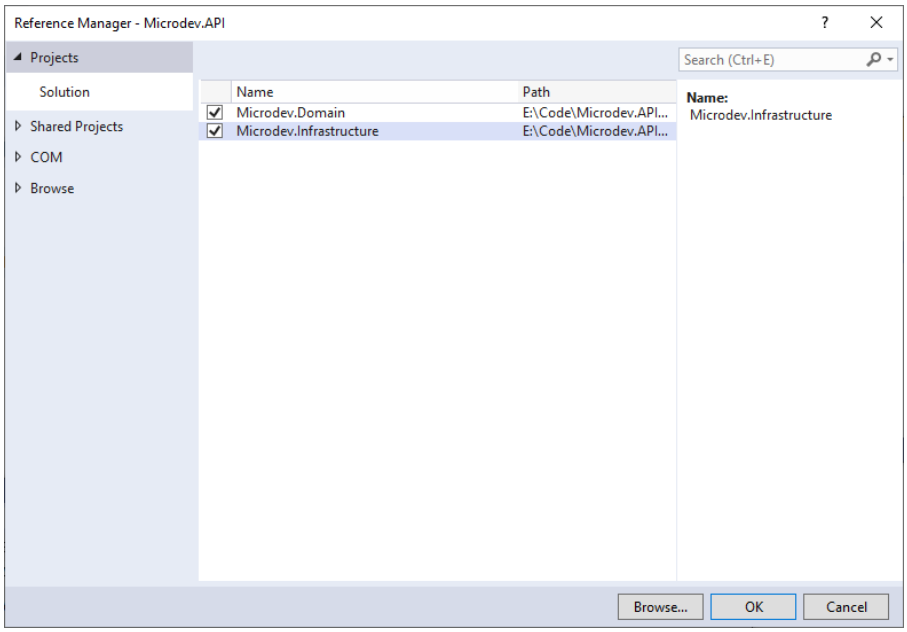
MicrodevAppDbContext هم مانند سایر سرویس‌های موجود ASP.Net Core، باید در DI Container رجیستر نمایید.

قبل از رجیستر DbContext باید:

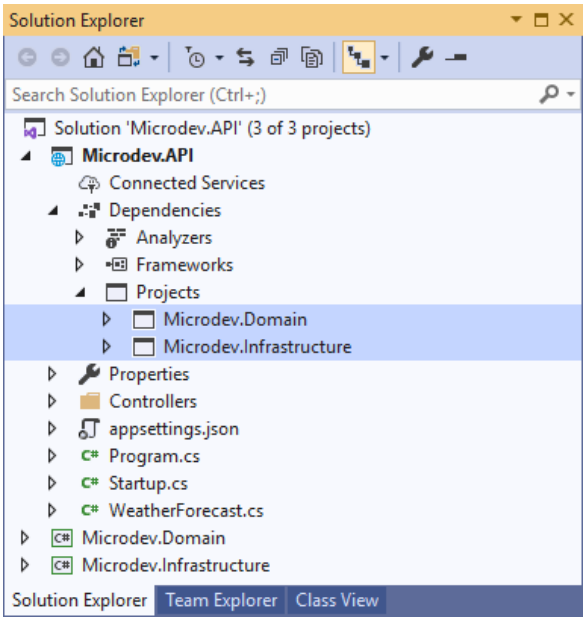
- در پروژه **Microdev.API**، بر روی **Dependencies** راســت کلیک نمایید و ســپس **Add Reference** را انتخاب نمایید.



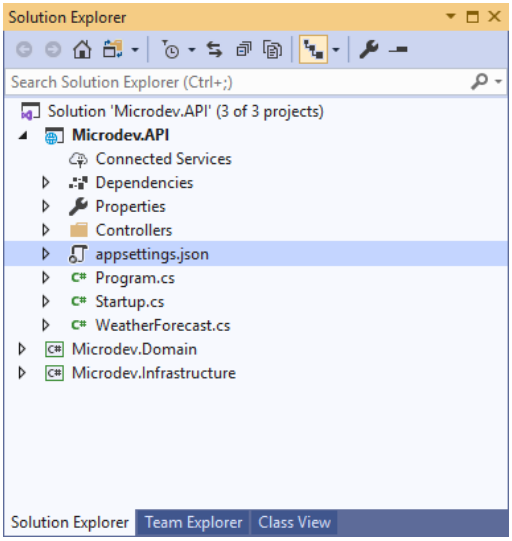
- حالا **Reference Manager** باز شــده، بنابرایم در آن پروژه‌های **Microdev.Domain** و **Microdev.Infrastructure** را انتخاب و بر روی **OK** کلیک نمایید.



ساختار پروژه:

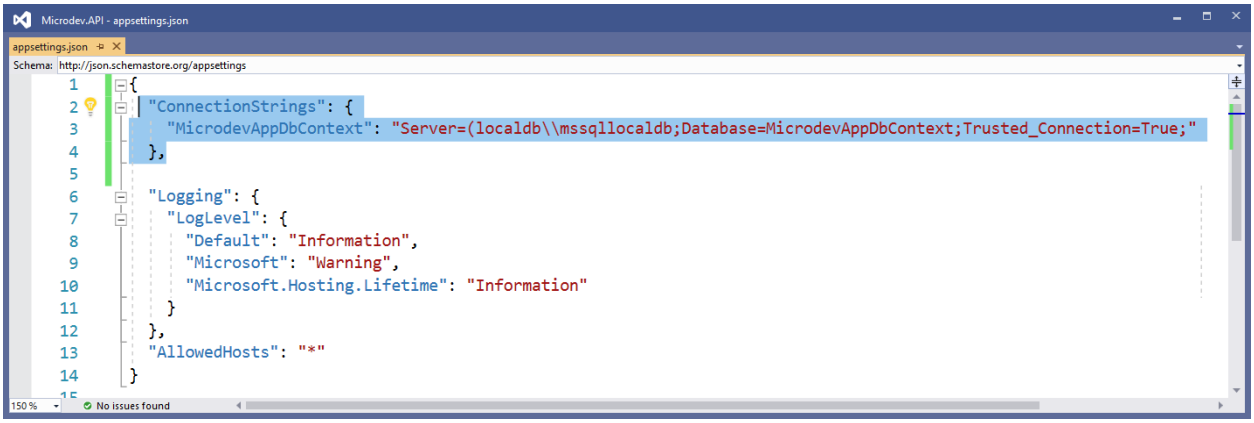


مرحله بعد تغییر **Connection String** در **appSetting.json** است:




```
"ConnectionStrings": {
    "MicrodevAppDbContext":
    "Server=(localdb)\\mssqllocaldb;Database=MicrodevAppDbContext;Trusted_Co
    nnection=True;"
  },
```

نتیجه نهایی در **appsetting.json** :



در پایان باید عبارت using پایین در Startup.cs اضافه کنید:

```
using Microdev.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
```

EF Core برای رجیستر کردن DbContext یک اکستنشن متد جنریک به نام AddDbContext<T> آماده کرده که شما می‌توانید همانند کد پایین آن را به متد ConfigureServices کلاس Startup پروژه‌ی Microdev.API اضافه کنید:

```
public void ConfigureServices(IServiceCollection services)
{
    var applicationConnectionString =
    Configuration.GetConnectionString("MicrodevAppDbContext");

   services.AddDbContext<MicrodevAppDbContext>(options =>
   options.UseSqlServer(applicationConnectionString));
  services.AddControllers();
}
```

connection string از بخش ConnectionStrings درون Configuration گرفته می‌شود.

DbContext اپلیکیشن با استفاده از پارامتر جنریک رجیستر می‌شود.

database provider مشخص کننده است.

Data Seeding چیست؟

در بسیاری از مواقع با اجرای اپلیکیشن، نیاز است تا برخی از جداول دیتابیس با اطلاعات پیش‌فرضی مقداردهی اولیه شوند. راه حل این مسأله، Seed دیتابیس است.

Seed دیتابیس، این امکان را به ما می‌دهد تا در حین اجرای اپلیکیشن، اطلاعاتی را به جداولی از دیتابیس اضافه نماییم.

برای اعمال قابلیت Seed به اپلیکیشن:

کلاس DbContext متدی به نام OnModelCreating دارد که Instanceای از ModelBuilder را به عنوان پارامتر در نظر می‌گیرد. وقتی فریم‌ورک برای ایجاد مدل و Mappingهای آن در حافظه ایجاد می‌شود، این متد توسط فریم‌ورک فراخوانی خواهد شد. مثال زیر Seed Data را برای Department و Employee در OnModelCreating پیکربندی می‌کند:

```
using Microsoft.EntityFrameworkCore;
using Microdev.Domain.Entities;
namespace Microdev.Infrastructure.Data
{
    public class MicrodevAppDbContext: DbContext
    {
        public MicrodevAppDbContext(DbContextOptions options) : base(options)
        {
        }
        public DbSet<Department> Departments { get; set; }

        protected override void OnModelCreating(ModelBuilder
        modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<Department>().ToTable("Department");

            modelBuilder.Entity<Department>().HasData(
                new Department(id: 1, name: "Programming"),
                new Department(id: 2, name: "Fasico"),
                new Department(id: 3, name: "IT"),
                new Department(id: 4, name: "BFC")
                );
            modelBuilder.Entity<Employee>()
                .Property(p => p.Salary).HasColumnType("Decimal(10,2)");


            modelBuilder.Entity<Employee>().HasData(
            new Employee(id: 1, firstName: "Zahra", lastName: "Bayat",
            bossId: 2, salary: 2000, departmentId: 1),
            new Employee(id: 2, firstName: "Ali", lastName: "Bayat",
            bossId: 1, salary: 3000, departmentId: 1),
            new Employee(id: 3, firstName: "Sara", lastName: "Masoodi",
            bossId: 1, salary: 3000, departmentId: 1),
            new Employee(id: 4, firstName: "Mehdi", lastName:
            "Mohamadi", bossId: 1, salary: 3000, departmentId: 1),
            new Employee(id: 5, firstName: "Amin", lastName: "Sadeghi",
            bossId: 1, salary: 1000, departmentId: 1),
```

پیکربندی Seed Data برای Department

پیکربندی Seed Data برای Employee

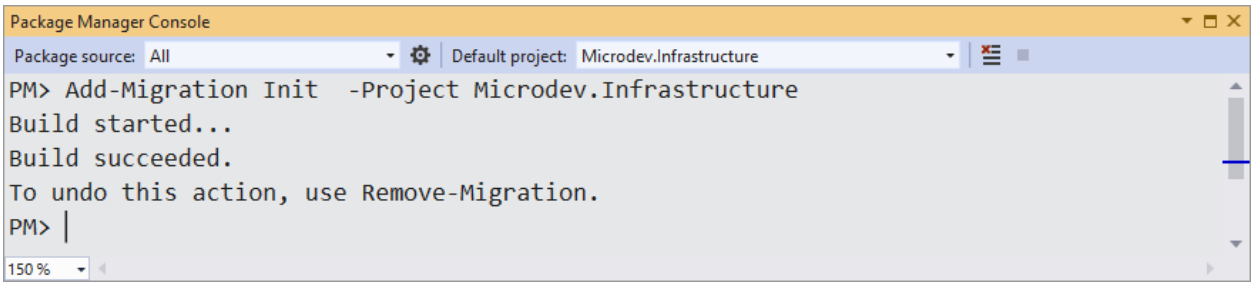
```
                new Employee(id: 6, firstName: "Shadi", lastName: "Sohbati",
                bossId: 2, salary: 1000, departmentId: 1),
                new   Employee(id:   7,   firstName:   "Somaye",   lastName:
                "Mahdavi", bossId: 2, salary: 1000, departmentId: 2),
                new Employee(id: 8, firstName: "Maryam", lastName: "Zahedi",
                bossId: 2, salary: 1000, departmentId: 2),
                new Employee(id: 9, firstName: "Mary", lastName: "Zibayee",
                bossId: 2, salary: 1000, departmentId: 2)
                    );
        }
    }

}
```

ایجاد و آپدیت دیتابیس با Migration

Migration راهی آسان برای ایجاد و به روزرسانی دیتابیس است که می‌تواند همزمان با حفظ داده‌های موجود در دیتابیس، جداول را با مدل‌های اپلیکیشن همگام کند.

برای ایجاد Migration، دستور زیر را در Console Package Manager اجرا کنید:

```
Add-Migration Init -Project Microdev.Infrastructure
```

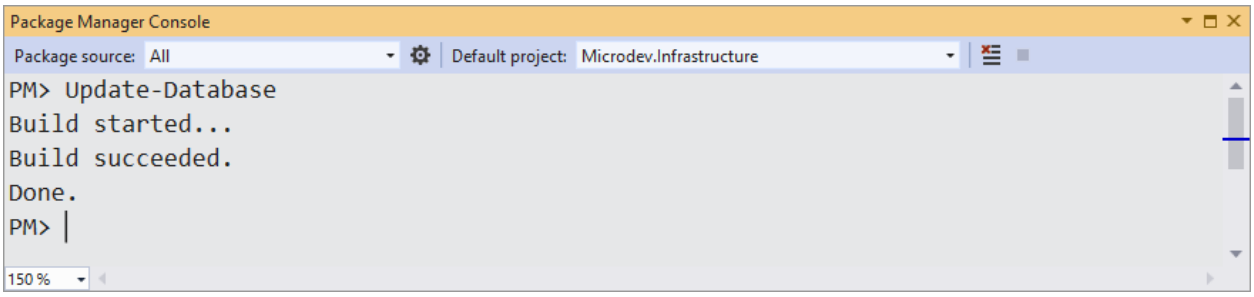


در دستور بالا:

- **Init، نام این Migration است.**
- **و گزینه project- هم مکان ایجاد Migration را مشخص می‌کند.**

این Migration هنوز در دیتابیس شما اعمال نشده است بنابراین اجرای دستور Update-Database برنامه شما را کامپایل می‌کند و با استفاده از connection String موجود در appsettings.json، به دیتابیس متصل می‌شود.

```
Update-Database
```

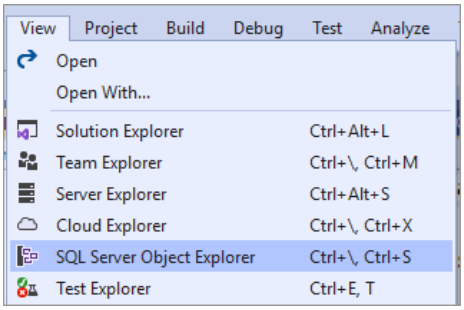


پس از اجرای دستور Update-Database، در صورتی که دیتابیس وجود نداشته باشد، دیتابیس جدید ایجاد می‌شود و اگر دیتابیسی وجود داشته باشد، از اسکریپت Migration برای بروزرسانی دیتابیس استفاده خواهد شد.

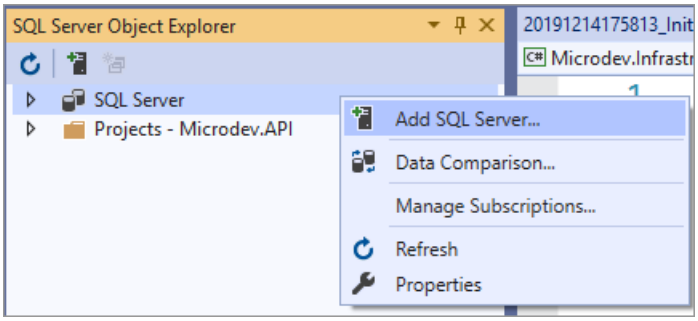
نکته!!

در کادر Package Console Manager قبل از اجرای دستور آپدیت دیتابیس، حتما نام پروژه را بر روی Microdev.Infrastructure بگذارید.

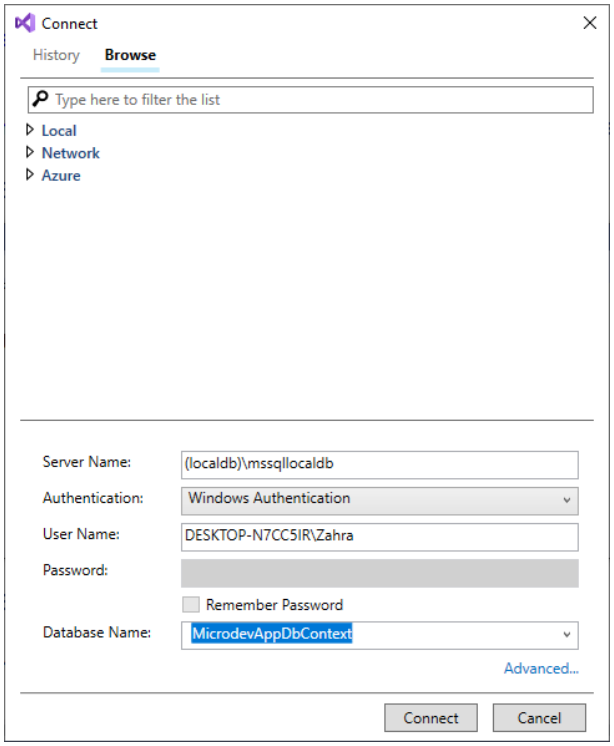
برای مشاهده دیتابیس در Visual Studio باید از منوی View گزینه‌ی SQL Server Object Explorer را انتخاب کنید.



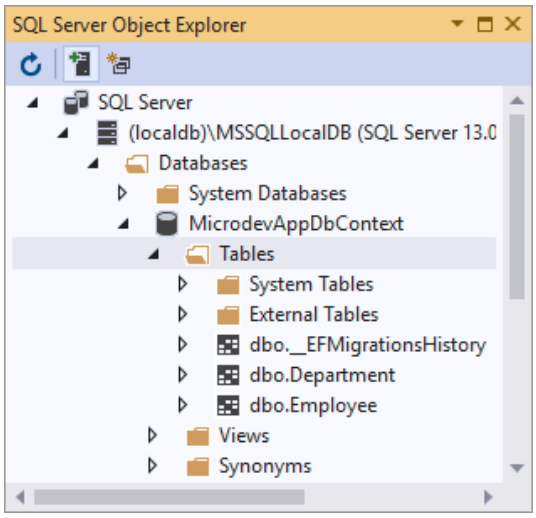
حالا بر روی SQL Server راست کلیک نمایید و سپس گزینه‌ی Add SQL Server... را انتخاب کنید.



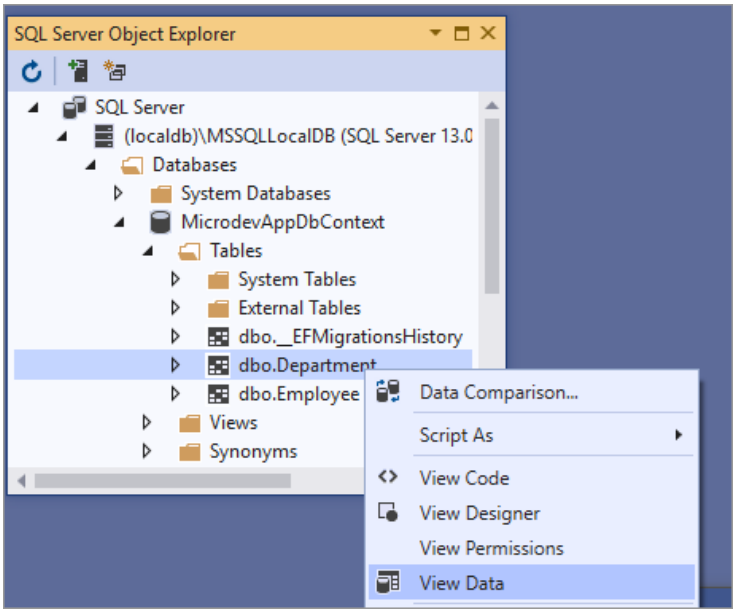
در کادر ارتباط با سـرور، در قسـمت Server Name عبارت (localdb)\mssqllocaldb را تایپ کنید. گزینه Authentication را بر روی Windows Authentication قرار دهیـد. و در پـایـان Database Name را مشخص و بر روی Connect کلیک نمایید.

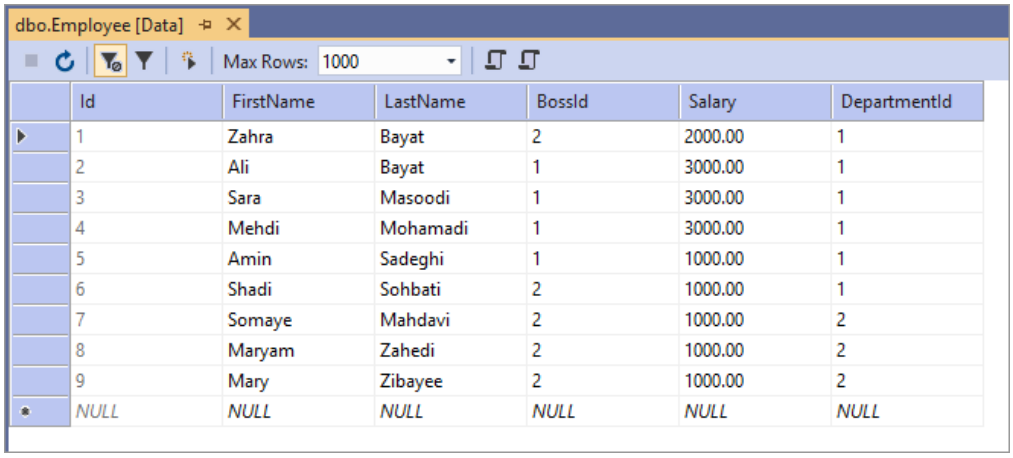


حالا ویژوال اسـتودیو به LocalDB متصـل می‌شـود و دیتابیس‌های موجود خود را در SQL Server Object Explorer نشان می‌دهد. می‌توانید نودها را باز کنید تا جداول ایجاد شده توسط EF Core را ببینید.



برای مشاهده داده‌ها، روی یک جدول راست کلیک نمایید و View Data را انتخاب کنید.



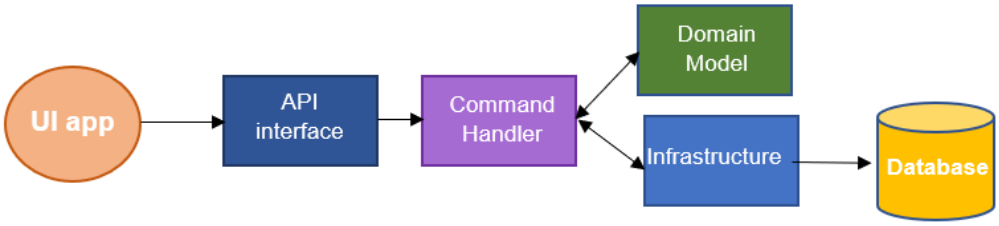


Id	FirstName	LastName	BossId	Salary	DepartmentId
1	Zahra	Bayat	2	2000.00	1
2	Ali	Bayat	1	3000.00	1
3	Sara	Masoodi	1	3000.00	1
4	Mehdi	Mohamadi	1	3000.00	1
5	Amin	Sadeghi	1	1000.00	1
6	Shadi	Sohbati	2	1000.00	1
7	Somaye	Mahdavi	2	1000.00	2
8	Maryam	Zahedi	2	1000.00	2
9	Mary	Zibayee	2	1000.00	2
NULL	NULL	NULL	NULL	NULL	NULL

ما اصـــول اولیه Entity Framework Core را آموختیم، بیایید از این اصـــول برای ارائه Featureهای برنامه Microdev استفاده نماییم.

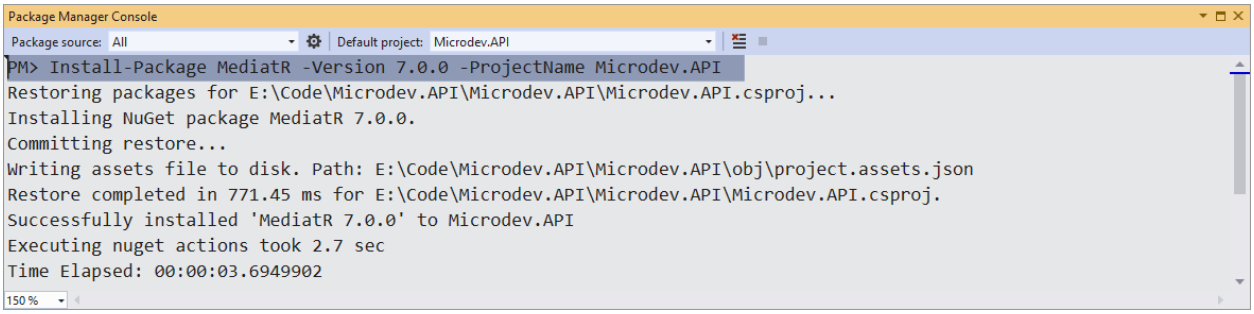
پیاده‌سازی Command

ما می‌خواهیم از کتابخانه MediatR اسـتفاده کنیم تا مدل Read را از مدل Write جدا کنیم. این موضـوع مهم می‌تواند Performance, Scalability و Security را در اپلیکیشـن‌های واقعی به حداکثر برسـاند. MediatR اینترفیسی است که ارتباط بین بخش‌های مختلف اپلیکیشن ما را برقرار می‌کند.



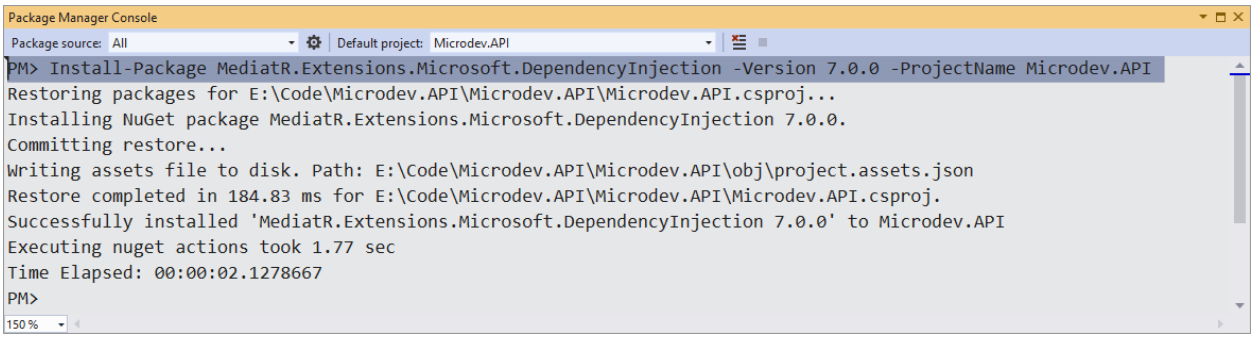
برای نصب MediatR دستور پایین را در Package Manager Console اجرا کنید:

Install-Package MediatR -Version 7.0.0 -ProjectName Microdev.API



MediatR برای مدیریت و لینک بین پیام‌ها به یک IoC Container نیاز دارد. شما برای این Container باید Package پایین را نصب کنید:

Install-Package MediatR.Extensions.Microsoft.DependencyInjection -Version 7.0.0 -ProjectName Microdev.API



در مرحله بعد، برای ارسال اطلاعات نیاز به یک دستور دارید، بنابراین:

- **در ریشه پروژه Microdev.API یک فولدر جدید به نام Application ایجاد کنید.**
- **سپس درون این فولدر یک فولدر دیگر با نام Commands ایجاد نمایید.**
- **در پایان باید کلاسی ایجاد کنید(`CreateDepartmentCommand`) که حتما اینترفیس MediatR → IRequest را پیاده‌سازی کند.(همانند کد پایین)**

```
using MediatR;

namespace Microdev.API.Application.Commands
{
    public class CreateDepartmentCommand : IRequest<bool>
    {
        public CreateDepartmentCommand()
        {

        }

        public CreateDepartmentCommand(string name)
        {
            Name = name;
        }

        public string Name { get; set;}
    }

}
```

در مرحله بعد، ما نیازمند Handlerهایی هستیم، بنابراین در فولدر Commands یک کلاس با نام DepartmentCommandHandler (که باید از اینترفیس IRequestHandler ارثبری کند) ایجاد کنید.

```
using MediatR;
using Microdev.Domain.Entities;
using Microdev.Infrastructure.Data;
using System.Threading;
using System.Threading.Tasks;

namespace Microdev.API.Application.Commands
{
    public class CreateDepartmentCommandHandler :
     IRequestHandler<CreateDepartmentCommand, bool>
    {
        private readonly MicrodevAppDbContext _context;

        public CreateDepartmentCommandHandler(MicrodevAppDbContext
        microdevAppDbContext )
        {
            _context = microdevAppDbContext;
        }
        public async Task<bool> Handle(CreateDepartmentCommand request,
        CancellationToken cancellationToken)
        {
            var department = new Department(request.Name);

            await _context.AddAsync(department);
            await _context.SaveChangesAsync();
```

```
            return true;
        }
    }
}
```

در کد بالا:

- **Instanceی از MicrodevAppDbContext را در Constructor کلاس CreatDepcaseCommandHandler تزریق کردیم.**
- **سپس با متد Handle به صورت Asynchronously یک Transaction دیتابیس ایجاد می‌نماییم.**
- **و در پایان با فراخوانی SaveChangesAsync شیء جدیدی از Department را در دیتابیس ذخیره می‌کنیم.**

> **نکته!!**
>
> **تمام عملیات موجود در Transaction انجام می‌شود، بنابراین عملیات یا به صورت موفق انجام می‌شود یا تمام عملیات انجام نمی‌شود.**

کلاس CreateDepartmentCommand از IRequest<bool> ارثبری کرد، که نوع bool، خروجی Command را نشان می‌داد.

هر نوع Command دارای Handler مخصوص به خود است، بنابراین ما کلاسی با نام CreateDepartmentCommandHandler ایجاد کردیم که باید از IRequestHandler<T,U> ارثبری کند و Task<U> را برگرداند.

این CommandHandler نیاز به دسترسی به MicrodevAppDbContext دارد که باید به Constructor پاس داده شود. این وابستگی‌ها توسط Dependency Injection داخلی Resolve می‌شود.

پس از همه این کارها، باید MediatR را در کلاس Startup پیکربندی کنید بنابراین باید کد زیر را در متد ConfigureService اضافه نمایید.

```
services.AddMediatR(typeof(CreateDepartmentCommandHandler));
```

کلاس Startup:

```
using MediatR;
```

```
using Microdev.API.Application.Commands;
using Microdev.Infrastructure.Data;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
          var applicationConnectionString =
          Configuration.GetConnectionString("MicrodevAppDbContext");

            services.AddDbContext<MicrodevAppDbContext>(options =>
            options.UseSqlServer(applicationConnectionString));
            services.AddMediatR(typeof(CreateDepartmentCommandHandler));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
        env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
```

سرویس MediatR

```
    }
}
```

افزودن UpdateDepartmentCommand

UpdateDepartmentCommand شبیه CreateDepartmentCommand است با این تفاوت که درون این Command یک پراپرتی Id هم وجود دارد.

حالا کلاس UpdateDepartmentCommand را به فولدر Commands اضافه کنید:

```
using MediatR;

namespace Microdev.API.Application.Commands
{
    public class UpdateDepartmentCommand : IRequest<bool>
    {

        public UpdateDepartmentCommand()
        {

        }

        public UpdateDepartmentCommand(int id, string name)
        {
            Id = id;
            Name = name;
        }

        public int Id { get; set;}
        public string Name { get; set;}
    }

}
```

و حالا در این فولدر کلاس UpdateDepartmentCommandHandler را هم اضافه نمایید:

```
using MediatR;
using Microdev.Infrastructure.Data;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Microdev.API.Application.Commands
{
    public class UpdateDepartmentCommandHandler :
    IRequestHandler<UpdateDepartmentCommand, bool>
    {
```

```
        private readonly MicrodevAppDbContext _context;

         public UpdateDepartmentCommandHandler(MicrodevAppDbContext
         applicationDbContext)
        {
            _context = applicationDbContext;
        }
         public async Task<bool> Handle(UpdateDepartmentCommand request,
         CancellationToken cancellationToken)
        {
            var department = _context.Departments.Find(request.Id);
               department.UpdateName ( request.Name);

            await _context.SaveChangesAsync();

            return true;
        }
    }

}
```

افزودن DeleteDepartmentCommand

برای انجام Command حذف کلاس DeleteDepartmentCommand را در فولدر Commands اضـــافه کنید:

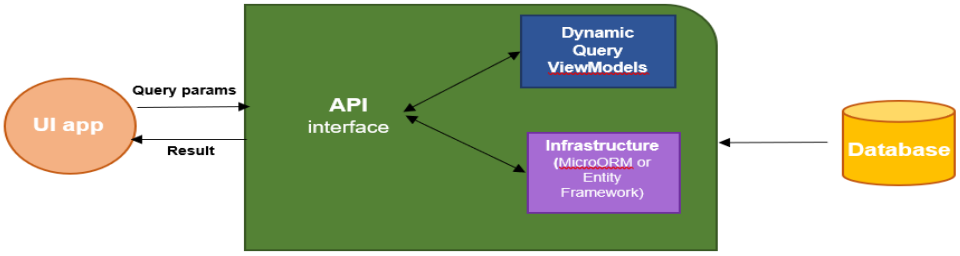
```
using MediatR;

namespace Microdev.API.Application.Commands
{
    public class DeleteDepartmentCommand : IRequest<bool>
    {

        public DeleteDepartmentCommand()
        {

        }

        public DeleteDepartmentCommand(int id)
        {
            Id = id;
        }

        public int Id { get; set; }
    }

}
```

حالا کلاس DeleteDepartmentCommandHandler را در این فولدر اضافه نمایید:

```
using MediatR;
using Microdev.Infrastructure.Data;
using System.Threading;
using System.Threading.Tasks;

namespace Microdev.API.Application.Commands
{
    public class DeleteDepartmentCommandHandler :
    IRequestHandler<DeleteDepartmentCommand, bool>
    {
        private readonly MicrodevAppDbContext _context;

        public DeleteDepartmentCommandHandler(MicrodevAppDbContext
        microdevAppDbContext)
        {
            _context = microdevAppDbContext;
        }
        public async Task<bool> Handle(DeleteDepartmentCommand request,
        CancellationToken cancellationToken)
        {
            bool isDeleted = false;
            var department = _context.Departments.Find(request.Id);

            if (department != null)
            {
                _context.Remove(department);
                await _context.SaveChangesAsync();
                isDeleted = true;
            }

            return isDeleted;
        }
    }
}
```

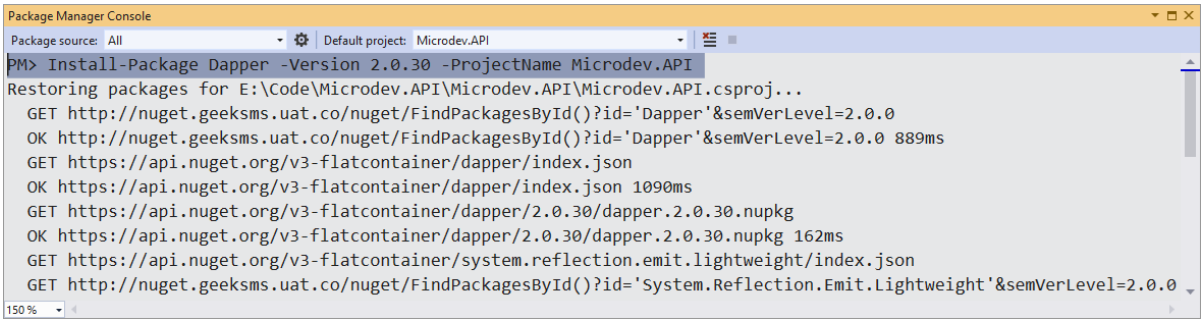
پیاده‌سازی Query

Dapper یک ORM کوچک، سبک و سریع می باشد که دارای محبوبیت زیادی است. وظیفه این ORM، مدیریت ارتباط بین محیط برنامه‌نویسی و دیتابیس می‌باشد. شما با کمک Dapper می‌توانید به سادگی دستورات خود را در قالب Stored Procedure و یا دستورات مستقیم SQL اجرا نمائید. Dapper کمک می‌کند قسمت‌هایی مانند مدیریت Connectionها، اجرای Commandهای SQLای و همچنین تبدیل نتایج درخواست (Select Result) به ViewModel بسیار ساده و سریع انجام شود.



برای نصب Dapper دستور پایین را در Package Manager Console اجرا کنید:

Install-Package Dapper -Version 2.0.30 -ProjectName Microdev.API



اکنون نیاز به یک کوئری داریم. بیایید در فولدر Application یک فولدر دیگر با نام Queries ایجاد و سپس به آن یک کلاس DepartmentQueries اضافه کنیم. این کلاس نیاز به پیاده‌سازی یک اینترفیس دارد که بیزینس موردنظر را در خود داشته باشد.

ما می‌توانیم درون ASP.NET Core، ارتباط بین DepartmentQueries و اینترفیس ارث بری‌شده را با Dependency Injection هندل کنیم.

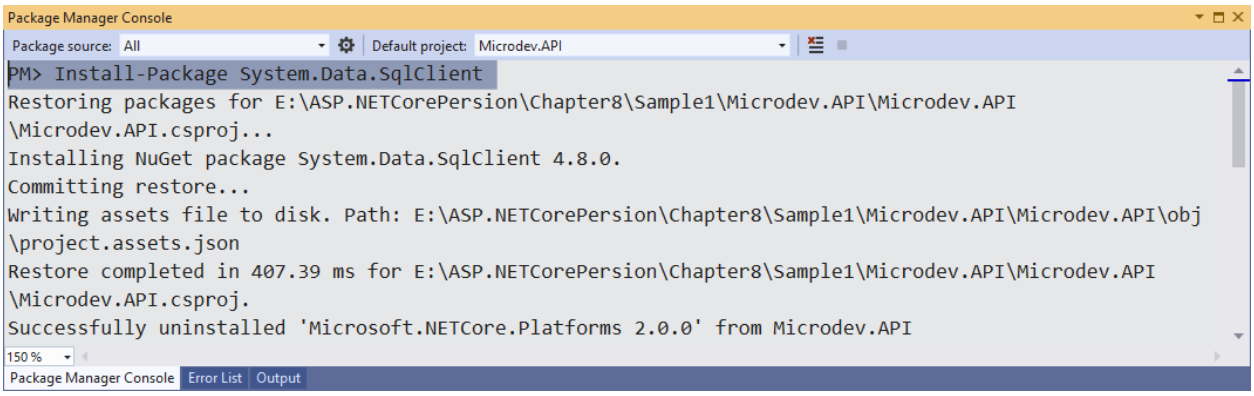
قبل از نوشتن کدهای درون DepartmentQueries بیایید نگاهی به نیازمندی‌های بیزینس بیندازیم:

- **نمایش کلیه دپارتمان ها به همراه کل حقوق در آنجا.**
- **نمایش اسامی تمامی کارمندانی که حقوق بیشتری از رئیس خود دارند.**
- **نمایش تمامی کارمندانی که بیشترین حقوق را در دپارتمان خود دارند.**
- **نمایش تمامی دپارتمان‌هایی که کمتر از ۲ نفر در آن حضور داشته باشند.**
- **نمایش تمامی کارمندانی که در همان اداره رئیس ندارند.**
- **نمایش تمامی دپارتمان‌ها به همراه تعداد افراد عضو.**

ما در این قســـمت تنها دو نمونه از این بیزنس‌ها را انجام می‌دهیم و باقی آن‌ها به عنوان تکلیف به شـــما داده می‌شود.

نکته!!

قبل از شروع کار باید System.Data.SqlClient را در Console Pakage manager نصب کنید.



از این رو:

قبل از پیاده‌ســـازی کوئری‌هایـیک کلاس‌بانـام DepartmentWithSalaryDTO در فولدر Queries ایجاد کنید.

```
namespace Microdev.API.Application.Queries
{
    public class DepartmentWithSalaryDTO
    {
        public string Name { get; set; }
        public decimal Sum { get; set; }
    }

}
```

سپس در فولدر Queries یک اینترفیس با نام IDepartmentQueries ایجاد نمایید.

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Microdev.API.Application.Queries
{
    public interface IDepartmentQueries
    {
       Task<IEnumerable<DepartmentWithSalaryDTO>>
       GetAllDepartmentsWithSalaryAsync();
        Task<IEnumerable<string>> GetExpensiveEmployeesAsync();
```

```
        //ToDo: Implement these queries at home
        //Task<IEnumerable<EmployeeDTO>> GetMostExpensiveEmployeesAsync();
        //Task<IEnumerable<DepartmentDTO>> GetThinDepartmentsAsync();
        //Task<IEnumerable<>> GetOutsiderEmployeesAsync();
        //Task<IEnumerable<DepartmentWithCountDTO>>
       GetAllDepartmentsWithCountAsync();
    }
}
```

اکنون می‌توانیم کوئری‌های خود را پیاده‌سازی کنیم. ساده‌ترین روش برای انجام این کار استفاده از Dapper است که می‌توان یک عبارت SQL را به اکستنشن متد QueryAsync در Dapper ارسال کرد. اکستنشن متد QueryAsync در Dapper به شما امکان می‌دهد داده‌ها را از دیتابیس بازیابی و سپس داده‌ها را در آبجکت مدل خود پر کنید. بیایید با هم این موضوع جذاب را شروع کنیم:

- **در فولدر Queries یک کلاس با نام DepartmentQueries.cs اضافه نمایید.**
- **سپس این کلاس باید از اینترفیس IDepartmentQueries ارث‌بری کند.**
- **در پایان هم باید متدهای درون این اینترفیس درون این کلاس پیاده‌سازی شود.**

```
using Dapper;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Threading.Tasks;

namespace Microdev.API.Application.Queries
{
    public class DepartmentQueries : IDepartmentQueries
    {
        private readonly string _connectionString;

        public DepartmentQueries(string connectionString)
        {
            _connectionString = connectionString;
        }
        /// <summary>
        /// List all departments along with the total salary there
        /// </summary>
        /// <returns></returns>
         public async Task<IEnumerable<DepartmentWithSalaryDTO>>
         GetAllDepartmentsWithSalaryAsync()
        {
                        using (IDbConnection connection = new
            SqlConnection(_connectionString))
            {
```

```
                connection.Open();

                return (await
                connection.QueryAsync<DepartmentWithSalaryDTO>
                 (@"SELECT  d.Name ,
                     ISNULL(g.Sum, 0) AS Sum
                   FROM    Department AS d
                     LEFT OUTER JOIN ( SELECT    e.DepartmentId ,
                                                 SUM(e.Salary) AS Sum
                                       FROM      Employees AS e
                                       GROUP BY  e.DepartmentId
                                     ) AS g ON d.id = g.DepartmentId
                     ORDER BY Sum DESC")).ToList();
            }
        }
        /// <summary>
       /// List employees (names) who have a bigger salary than their boss
        /// </summary>
        /// <returns></returns>
         public async Task<IEnumerable<string>>
         GetExpensiveEmployeesAsync()
        {
            using (IDbConnection connection = new
            SqlConnection(_connectionString))
            {
                connection.Open();

                return (await connection.QueryAsync<string>
                (@"SELECT  e.FirstName+' '+e.LastName
                    FROM    Employees AS e
                            JOIN Employees AS b ON e.BossId = b.Id
                    WHERE   e.Salary < b.Salary")).ToList();
            }
        }

    }
}
```

در کد بالا:

- Constructor injection یک connectionString را برای سرویس فراهم می‌کند.
- متد GetAllDepartmentsWithSalaryAsync تمامی دپارتمان‌ها رابه همراه کل حقوق در آنجا بازیابی‌می‌کند، آنها را در حافظه ذخیره و سپس‌لیستی ازآبجکت‌های DepartmentWithSalaryDTO را برمی‌گرداند.

- **متد GetExpensiveEmployeesAsync هم‌نام تمام کارمندانی که حقوق بیشتری از رئیس خود دریافت کردند را برمی‌گرداند.**

از آنجا که ما یک سرویس جدید ایجاد کرده‌ایم، به یاد داشته باشید که کد پایین را به متد ConfigureService خود در فایل Startup.cs اضافه نمایید.

```
services.AddScoped<IDepartmentQueries, DepartmentQueries>(serviceProvider =>
{
    return new DepartmentQueries(applicationConnectionString);
});
```

کد بالا نحوه پیکربندی DI برای DepartmentQueries را نمایش می‌دهد.

```
using MediatR;
using Microdev.API.Application.Commands;
using Microdev.API.Application.Queries;
using Microdev.Infrastructure.Data;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Microdev.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
             var applicationConnectionString =
             Configuration.GetConnectionString("MicrodevAppDbContext");

            services.AddDbContext<MicrodevAppDbContext>(options =>
            options.UseSqlServer(applicationConnectionString));
            services.AddMediatR(typeof(CreateDepartmentCommandHandler));
            services.AddScoped<IDepartmentQueries,
            DepartmentQueries>(serviceProvider =>
            {
```

ریجستر سرویس DepartmentQueries

```
            return new DepartmentQueries(applicationConnectionString);
        });

        services.AddControllers();
    }

     public void Configure(IApplicationBuilder app, IWebHostEnvironment
     env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
  }
}
```

افزودن Controllerها

حالا که Commandها و Queryها نوشـــته شـــدند، شـــما باید یک کنترلر را برای موجودیت Department پیاده‌سـازی کنید. کنترلر برای مشـاهده لیسـتی از دپارتمان‌ها، ایجاد یک دپارتمان جدید و ویرایش جزئیات یک دپارتمان موجود، اکشن‌متدهایی را در معرض نمایش قرار می‌دهد.

همـانطور کـه می‌دانیـد، کلاس‌هـای کنترلر بـه طور پیش‌فرض در فولـدر Controllers قرار می‌گیرنـد و از کلاس ControllerBase ارث‌بری می‌کنند.

- **در فولدر Controllers یک کلاس با نام DepartmentsController ایجاد نمایید.**
- **این کنترلر باید از کلاس ControllerBase ارث‌بری کند.**

کد پایین را در کنترلر موردنظر قرار دهید.

```
using Microdev.API.Application.Commands;
using MediatR;
using Microsoft.AspNetCore.Mvc;
```

```
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace Microdev.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class DepartmentsController : ControllerBase
    {
        readonly IMediator _mediator;
        readonly ILogger<DepartmentsController> _logger;

        public DepartmentsController(IMediator mediator,
        ILogger<DepartmentsController> logger)
        {
            _mediator = mediator;
            _logger = logger;
        }

    }

}
```

در کد بالا:

- **ما یک کلاس API controller را بدون هیچ متدی تعریف کردیم.**
- **با افزودن اتریبیوت [ApiController] به DepartmentsController اطمینان حاصل می‌شود که کنترلر فقط به درخواست‌های Web API پاسخ می‌دهد.**
- **همچنین درون DepartmentsController از DI برای تزریق IMediator و < ILogger DepartmentsController> استفاده می‌شود.**

افزودن اکشن متدها

وقتی در مورد یک اکشن‌متد صحبت می‌شود، در حقیقت منظور یک متد در یک کلاس است. بنابراین، یک کنترلر متدهایی دارد که ما آنها را اکشن‌متد می‌نامیم. اکشن‌متدها با استفاده از سطح دسترسی Public و احتمالاً پارامترهای ورودی تعریف می‌شوند و می‌توانند هر نوعی را برگردانند.(اما معمولا نوع IActionResult است.)

اکشن‌متدها همچنین می‌توانند با یکی از افعال [HttpGet]، [HttpPut]، [HttpPost]، [HttpDelete]، [HttpHead] و [HttpPatch] نیز همراه باشند.

بیایید با هم APIهایی را به کنترلر DepartmentsController اضافه کنیم:

```
using Microdev.API.Application.Commands;
using MediatR;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;
using System.Collections.Generic;
using Microdev.API.Application.Queries;

namespace Microdev.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class DepartmentsController : ControllerBase
    {
        readonly IMediator _mediator;
        readonly ILogger<DepartmentsController> _logger;
        private readonly IDepartmentQueries _departmentQueries;

         public DepartmentsController(IMediator mediator,
         ILogger<DepartmentsController> logger, IDepartmentQueries
         departmentQueries)
        {
            _mediator = mediator;
            _logger = logger;
            _departmentQueries = departmentQueries;
        }

        [HttpGet("departments-salary")]
          public async
          Task<ActionResult<IEnumerable<DepartmentWithSalaryDTO>>>
          GetAllDepartmentsWithSalaryAsync()
        {
          var departments = await
         _departmentQueries.GetAllDepartmentsWithSalaryAsync();

            return Ok(departments);
        }

        [HttpGet("expensive-employees")]
         public async Task<ActionResult<IEnumerable<string>>>
         GetExpensiveEmployeesAsync()
        {
           var employees = await
          _departmentQueries.GetExpensiveEmployeesAsync();

            return Ok(employees);
        }

    }
```

```
}
```

هنگامی که یک درخواست از طرف سرور وارد می‌شود، ما URLی داریم که ASP.NET Core می‌گیرد و سعی می‌کند آن را با مسیرهای موجود در پروژه مطابقت دهد. سپس هنگامی که یک مسیر پیدا شد، به اکشن و کنترلر آن مسیر نگاه و در نهایت آن را اجرا می‌کند.

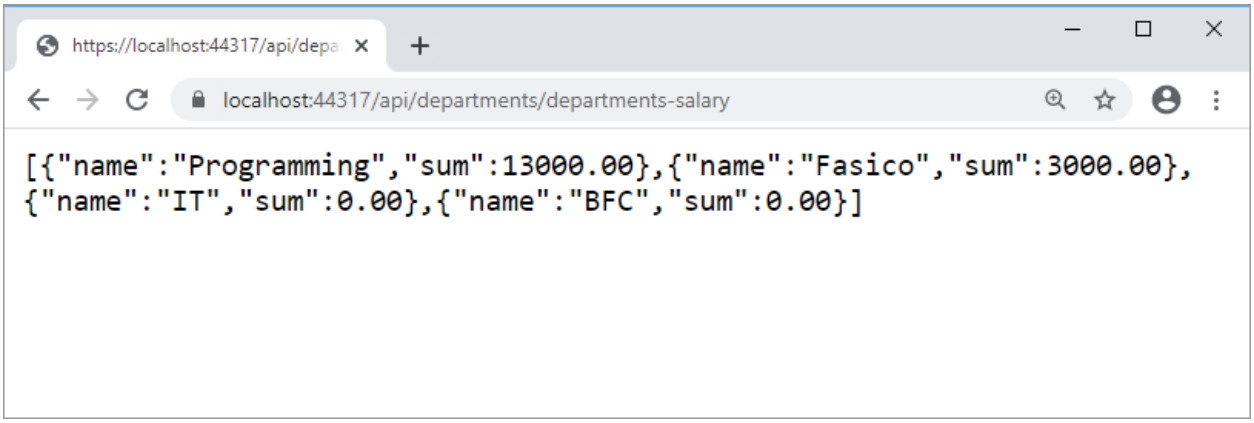
متدهای بالا دو GET endpoint پایین را پیاده‌سازی می‌کنند:

GET /api/departments/departments-salary

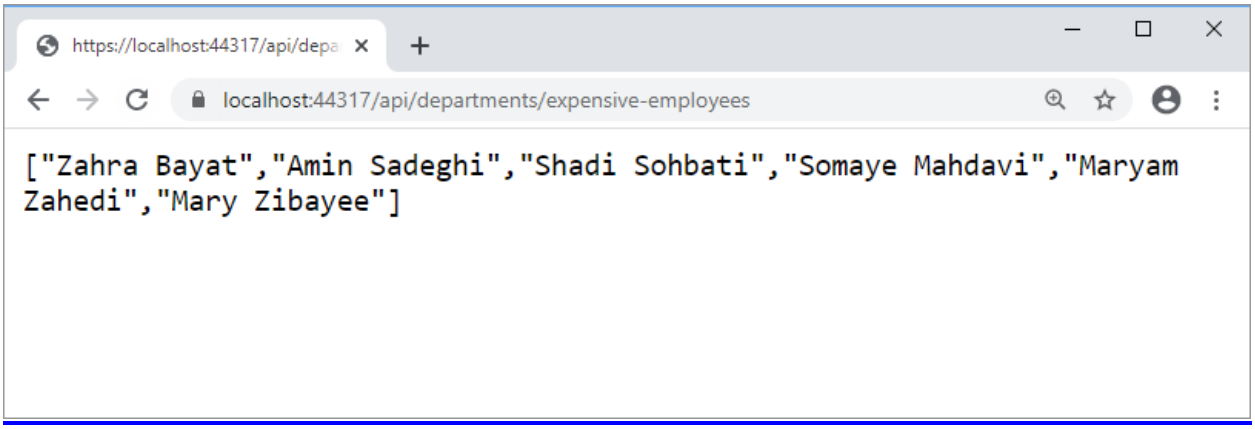
GET /api/departments/expensive-employees

با صدا زدن این دو Endpoint در مرورگر، برنامه را امتحان کنید.

http://localhost:44317/api/departments/departments-salary



http://localhost:44317/api/departments/expensive-employees

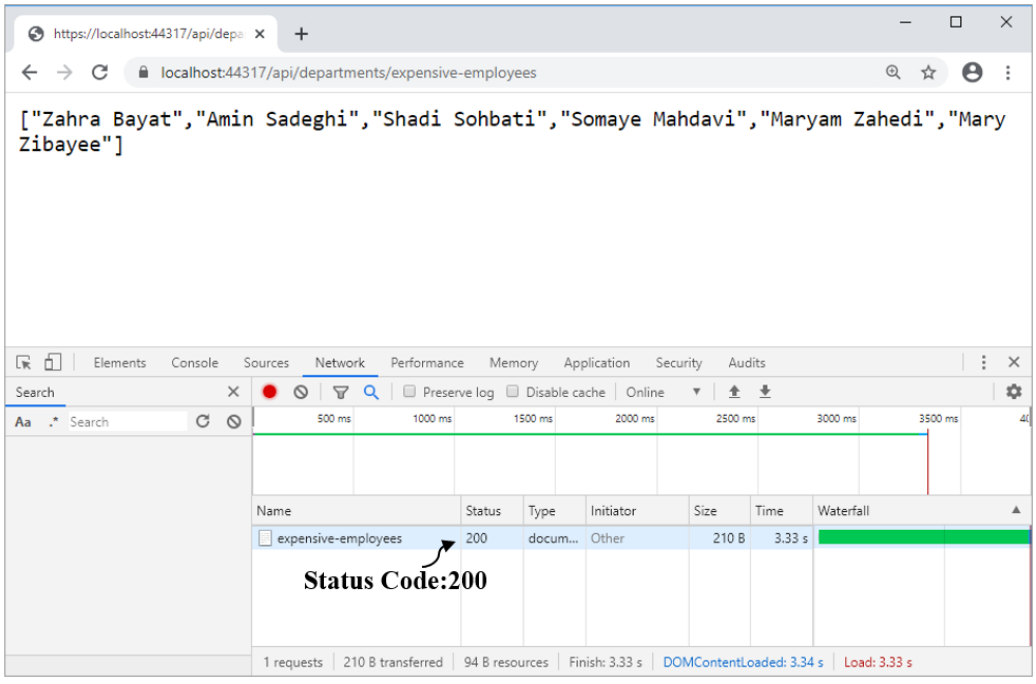


متدهای GetAllDepartmentsWithSalaryAsync و GetExpensiveEmployeesAsync خروجی ASP.NET Core را Serialize کرده و خروجی JSON را در Response می‌نویسد.

بنابراین همانطور که می‌بینید، با فراخوانی متد GetAllDepartmentsWithSalaryAsync پاسخ JSON پایین تولید می‌شود:

```
[
    {
        "name": "Programming",
        "sum": 13000
    },
    {
        "name": "Fasico",
        "sum": 3000
    },
    {
        "name": "IT",
        "sum": 0
    },
    {
        "name": "BFC",
        "sum": 0
    }
]
```

نوع برگشتی متدهای GetAllDepartmentsWithSalaryAsync و GetExpensiveEmployeesAsync نوع ActionResult<T> است که می‌تواند طیف گسترده‌ای از Status Codeهای HTTP را نشان دهد. **به عنوان مثال:** اگر هیچ خطایی رخ ندهد، کد ۲۰۰ را به همراه JSON برمی‌گرداند، در غیر این صورت کد 5XX برگردانده می‌شود.



ایجاد اکشن‌متد CreateDepartment

در قدم بعدی می‌خواهیم یک اکشـــن‌متد برای ایجاد یک دپارتمان جدید ایجاد کنیم، این متد نام دپارتمان را از Body درخواست HTTP گرفته و یک دپارتمان جدید را ایجاد می‌کند.

اکشن‌متد پایین را به DepartmentController اضافه نمایید:

```
// POST: api/Departments
[HttpPost]
public async Task<ActionResult<bool>>
CreateDepartmentAsync([FromBody]CreateDepartmentCommand
createDepartmentCommand)
{
    bool commandResult = false;

    _logger.LogInformation(
       "----- Sending command: {CommandName} - {IdProperty}: {CommandId}
       ({@Command})",
        createDepartmentCommand.GetType().Name,
        nameof(createDepartmentCommand.Name),
        createDepartmentCommand.Name,
        createDepartmentCommand);

    commandResult = await _mediator.Send(createDepartmentCommand);

    if (!commandResult)
    {
        return BadRequest();
    }

    return Ok();
}
```

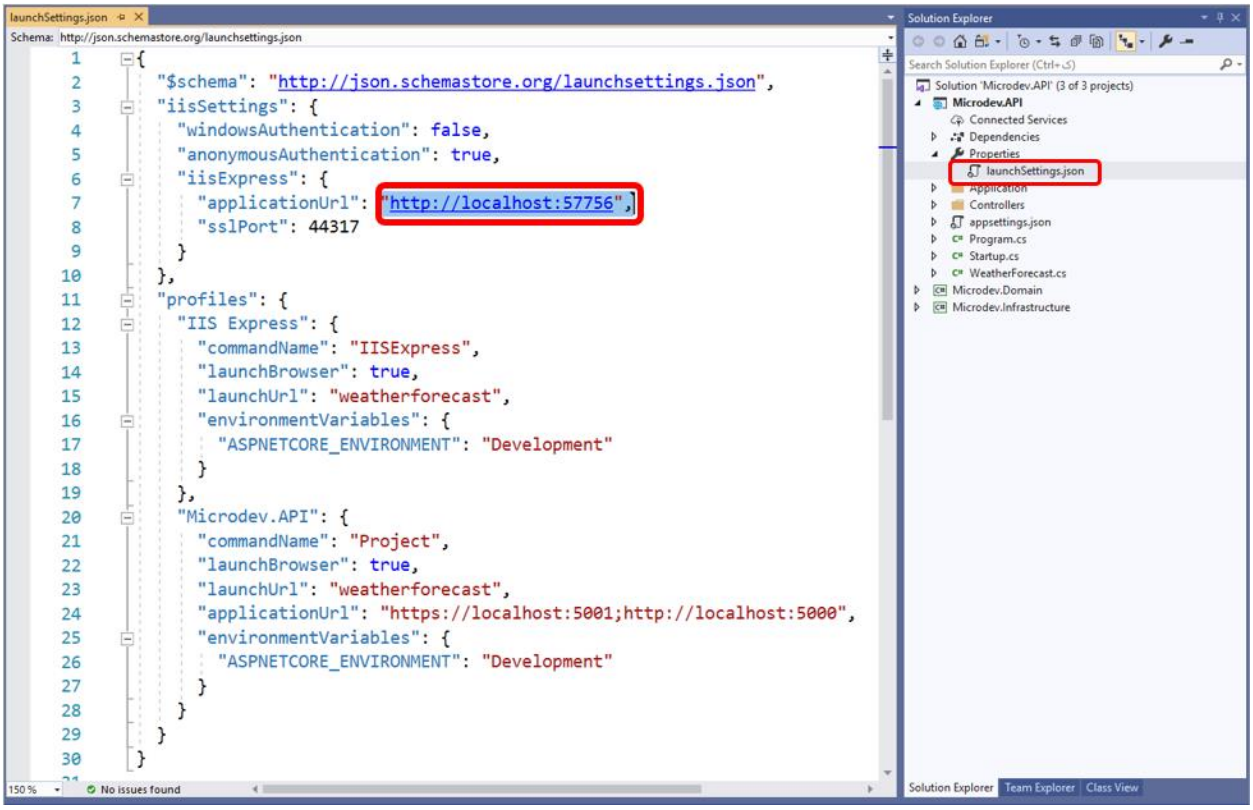
تست APIها با استفاده از PowerShell

حالا که اکشـن CreateDepartment را به APIها اضـافه کردید، وقت آن رسـیده که آن را تسـت نمایید. من می‌خواهم برای تسـت از دسـتور curl پاورشـل اسـتفاده کنم اما شـما می‌توانید تسـت APIها را با Fiddler یا Postman هم انجام دهید.

نکته!!

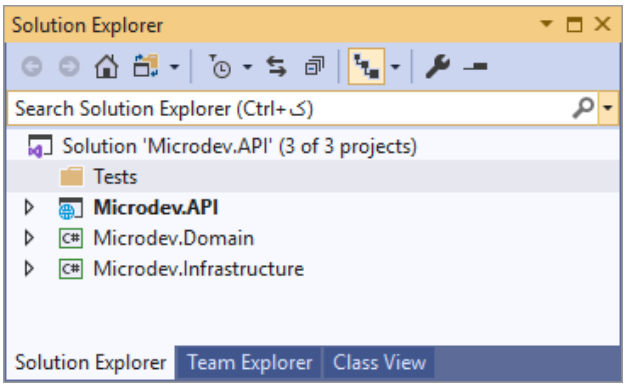
IIS Express به صـورت تصـادفی به برنامه‌های شـما پورت اختصـاص می‌دهد که می‌توانید آن را در فایل lanchSettings.json مشـاهده و یا در صـورت نیاز تغییر دهید. شـما می‌توانید برای ارسـال

درخواســـت post از این url در فـایـل CreateDepartment.ps1 (در ادامـه توضـــیح داده شـــده است)استفاده نمایید.



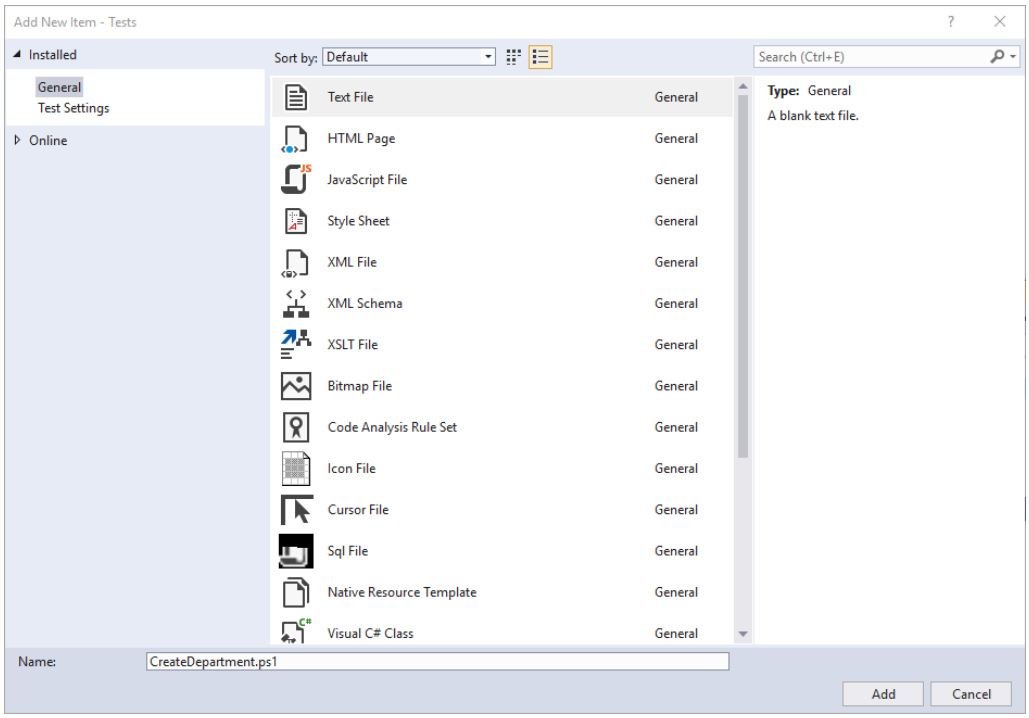
برای تست APIها:

- ابتدا بر روی Solution راست کلیک کنید.
- سپس گزینه‌ی Add و در زیر منوی آن New Folder را انتخاب کنید.
- در پایان نام فولدر را Tests بگذارید.



ایجاد CreateDepartment.ps1 :

- **بر روی فولدر Tests کلیک راست کنید و گزینه Add→ New Item را انتخاب نمایید.**
- **در کادر Add New Item→ قالب Text File را برگزینید.**
- **نام فایل را CreateDepartment.ps1 گذاشته و سپس رو دکمه Add کلیک کنید.**



- **حالا در فایل CreateDepartment.ps1 کدهای پایین را اضافه نمایید:**

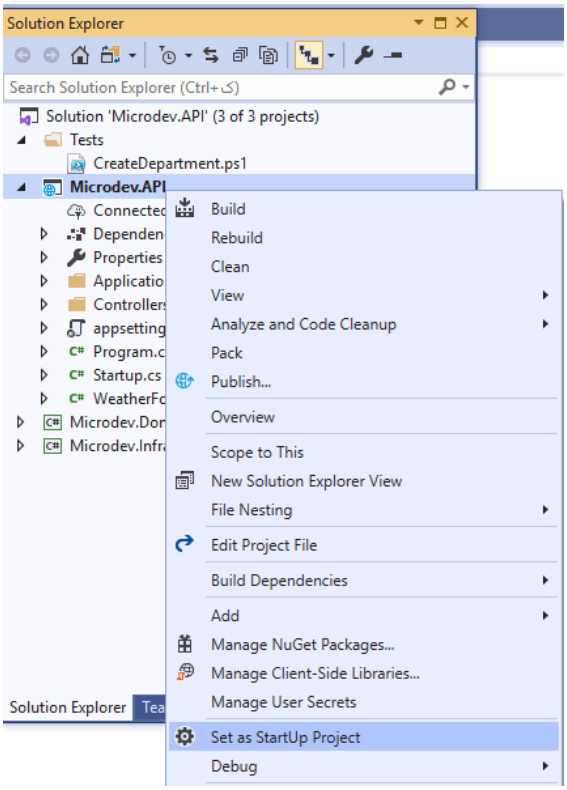
```
$uri = "http://localhost:57756/api/departments/"
$body = '
{
 "name":"New Department"
}
'
$headers = @{
    'Content-Type'='application/json'
    'Authorization' = 'TokenValue'
}
curl -Uri $uri -Method Post -Headers $headers -Body $body
```

در کد بالا:

با دستور curl پاورشل، یک درخواست HTTP همراه با بدنه ورودی به API خود فرستادیم، تا یک Response به صورت JSON به ما برگرداند.

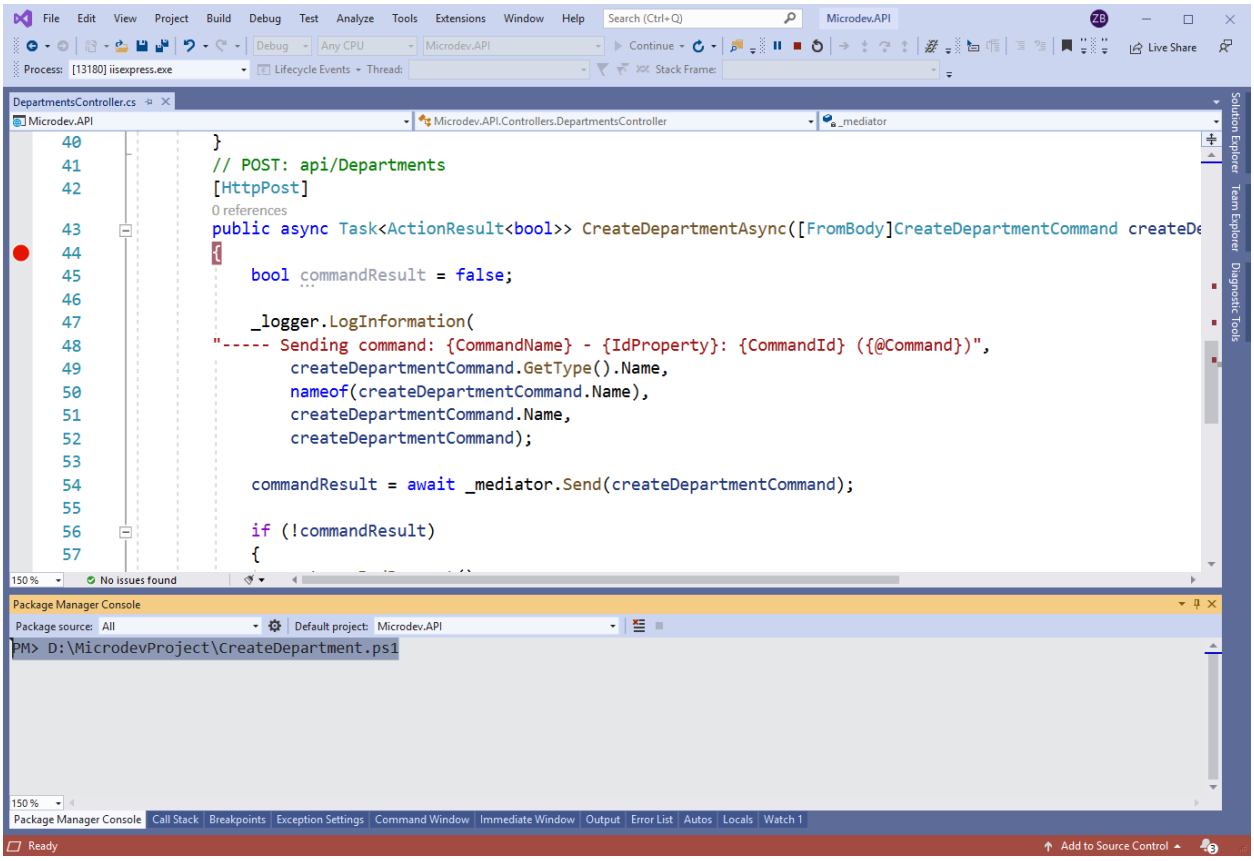
- **Uri-: URI مخفف Uniform Resource Identifier می‌باشد و برای مشخص نمودن منبع موردنظر Web Request است. این پارامتر فقط از HTTP یا HTTPS پشتیبانی می‌کند.**
- **Method-: این پارامتر متد استفاده شده برای Web Request را مشخص می‌کند. مقادیر قابل قبول برای این پارامتر، تمام متدهای HTTP مانند GET، POST، PUT، DELETE و ... است.**
- **Headers-: این پارامتر Headerهای یک Web Request را مشخص می‌کند. شما می‌توانید یک HashTable یا Dictionary را وارد کنید.**
- **Body-: از پارامتر Body- برای مشخص کردن بدنه Request استفاده می‌شود.**

حالا بر روی پروژه Microedev.API راست کلیک نمایید و Set as StartUp Project را انتخاب نمایید.



اپلیکیشن را اجرا و سپس دستور زیر را در Console Package Manager اجرا کنید. (آدرس پایین مکان وجود فایل CreateDepartment.ps1 می‌باشد.)

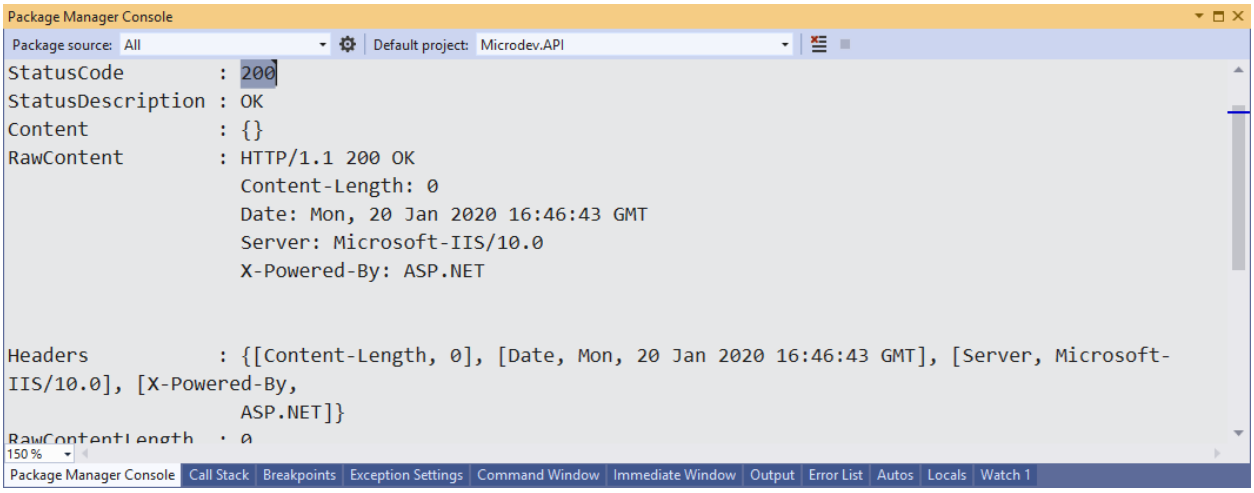
D:\MicrodevProject\CreateDepartment.ps1



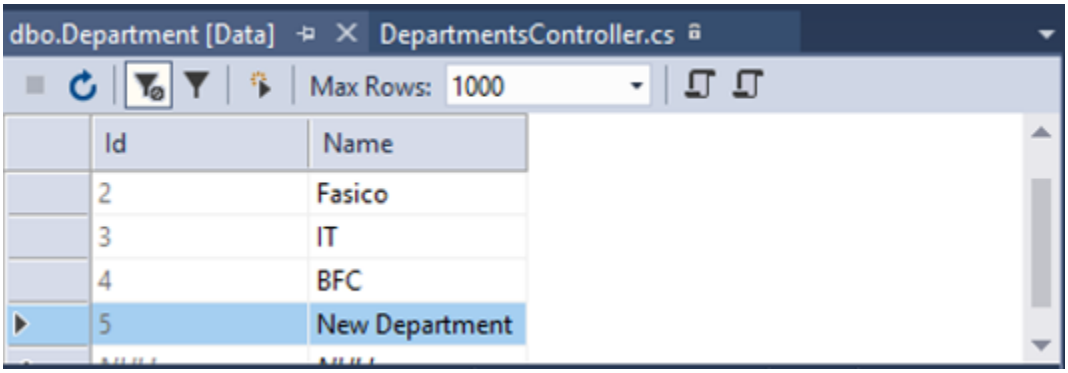
وقتی درخواست اجرا شد، در شکل زیر بر روی Breakpointی که گذاشته بودیم می‌رود و اکشن Post همانطور که انتظار داشتیم عمل می‌نماید.

```
42  // POST: api/Departments
43  [HttpPost]
    0 references | Ali, 1 day ago | 1 author, 2 changes
44  public async Task<ActionResult<bool>> CreateDepartmentAsync([FromBody]CreateDepartmentCommand createDepartmentCommand)
45  {
46      bool commandResult = false;
47
48      _logger.LogInformation(
49          "----- Sending command: {CommandName} - {IdProperty}: {CommandId} ({@Command})",
50          createDepartmentCommand.GetType().Name,
51          nameof(createDepartmentCommand.Name),
52          createDepartmentCommand.Name,
53          createDepartmentCommand);
54
55      commandResult = await _mediator.Send(createDepartmentCommand);
56
57      if (!commandResult)
58      {
59          return BadRequest();
60      }
61
62      return Ok();
63  }
```

در صـورت موفقیت آمیز بودن عملیات، متد DepartmentAsync کد وضـعیت HTTP 200 را برمی‌گرداند و خروجی زیر نمایش داده می‌شـود. این کد نشـان دهنده این اسـت که Department جدید در دیتابیس ذخیره شده است.



برای مشـاهده داده‌ها، بر روی جدول Department کلیک راسـت و سـپس گزینه View Data را انتخاب کنید.



افزودن اکشن‌متد UpdateDepartment

UpdateDepartment مشابه CreateDepartment است فقط از HTTP PUT استفاده کند.

اکشن‌متد UpdateDepartmentAsync را مانند کد زیر به کنترلر DepartmentController اضافه کنید:

```
// PUT: api/Departments
[HttpPut]
public async Task<ActionResult> UpdateDepartmentAsync(UpdateDepartmentCommand
updateDepartmentCommand)
{
```

```
    bool commandResult = false;

    _logger.LogInformation(
        "----- Sending command: {CommandName} - {IdProperty}: {CommandId}
        ({@Command})",
         updateDepartmentCommand.GetType().Name,
         nameof(updateDepartmentCommand.Id),
         updateDepartmentCommand.Id,
         updateDepartmentCommand);

    commandResult = await _mediator.Send(updateDepartmentCommand);

    if (!commandResult)
    {
        return BadRequest();
    }

    return Ok();
}
```

تست متد **UpdateDepartmentAsync**:

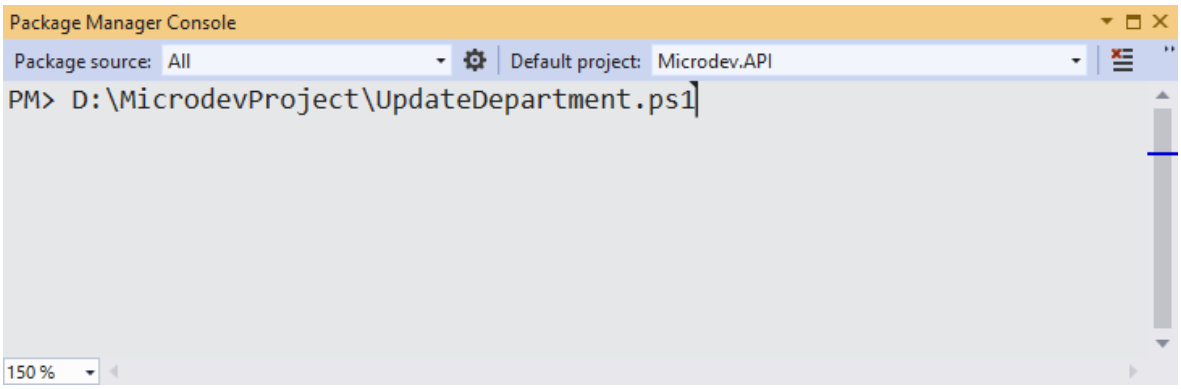
- روی فولدر **Tests** کلیک راست کنید.
- در کادر **New Item→Add** قالب **Text File** را برگزینید.
- نام فایل را **UpdateDepartment.ps1** گذاشته و سپس بر روی **Add** کلیک کنید.

حالا کد پایین را در این فایل اضافه کنید:

```
$uri = "http://localhost:57756/api/departments/"

$body = '
{
 "id":5,
 "name":"Updated Department"
}
'

$headers = @{
    'Content-Type'='application/json'
    'Authorization' = 'TokenValue'
}

curl -Uri $uri -Method Put -Headers $headers -Body $body
```

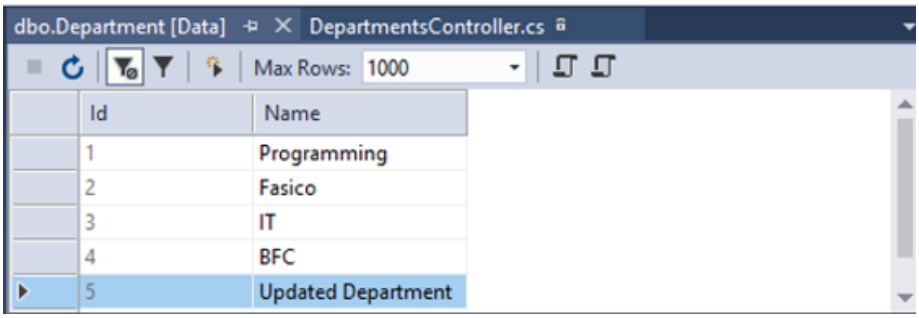
پس از اجرای اپلیکیشن، دستور زیر را در Console Package Manager اجرا کنید.

D:\MicrodevProject\UpdateDepartment.ps1



دستورات بالا نام Department با Id=5 را تغییر می‌دهد.

برای مشاهده داده‌ها بر روی جدول Department کلیک راست کنید، سپس گزینه View Data را انتخاب نمایید.(یا اگر جدول باز است دکمه های Shift+Alt+R را بزنید)



افزودن اکشن‌متد DeleteDepartment

اکشـن‌متد DeleteDepartmentAsync را مانند کد زیر به کنترلر DepartmentController اضـافه کنید:

```
// DELETE: api/Departments/id
[HttpDelete("{id:int}")]
public async Task<ActionResult> DeleteDepartmentAsync(int id)
{
    bool commandResult = false;

    var deleteDepartmentCommand = new DeleteDepartmentCommand(id);

    _logger.LogInformation(
       "----- Sending command: {CommandName} - {IdProperty}: {CommandId}
       ({@Command})",
        deleteDepartmentCommand.GetType().Name,
        nameof(deleteDepartmentCommand.Id),
        deleteDepartmentCommand.Id,
        deleteDepartmentCommand);
```

```
        commandResult = await _mediator.Send(deleteDepartmentCommand);

    if (!commandResult)
    {
        return BadRequest();
    }

    return Ok();
}
```

تست اکشن DeleteDepartmentAsync :

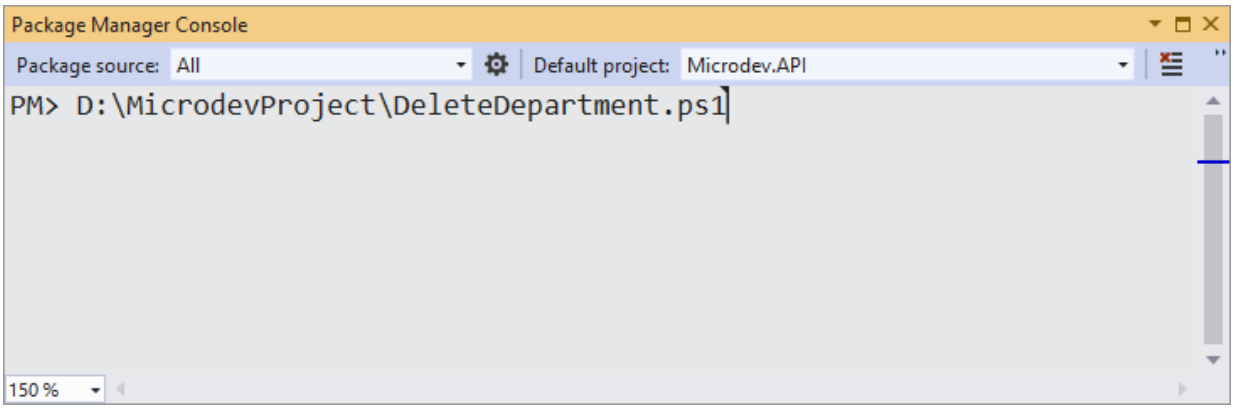
- روی فولدر Tests کلیک راست کنید.
- در کادر New Item→Add قالب Text File را برگزینید.
- نام فایل را DeleteDepartment.ps1 گذاشته و سپس بر روی Add کلیک کنید.

حالا کد پایین را در این فایل اضافه کنید:

```
$uri = "http://localhost:57756/api/departments/5"

$headers = @{
    'Content-Type'='application/json'
    'Authorization' = 'TokenValue'
}

curl -Uri $uri -Method Delete -Headers $headers
```

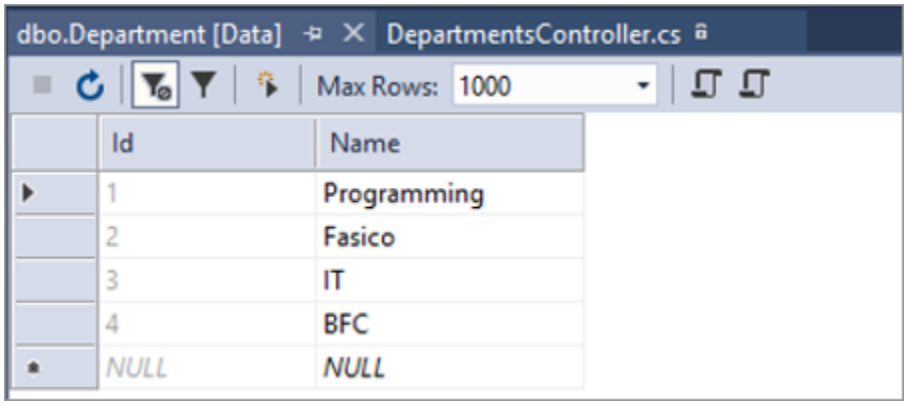
پس از اجرای برنامه، دستور زیر را در Console Package Manager اجرا کنید.

D:\MicrodevProject\DeleteDepartment.ps1



دستورات بالا Departmentی با Id=5 را از جدول Department حذف می‌کند.

برای مشاهده داده‌ها بر روی جدول **Department** کلیک راست کنید، سپس گزینه **View Data** را انتخاب نمایید.(یا اگر جدول باز است دکمه های **Shift+Alt+R** را بزنید)



مسیر پروژه نمونه انجام شده در Github:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter8/Sample1

تمرین

قبل از شروع فصل بعدی در مورد سوالات زیر تحقیق کنید:

- ✓ **Authentication و Authorization چیست؟**
- ✓ **چطور Authentication و Authorization را در اپلیکیشن پیاده‌سازی کنیم؟**

Interview Questions

Q1: What is the benefit of using REST in Web API?

Q2: Name some of the commonly used HTTP methods used in REST based architecture?

Q3: What is ASP.Net Core Web API?

Q4: What are main return types supported in Web API?

Q5: Which protocol Web API supports?

Q6: By default, Web API sends HTTP response with which of the status code for all uncaught exception?

Q7: How can you restrict access methods to specific HTTP verbs in ASP.Net Core Web API?

Q8: What is Entity Framework Core?

Q9: How does Entity Framework work?

Q10: What is DbContext and DbSet in Entity Framework Core?

Quiz

Q1: Consumer of our API is _____.

1. Single Page Applications
2. Mobile Applications
3. Web services
4. All

Q2: Which protocol is used in REST?

1. HTTP
2. AMQP
3. UDP
4. FTP

Q3: HTTP POST method is used to_____.

1. Fetch data without modifying server data.
2. Update server data.
3. Remove a resource.
4. Create new resource.

Q4: Suppose you need to update a department's name. Which HTTP action verb is the best fit for this request?

1. GET
2. POST
3. PUT
4. DELETE

Q5: What is JSON?

1. JSON is a format for storing data.
2. JSON is text, and we can convert any object into it.
3. JSON is a format for transporting data.
4. All

Q6: Entity is a class that defined to_____.

1. represent a table for storing data in database.
2. represent column of table in database.
3. represent rows in the tables.
4. All

Q7: Entity Framework Core is_____.

1. an ORM that enable you to interact with database.

2. a library to persist data.
3. a library that maps entity class to table.
4. All.

Q8: DbContext is _____.

1. used for all your database calls
2. bridge between your domain or entity classes and the database.
3. Set of APIs simplify your application interaction with the database.
4. All.

Q9: What is Data Seeding?

1. Data seeding is a way to populate the database with at the time it is created.
2. Data seeding is the process of populating a database with an initial set of data.
3. Data seeding provide initial values for lookup lists, for demo purposes, proof of concepts etc.
4. All.

Q10: What is Migration?

1. Migration is a way to keep the database schema in sync with the EF Core model.
2. Migration is a feature in EF Core to create database with the EF Core model.
3. Migration provides a way to incrementally update the database.
4. All.

Q11: What is MediatR?

1. MediatR is an implementation of the mediator pattern.
2. MediatR is a library to implementing the CQRS pattern.
3. MediatR is a library to providing a way to have “one model in goes to something to get one model out” without creating bloated service layers.
4. All

Q12: What is Dapper?

1. Dapper is a library that extends the IDbConnection.
2. Dapper is a library that providing useful extension methods to query your database.
3. Dapper is a library to write SQL query with great performance.
4. All.

Answer

1-Correct Answer: ALL

2-Correct Answer: HTTP

3-Correct Answer: Create new resource

4-Correct Answer: PUT

5-Correct Answer: All

6-Correct Answer: Represent a table for storing data in database

7-Correct Answer: All

8-Correct Answer: All

9-Correct Answer: All

10-Correct Answer: All

11-Correct Answer: All

12-Correct Answer: All

خلاصه فصل

- ✓ Web API تعدادی متد است که می‌تواند برای دسترسی یا تغییر داده‌ها بر روی یک سرور استفاده شود.
- ✓ از کنترلرها برای گروه‌بندی منطقی بخشی از اپلیکیشن استفاده می‌شود.
- ✓ Web API به جای HTML UI، یک شی JSON برمی‌گرداند.
- ✓ یک Domain Model مجموعه کلاس‌هاییست که برای اهداف تجاری موردنیاز است.
- ✓ EF Core یک ORM است که به شما امکان می‌دهد با یک دیتابیس ارتباط برقرار کنید.
- ✓ EF Core کلاس Entity را به جداول، Propertyهای Entity را به ستون‌های جداول و Instanceهایی از شیء Entity را با سطرهای جداول، Map می‌کند.
- ✓ برای نصب EF Core، باید پکیج Microsoft.EntityFrameworkCore.SqlServer NuGet را نصب کنیم.
- ✓ DbContext برای فراخوانی‌های دیتابیس استفاده می شود.
- ✓ از اکستنشن‌متد AddDbContext <T> برای رجیستر DbContext استفاده می‌شود.
- ✓ Data Seeding فرایند مقداردهی اولیه جداول در دیتابیس است.
- ✓ Migration راهی برای ایجاد و آپدیت دیتابیس است.
- ✓ برای افزودن Migration، دستور Add-Migration MigrationName را در Package Manager Console اجرا کنید.
- ✓ دستور Update-Database با استفاده از connection string به دیتابیس متصل می‌شود.
- ✓ کتابخانه MediatR یک پیاده‌سازی ساده از دیزاین پترن Mediator است که باعث کاهش وابستگی بین اشیاء در برنامه‌های شما می‌شود.
- ✓ ما از MediatR استفاده می‌کنیم تا Queryها را از Commandها جدا کنیم.
- ✓ می‌توانیم با PowerShell، Postman، jQuery، Angular یا هر Client دیگر API را فراخوانی کنید.

فصل نهم: Authentication و Authorization چیست؟

آنچه خواهید آموخت:

- مقدمه ای در مورد Authenticationو Authorization
- پیاده‌سازی Authentication و Authorization در ASP.NET Core
- پیکربندیASP.NET Core Identity
- افزودن فرم لاگین

مقدمه ای در مورد Authentication و Authorization

امنیت نگرانی اصلی هر وب اپلیکیشنی است بنابراین، تمام اپلیکیشن‌های وب نیاز به اجرای مکانیزم‌های امنیتی قوی دارند. این موضوعی بسیار مهم است و باید جدی گرفته شود.

وقتی می‌خواهید امنیت را در اپلیکیشن خود داشته باشید، باید دو جنبه مهم را در نظر بگیرند:

- ✓ **Authentication یا احراز هویت : فرآیند تعیین اینکه شما چه کسی هستید.**
- ✓ **Authorization یا مجوز : روند تعیین کارهایی که مجاز به انجام آن هستید.**

به طور کلی، Authentication فرآیند تأیید هویت فرد یا سیستم است، در حالی که Authorization، فرآیند تأیید این است که کاربر Authenticate شده، آیا از مجوز کافی برای انجام کارهای خاص برخوردار است یا خیر؟

نکته!!

قبل از اینکه مشخص نمایید کاربر به انجام چه کارهایی مجاز است، باید بدانید که کاربر کیست. بنابراین اول Authentication انجام می‌شود و به دنبال آن باید Authorization انجام شود.

Authentication در ASP.NET Core

همانطور که بالاتر گفته شد، اضافه کردن Authentication به هر وب اپلیکیشنی ضروری است. خوشبختانه ASP.NET Core برای حل این موضوع پیچیده، یک Authentication و Authorization توکار درون خود دارد.

ASP.NET Core Identity سیستمی برای اضافه کردن قابلیت Login به اپلیکیشن‌هاست که به شما کمک می‌کند ویژگی‌های امنیتی[۱۳] و هویتی[۱۴] را به اپلیکیشن خود اضافه کنید.

ASP.NET Identity با استفاده از کلاس‌های UserManager و SignInManager مکانیزمی را برای مدیریت و تعیین هویت کاربران فراهم می‌کند.

درک و پیاده‌سازی امن اپلیکیشن با استفاده از فریم‌ورک ASP.NET Core Identity بسیار ساده شده است.

---

[۱۳] Security

[۱۴] Identity

پیاده‌سازی Authentication در ASP.NET Core

اضـافه کردن Authentication ضـروری اسـت اما قبل از اینکه دسـت به کد شـویم و Authorization را در ASP.NET Core را بررسی کنیم، بیایید جریان معمولی یک اپلیکیشن ASP.NET Core را با هم ببینیم:

- **Client برای شناسایی کاربر جاری یک شناسه[15] و یک رمز[16] به برنامه ارسال می‌کند.**
- **برنامه ASP.NET Core تأیید می‌کند که شناسه با کاربر شناخته شده‌ی اپلیکیشن مطابقت دارد و رمز مربوطه صحیح است.**
- **اگر شناسه و رمز معتبر باشد اپلیکیشن می‌تواند Principal درخواست جاری را تنظیم کند.**

Principal چیست؟

در ASP.NET Core، هر درخواست به یک کاربر مرتبط است، که به آن Principal گفته می‌شود. کاربری که Authenticate نشده باشد، یک Anonymous user است.

برای دسترسی به Principal درخواست جاری، می‌توانید از HttpContext.User استفاده کنید.

برای افزودن Authentication باید:

- **AddAuthentication را در متد ConfigureService صدا بزنیم. این متد مسئول تنظیم Principal جاری برای یک Request است.**
- **و در متد Configure لازم است که UseAuthentication را فراخوانی کنیم تا همه چیز را برای Authentication اعتبارسنجی تنظیم کند.**

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.Services;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
```

[15] Identifier
[16] Secret

```
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<EmployeeService>();
            services.AddTransient<TestEmployeeService>();
            services.AddTransient<Func<EnviromentServiceType,
           IEmployeeService>>(serviceProvider => key =>
            {
                switch (key)
                {
                    case EnviromentServiceType.ProductionEmployeeService:
                       return
                       serviceProvider.GetRequiredService<EmployeeService>();
                    case EnviromentServiceType.TestEmployeeService:
                      return
                      serviceProvider.GetRequiredService<TestEmployeeService>
                      ();
                    default:
                       throw new NotImplementedException($"Service of type
                      {key} is not implemented.");
                }
            });
            services.AddAuthentication();
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment
        env)
        {

            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseAuthentication();
            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "employee",
                    pattern:
                   "{controller=Employee}/{action=GetAllEmployee}/{id?}");
```

افزودن سرویس AddAuthentication

افزودن AuthenticationMiddleware به Pipeline

```
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });


    }
}
}
```

پیکربندی ASP.NET Core Identity

برای پیکربندی ASP.NET Core Identity، باید مراحل زیر را دنبال کنید:

۱) تعریف کلاس User:

Authentication به اپلیکیشن‌ها اجازه می‌دهد تا یک کاربر خاص را شناسایی کنند. IdentityUser حاوی اطلاعاتی در مورد کاربران اپلیکیشن است. بنابراین اولین مرحله، ایجاد یک کلاس جدید به نام User در فولدر Models اپلیکیشن است که باید از کلاس IdentityUser ارث‌بری کند.

```
using Microsoft.AspNetCore.Identity;

namespace Microdev.ASPNETCore.Models
{
    public class User : IdentityUser
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

    }

}
```

User باید از IdentityUser ارث‌بری کند.

نکته!!

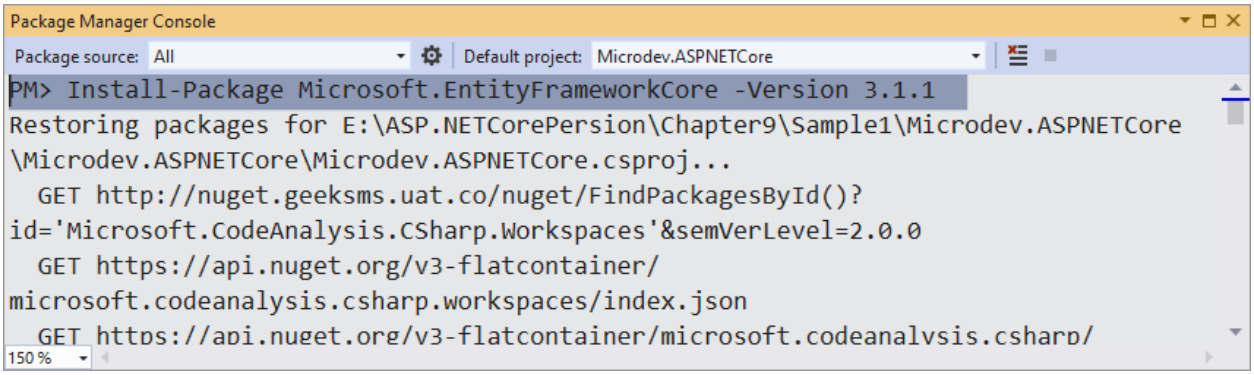
کلاس پایه IdentityUser دارای Propertyهای اختیاری است که بسته به نیاز اپلیکیشن ممکن است مورد استفاده قرار گیرد (به عنوان مثال: PasswordHash) اما علاوه بر این Propertyها، شما می‌توانید Propertyهایی برای ذخیره نام‌های کاربری و نام خانوادگی کاربر نیز داشته باشید.

۲) اضافه کردن DbContext برای Identity:

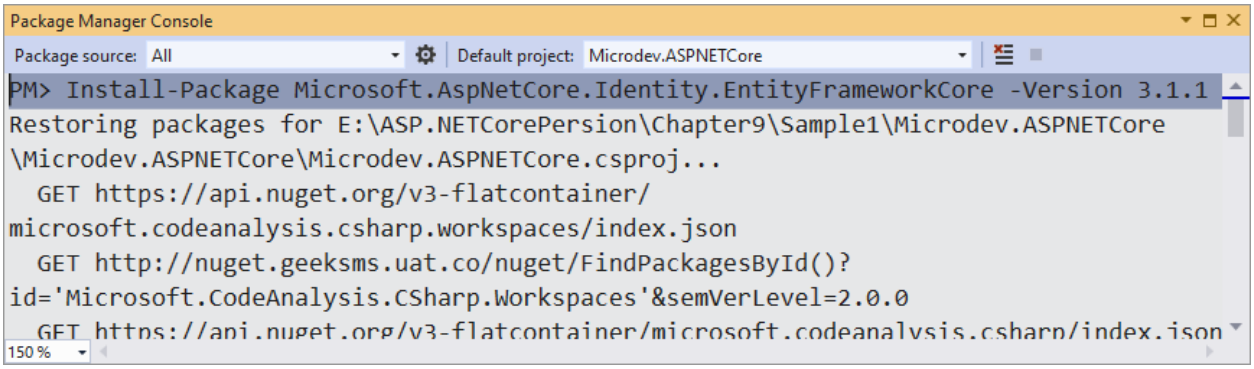
مرحله بعدی اضافه کردن کلاس MicrodevDbContext در فولدر Models است. اما قبل از آن باید پکیج‌های پایین را نصب کنید.

Package Manager را از مســـــیـر Tools> NuGet Package Manager> Console Manager باز کنید و سپس روبروی دستور PM> دستورات زیر را وارد نمایید:

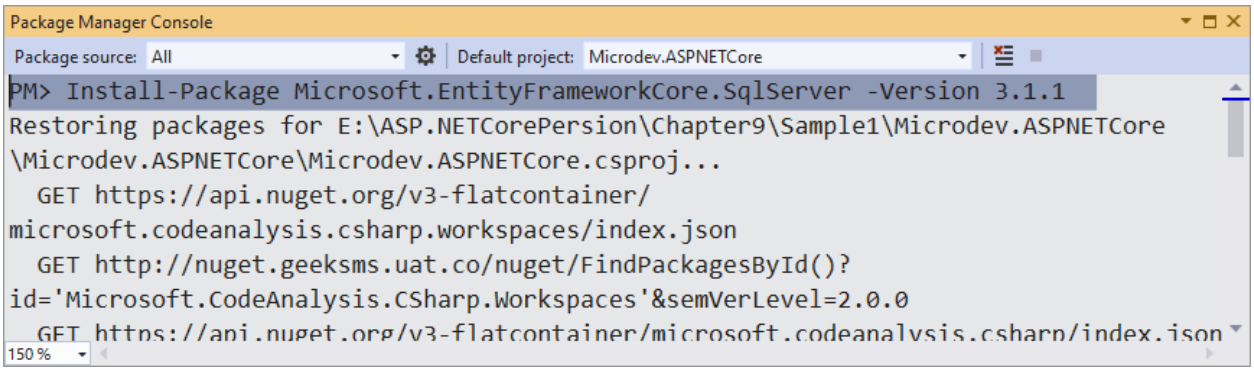
Install-Package Microsoft.EntityFrameworkCore -Version 3.1.1



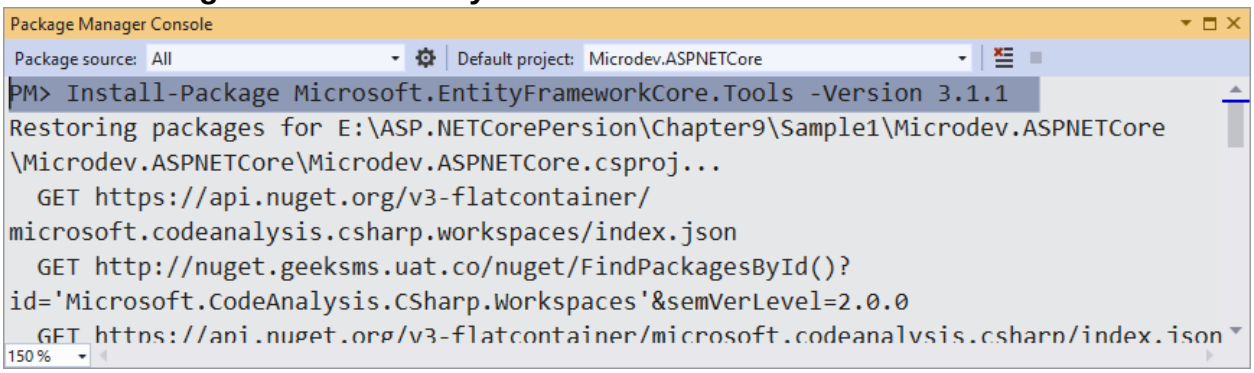
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore -Version 3.1.1



Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 3.1.1



Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.1.1



اضـافه کردن DbContext برای Identity بسـ یار سـاده اسـت. فقط DbContext برنامه شـما باید از IdentityDbContext ارثبری کند.

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

namespace Microdev.ASPNETCore.Models
{
    public class MicrodevDbContext : IdentityDbContext<User>
    {
        public MicrodevDbContext (DbContextOptions<MicrodevDbContext>
        options) : base(options)
        {
        }

        public DbSet<Employee> Employees { get; set; }
        public DbSet<Department> Departments { get; set; }

       protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<Employee>()
            .Property(p => p.Salary).HasColumnType("Decimal(10,2)");
        }
    }

}
```

DbContext باید از IdentityDbContext ارثبری کند.

۳) افزودن appsettings.json

همانطور که در فصـل قبل ذکر شـد، برای ایجاد دیتابیس باید یک فایل appsettings.json داشـته باشـ ید. بنابراین، در فایل appsettings.json پروژه کدهای پایین را اضافه نمایید:

appsettings.json:

```
{
  "ConnectionStrings": {
    "MicrodevConnection": "Server=
    (localdb)\\mssqllocaldb;Database=MicrodevDataBase;Trusted_Connection=Tru
    e;MultipleActiveResultSets=true"
  },

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

افزودن ConnectionString

۴) آپدیت متدهای ConfigureService و Configure

اکنون برای پیکربندیASP.NET Core Identity، باید متدهای ConfigureService و Configure را در کلاس Startup را آپدیت کنید:

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.Services;
using Microsoft.Extensions.Hosting;


namespace Microdev.ASPNETCore
{
    public class Startup
    {
        private readonly IConfiguration _configuration;

        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<EmployeeService>();
            services.AddTransient<TestEmployeeService>();
```

```
    services.AddTransient<Func<EnviromentServiceType,
    IEmployeeService>>(serviceProvider => key =>
    {
        switch (key)
        {
            case EnviromentServiceType.ProductionEmployeeService:
               return
               serviceProvider.GetRequiredService<EmployeeService>();
            case EnviromentServiceType.TestEmployeeService:
               return
               serviceProvider.GetRequiredService<TestEmployeeService
               >();
            default:
              throw new NotImplementedException($"Service of type
              {key} is not implemented.");
        }
    });
    var connection =
    _configuration.GetConnectionString("MicrodevConnection");
    services.AddDbContext<MicrodevDbContext>(
        options => options.UseSqlServer(connection)
    );
    services.AddIdentity<User, IdentityRole>(cfg =>
    {
        cfg.User.RequireUniqueEmail = true;
    }).AddEntityFrameworkStores<MicrodevDbContext>();

    services.AddAuthentication();
    services.AddControllersWithViews();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{

    if (env.IsDevelopment())
    {
        app.UseStatusCodePages();
    }
    else
    {
        app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
    }

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();
```

EF از ASP.NET Core Identity استفاده می کند.

Core

این Middleware سیستم Identity و پیکربندی کاربر و انواع Role ها را انجام می‌دهد

پیکربندی Identity برای ذخیره داده در EF Core

حتما این دو Middleware باید بعد از UseRouting قرار گیرد.

```
            app.UseEndpoints(endpoints =>
              {
                  endpoints.MapControllerRoute(
                  endpoints.MapControllerRoute(
                      name: "default",
                      pattern: "{controller=Home}/{action=Index}/{id?}");

                      name: "employee",
                      pattern:
                      "{controller=Employee}/{action=GetAllEmployee}/{id?}");

              });
        }
    }

}
```

همانطور که می‌بینید، تنظیمات فوق به ASP.NET Core Identity می‌گوید از کلاس User اســـتفاده کند و ذخیره مدل Identity باید با استفاده از Entity Framework Core انجام شود.

در این قســـمت می‌توانید گزینه‌های دیگری را نیز تعیین کنید. **به عنوان مثال:** همانطور که می‌بینید می‌توانید بگویید که داشتن ایمیل منحصر به فرد، ضروری است.

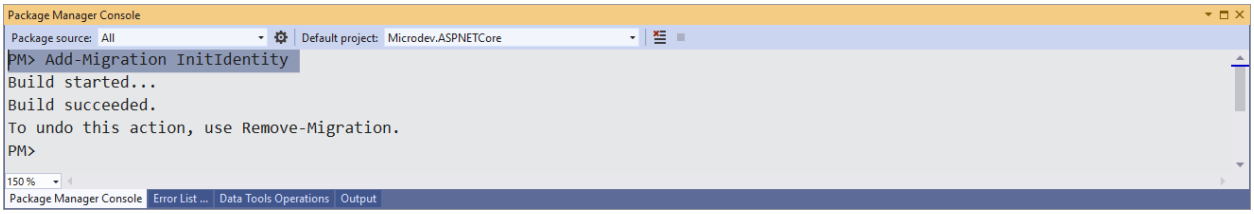
نکته!!

در اینجا گزینه‌های دیگری برای پیکربندی پسورد قوی، نام کاربری، مسیری برای ورود به‌سیستم و موارد دیگر در دسترس است.

۵) دیتابیس را به روز کنید.

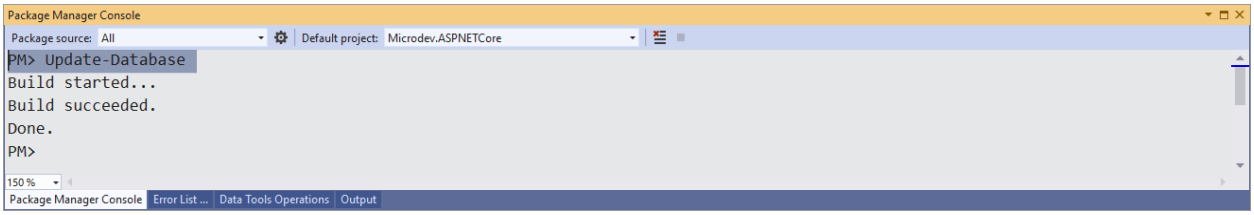
همانطور که می‌دانید، هر زمان که مدل EF Core تغییر کند، باید یک Migration جدید ایجاد و تغییرات را در دیتابیس اعمال نمایید. بنابراین، در این مرحله باید یک Migration جدید به نام InitIdentity اضافه کنید.

```
Add-Migration InitIdentity
```

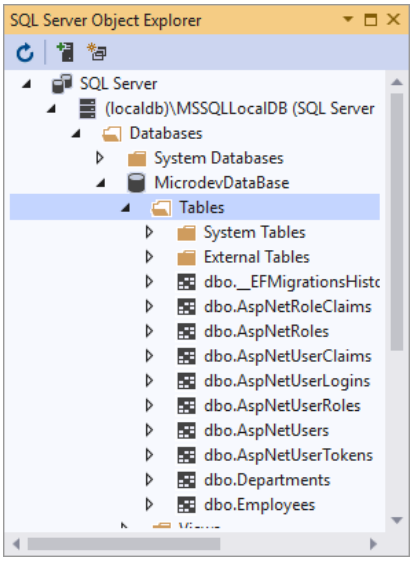


و سپس از کد زیر برای اعمال تغییرات در دیتابیس استفاده نمایید:

```
Update-Database
```



کلاس پایه IdentityDbContext شامل DbSetهای ضروری برای ذخیره کردن User می‌باشد. همانطور که در تصویر زیر می‌بینید، این DbSetها پس از Update-Database به دیتابیس اضافه شده است.



استفاده از Authorization در اپلیکیشن

Authorization فرآیندیست برای تعیین اینکه آیا کاربر می‌تولند عملی را در اپلیکیشن انجام دهدیا خیر؟ Authorization قبل از اجرای اکشن‌متدها اتفاق می‌افتد و از دسترسی کاربران ناشناس به اپلیکیشن جلوگیری می‌کند.

ساده‌ترین روش Authorization استفاده از اتربیوت [Autorize] است. این اتربیوت در بالای کنترلر یا اکشن‌متد اضافه می‌شود و تضمین می‌کند که فرد باید وارد سیستم شده باشد.

```
using System;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.Services;

[Authorize]
```

اعمال [Authorize] به کنترلر

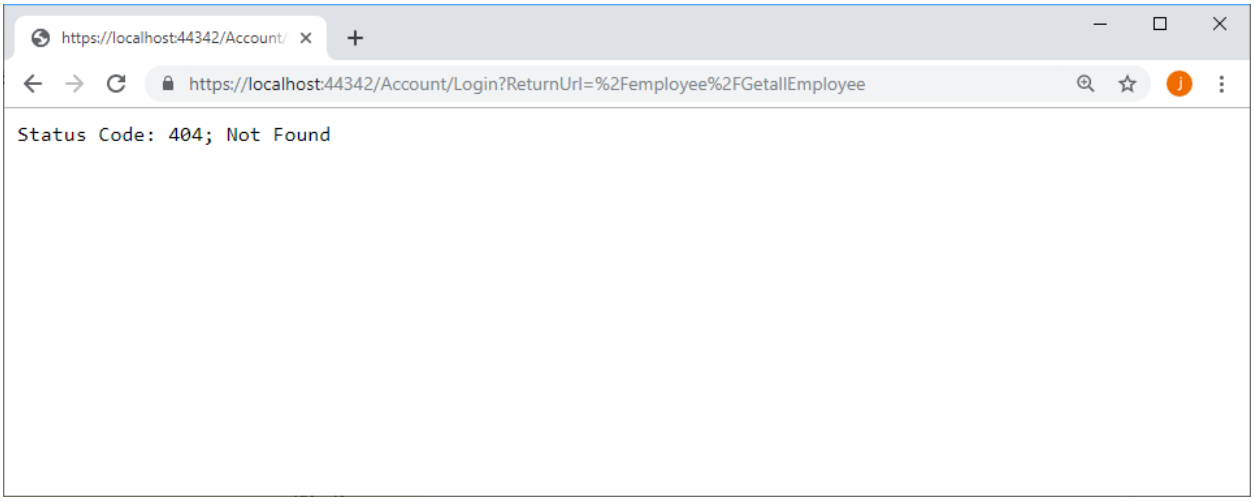
```
namespace Microdev.ASPNETCore.Controllers
{
    public class EmployeeController: Controller
    {
        readonly Func<EnviromentServiceType, IEmployeeService> _service;

        public EmployeeController(Func<EnviromentServiceType,
        IEmployeeService> enviromentServiceType)
        {
            _service = enviromentServiceType;
        }

        public IActionResult CreateEmployee()
        {
            var service =
           _service(EnviromentServiceType.TestEmployeeService);
            var model = service.CreateEmployee();
            return View(model);
        }

        public IActionResult GetEmployee(int employeeId)
        {
           var service = _service(EnviromentServiceType.TestEmployeeService);
            var model =service.GetEmployee(employeeId);
            return View(model);
        }

        public IActionResult GetAllEmployee()
        {
           var service = _service(EnviromentServiceType.TestEmployeeService);
            var model = service.GetAllEmployee();
            return View(model);


        }

    }
}
```

هم EmployeeService و هم TestEmployeeService هربار به سازنده تزریق می‌شوند.

هرگونه اکشــن‌متد یا کنترلری که اتربیوت [Authorize] داشــته باشــد، فقط توســط یک کاربر معتبر قابل اجرا است.

بیایید اپلیکیشن را با هم تست کنیم، لطفا برنامه را اجرا و وارد آدرس زیر شوید.

https://localhost:44342/Employee/GetAllEmployee



همانطور که می‌بینید، به طور خودکار به صفحه ورود به سیستم هدایت می‌شوید، زیرا کنترلر Employee دارای اتربیوت [Authorize] است و شما به عنوان به یک کاربر غیرمجاز، اجازه دسترسی به کنترلر را ندارید.

پس کاربران برای دسترسی به اپلیکیشن، باید وارد سیستم شوند. مرحله بعدی پیاده‌سازی یک کنترلر است که درخواست‌های لاگین به سیستم را دریافت و کاربر را تأیید کند.

نکته!!

دیدن Status Code در تصویر بالا به دلیل استفاده از StatusCodePagesMiddleware در متد Configure است.

لطفاً در فولدر Controllers یک فایل کنترلر جدید به نام AccountController.cs ایجاد و سپس یک متد جدید به نام Login در آن اضافه نمایید:

AccountController.cs:

```
using Microsoft.AspNetCore.Mvc;

namespace Microdev.ASPNETCore.Controllers
{
    public class AccountController: Controller
    {
        public AccountController()
        {

        }
        public IActionResult Login()
        {
```

```
            return View();
        }
    }

}
```

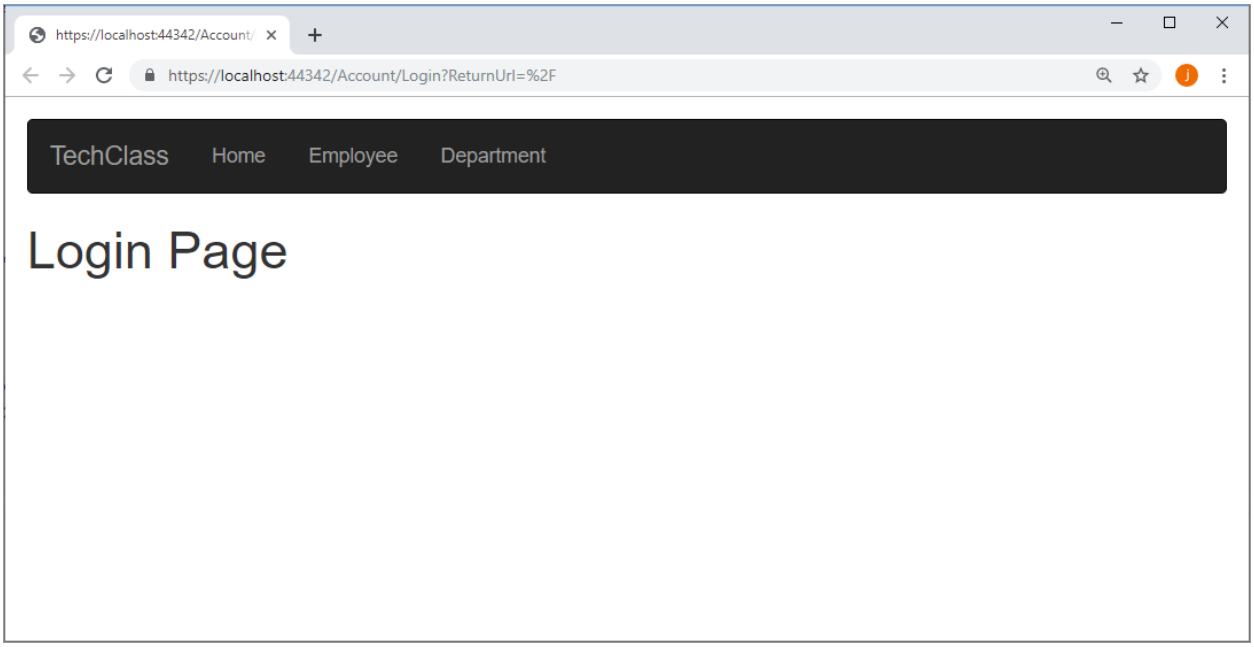
حالا جهت اعتبارسنجی کاربر، باید در فولدر Views / Account یک View جدید با نام Login.cshtml اضافه نمایید.

Login.cshtml:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Login</title>
</head>
<body>
    <h1>Login Page</h1>
</body>

</html>
```

حالا اپلیکیشن را اجرا و وارد مسیر زیر شوید.

https://localhost:44342/Employee/GetAllEmployee



از آنجا که شما در حال حاضر یک کاربر غیرمجاز هستید و سعی دارید وارد سیستم شوید، بنابراین به صفحه ورود به سیستم هدایت خواهید شد.

افزودن فرم لاگین

بیایید فرم login کاربر را با هم اصلاح کنیم:

۱) در فولدر ViewModels یک ViewModel جدید به نام LoginModel اضافه کنید:

```
using System.ComponentModel.DataAnnotations;

namespace Microdev.ASPNETCore.ViewModels
{
    public class LoginViewModel
    {
        [Required]
        public string UserName { get; set; }
        [Required]
        public string Password { get; set; }
        public bool RememberMe { get; set; }
        public string ReturnUrl { get; set; }
    }


}
```

۲) با فرض اینکه ما یک حساب کاربری داریم، مرحله بعدی انجام مرحله Authentication، با استفاده از کلاس SignInManager<User> است. بنابراین همانند کد پایین در AccountController، فیلدهایی برای SignInManager و UserManager اضافه و سپس یک SignInManager <User> و UserManager <User> را به سازنده تزریق نمایید:

ویرایش AccountController:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.Models;
using System.Threading.Tasks;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Controllers
{
  public class AccountController : Controller
  {
    private readonly SignInManager<User> _signInManager;
```

تأیید نام کاربری و رمز عبور با استفاده از SignInManager که به حساب کاربری کنترل تزریق شده است انجام می‌شود.

```
        private readonly UserManager<User> _userManager;

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


        }


    //...

}


}
```

کلاس UserManager برای مدیریت کاربران در ASP.NET Identity Core استفاده می شود.

۳) و سپس دو متد جدید دیگر با نام‌های Login و Logout اضافه کنید.

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.ViewModels;

namespace Microdev.ASPNETCore.Controllers
{
    public class AccountController : Controller
    {
        private readonly SignInManager<User> _signInManager;
        private readonly UserManager<User> _userManager;

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

        public IActionResult Login()
        {
            if (this.User.Identity.IsAuthenticated)
            {
                return RedirectToAction("Index", "Home");
            }
```

```
        return View();
    }

    [HttpPost, ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginViewModel model)
    {

       if (ModelState.IsValid)
        {
            var result = await
            _signInManager.PasswordSignInAsync(model.UserName,
              model.Password,
              model.RememberMe,
              false);

            if (result.Succeeded)
            {
                if (Url.IsLocalUrl(model.ReturnUrl))
                {
                    return Redirect(model.ReturnUrl);
                }
                else
                {
                    return RedirectToAction("Index", "Home");
                }
            }
        }

        ModelState.AddModelError("", "Failed to login");

        return View();

    }

    [HttpGet]
    public async Task<IActionResult> Logout()
    {
        await _signInManager.SignOutAsync();
        return RedirectToAction("Index", "Home");
    }

  }
}
```

اتربیوت ValidateAntiForgeryTokenبرای محافظت در برابر Requestهای cross-site می‌باشد.

متد PasswordSignIn احراز هویت را انجام می دهد.

نتیجه متد PasswordSignInAsync یک شیء SignInResult است که یک property بولین را برای شما تعریف می‌کند و نشان می دهد روند احراز هویت موفق بوده است یا خیر.

ReturnUrl نشانی اینترنتی را مشخص می‌کند که کاربر پس از تأیید اعتبار باید به آن فرستاده شود.

متد SignOutAsync هر session موجود را که کاربر دارد کنسل می‌کند.

۴) حالا باید محتوای Login.cshtml را در فولدر Views / Account تغییر دهید.

```
@model LoginViewModel
```

```
<div class="row">
    <div class="col-md-4 col-md-offset-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly"></div>
            <div class="form-group">
                <label asp-for="UserName">Username</label>
                <input asp-for="UserName" class="form-control" />
                 <span      asp-validation-for="UserName"      class="text-
                 warning"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password">Password</label>
                <input   asp-for="Password"   type="password"   class="form-
                control" />
                <span      asp-validation-for="Password"      class="text-
                warning"></span>
            </div>
            <div class="form-group">
            <input  asp-for="RememberMe"  type="checkbox"  class="checkbox-
             inline"/>
                <label asp-for="RememberMe">Remember Me?</label>
                  <span    asp-validation-for="RememberMe"    class="text-
                warning"></span>
            </div>
            <div class="form-group">
              <input type="submit" value="Login" class="btn btn-success" />
            </div>
        </form>
    </div>

</div>
```

۵) قبل از نمایش Login یا logout برای یک کاربر معتبر، فایل Views/Shared/_Layout.cshtml را آپدیت نمایید:

```
<html>
<head>
    <meta name="viewport" content="width=device-width" />
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css">
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.0/jquery.min.js"></scri
pt>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></s
cript>
    <title>@ViewBag.Title</title>

</head>
```

```
<body class="panel-body">
    <nav class="navbar navbar-inverse">
        <div class="container-fluid">
            <div class="navbar-header">
                <a class="navbar-brand" href="#">Microdev</a>
            </div>
            <ul class="nav navbar-nav">
                <li><a href="#">Home</a></li>
                <li><a href="#">Employee</a></li>
                <li><a href="#">Department</a></li>
                @if (User.Identity.IsAuthenticated)
                {
                  <li><a asp-controller="Account" asp-
                action="Logout">Logout</a></li>
                }
                else
                {
                <li><a asp-controller="Account" asp-
                action="Login">Login</a></li>
                }
            </ul>
        </div>
    </nav>

    @RenderBody()


    @RenderSection("Footer", required: false)

</body>

</html>
```

Seeding داده‌های کاربر

همانطور که قبلاً گفتم، اضـــافه کردن داده‌ها در اجرای اولین بار اپلیکیشـــن را، Seeding دیتابیس می‌گویند. بنابراین، در این مرحله باید جدول User را مقداردهی اولیه کنیم.

خبرهای خوبی برای شـما دارم، می‌خواهم Seeding داده‌های کاربر را با روش ی دیگر پیاده‌سـازی کنم. پس در فولدر Models یک کلاس جدید به نام SeedData اضافه کنید.

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;

namespace Microdev.ASPNETCore.Models
```

```
{
    public class SeedData
    {
        private readonly MicrodevDbContext _ctx;
        private readonly UserManager<User> _userManager;

        public SeedData(MicrodevDbContext ctx,
                    UserManager<User> userManager)
        {
            _ctx = ctx;
            _userManager = userManager;
        }

        public async Task Seed()
        {
            _ctx.Database.EnsureCreated();

              var user = await
             _userManager.FindByEmailAsync("Info@Microdev.com");

            if (user == null)
            {
                user = new User()
                {
                    FirstName = "Jennifer",
                    LastName = "Lerman",
                    UserName = "Info@Microdev.com",
                    Email = "Info@Microdev.com"
                };

                 var result = await _userManager.CreateAsync(user,
                 "P@ssw0rd!");
                if (result != IdentityResult.Success)
                {
                 throw new InvalidOperationException("Failed to create
                 default user");
                }
            }

        }
    }
}
```

قبل از اجرای اپلیکیشن باید یک تغییر کوچک در **Startup.cs** ایجاد و کلاس **SeedData** را رجیستر کنید:

```
using System;
using Microsoft.AspNetCore.Builder;
```

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microdev.ASPNETCore.Models;
using Microdev.ASPNETCore.Services;
using Microsoft.Extensions.Hosting;

namespace Microdev.ASPNETCore
{
    public class Startup
    {
        private readonly IConfiguration _configuration;

        public Startup(IConfiguration configuration)
        {
            _configuration = configuration;
        }
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<EmployeeService>();
            services.AddTransient<TestEmployeeService>();
            services.AddTransient<SeedData>();
            services.AddTransient<Func<EnviromentServiceType,
           IEmployeeService>>(serviceProvider => key =>
            {
                switch (key)
                {
                    case EnviromentServiceType.ProductionEmployeeService:
                       return
                       serviceProvider.GetRequiredService<EmployeeService>();
                    case EnviromentServiceType.TestEmployeeService:
                       return
                       serviceProvider.GetRequiredService<TestEmployeeService
                       >();
                    default:
                      throw new NotImplementedException($"Service of type
                      {key} is not implemented.");
                }
            });
             var connection =
             _configuration.GetConnectionString("MicrodevConnection");
            services.AddDbContext<MicrodevDbContext>(
                options => options.UseSqlServer(connection)
            );
            services.AddIdentity<User, IdentityRole>(cfg =>
            {
                cfg.User.RequireUniqueEmail = true;
            }).AddEntityFrameworkStores<MicrodevDbContext>();
```

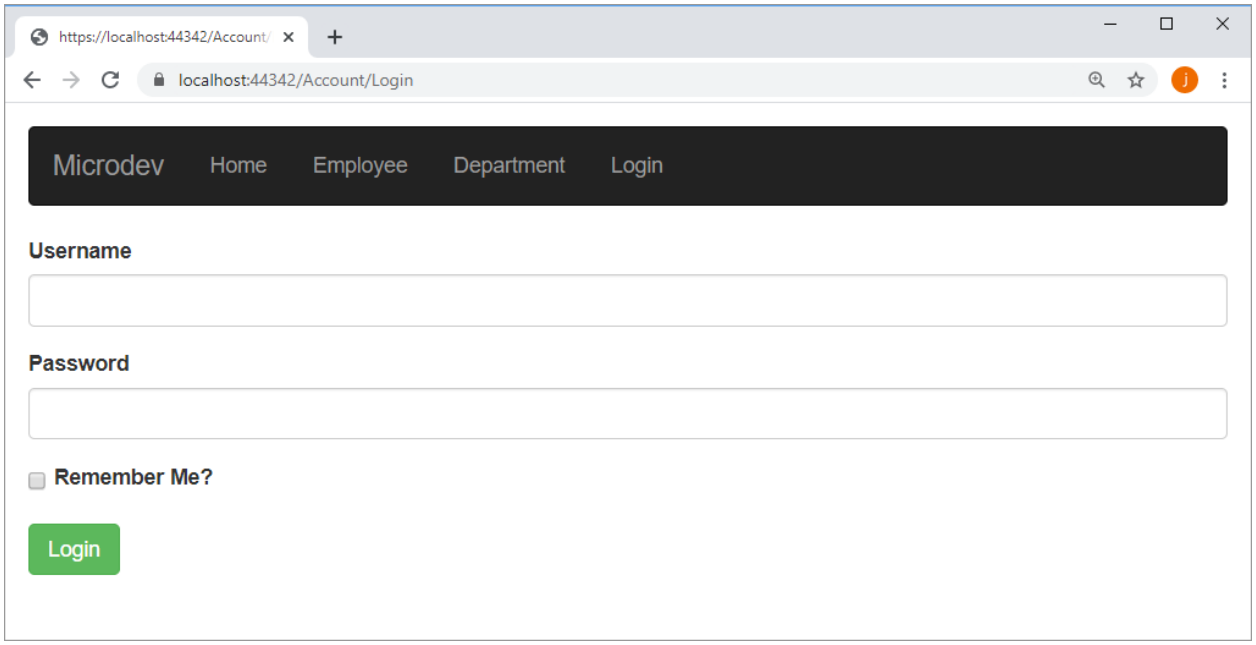
رجیستر سرویس SeedData

```
            services.AddAuthentication();
            services.AddControllersWithViews();

        }

         public void Configure(IApplicationBuilder app, IWebHostEnvironment
         env)
        {

            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Home/error/{0}");
            }

            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

           // Seed the database
              using (var scope = app.ApplicationServices.CreateScope())
              {
                  var seeder = scope.ServiceProvider.GetService<SeedData>();
                  seeder.Seed().Wait();
              }

             app.UseEndpoints(endpoints =>
              {
                  endpoints.MapControllerRoute(
                  endpoints.MapControllerRoute(
                      name: "default",
                      pattern: "{controller=Home}/{action=Index}/{id?}");

                      name: "employee",
                      pattern:
                      "{controller=Employee}/{action=GetAllEmployee}/{id?}");

              });
        }
    }

}
```

عملیات Seed دیتابیس

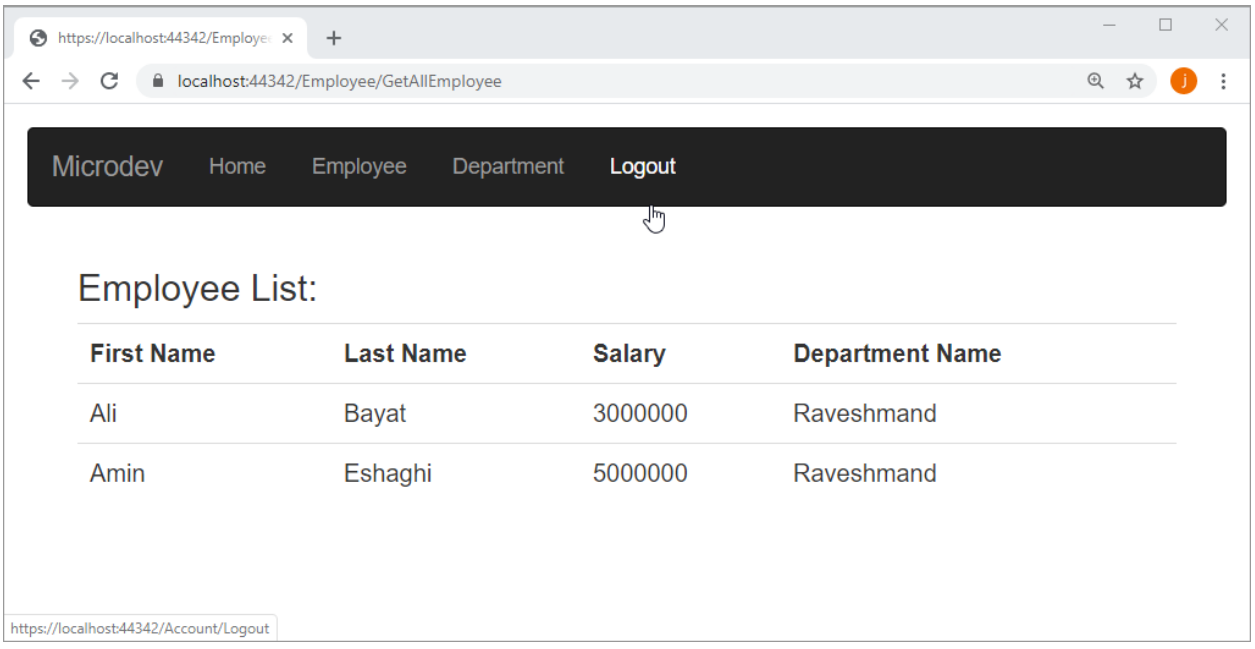
لطفاً اپلیکیشن را اجرا و بر روی **Login** در **Navbar** کلیک کنید:

اکنون نام کاربری Info@Microdev.com و پسورد P@ssw0rd! را وارد کنید و بر روی Login کلیک کنید.



نکته!!

پس از ورود به اپلیکیشن می‌توانید با کلیک بر روی Log Off از سیستم خارج شوید.



مسیر پروژه نمونه انجام شده در **Github**:

https://github.com/ZahraBayatgh/PracticalASP.NETCore/tree/master/src/Chapter9/Sample1

Interview Questions

To prepare for a job interview, please answer the following questions:

Q1: What is Authentication?

Q2: What is Authorization?

Q3: What is ASP.NET Identity?

Q4: How to configure ASP.NET identity?

Q5: Can you briefly explain how Authentication works?

Q6: What are the advantages of using Authentication?

Q7: How to configure Authentication in ASP.NET?

Q8: How to implement Forms authentication in ASP.Net Core?

Q9: How to use Authorization in the application?

Q10: What are the authorization methods?

Quiz

Q1: What is Authentication?

1. The process of determining what you're allowed to do.
2. The process of determining who you are.
3. Authentication is a system for adding login functionality to applications.
4. All of the above

Q2: _____ is the process of verifying if the authenticated user has the sufficient rights to do certain things.

5. Identity
6. Authorization
7. Authentication
8. Both 1 and 2

Q3: _____provides mechanisms for managing users?

1. Identity
2. UserManager .
3. ASP.NET Identity
4. SignInManager

Q4: Managing users and authenticating users via_____?

1. SignInManager , SignInResultManager
2. UserManager , SignInManager
3. UserManager , PasswordSignInManager
4. SignInManager , SignOutManager

Q5: _____ contains information about the users in the application.

1. IdentityDbContext
2. Identity
3. IdentityUser
4. All of the above

Q6: The simplest authorization method is to use the _____meta decorator.?

1. [Authorization]
2. [Authorize].
3. [Auth]
4. Both 1 and 2

Q7: Any action or controller that has the _____ attribute applied in this way can be executed only by an authenticated user.

1. [Authorization]

2. [Authorize]
3. [Authentication]
4. [Authenticat]

Q8: PasswordSignInAsync method is a _____object.

1. SignInResult
2. ManageInResult
3. PasswordInResult
4. Both 1 and 2

Q9: The_____ method cancels any existing session that the user has.

1. SignInResultAsync
2. SignOutAsync
3. PasswordSignInAsync
4. SignInAsync

Q10: _____specifies the URL that the user should be sent back to once they have been authenticated.

1. Url
2. ReturnUrl
3. SignOut
4. Both 1 and 2

Answer

1-Correct Answer: The process of determining who you are

2-Correct Answer: Authorization

3-Correct Answer: ASP.NET Identity

4-Correct Answer: UserManager , SignInManager

5-Correct Answer: IdentityUser

6-Correct Answer: [Authorize]

7-Correct Answer: [Authorize]

8-Correct Answer: SignInResult

9-Correct Answer: SignOutAsync

10-Correct Answer: ReturnUrl

خلاصه فصل

- ✓ **Authentication فرایندی است برای تعیین اینکه شما چه کسی هستید.**
- ✓ **Authentication تعیین کارهایی است که شما مجاز به انجام آن هستید.**
- ✓ **Authentication در ASP.NET Core توسط AuthenticationMiddleware ارائه شده است.**
- ✓ **Authentication به اپلیکیشن‌ها اجازه می‌دهد تا یک کاربر خاص را شناسایی کنند.**
- ✓ **ASP.NET Core Identity سیستمی است برای اضافه کردن قابلیت لاگین به اپلیکیشن‌ها که به شما کمک می‌کند ویژگی‌های امنیتی و هویتی را به برنامه خود اضافه کنید.**
- ✓ **ASP.NET Identity از طریق کلاس‌های UserManager و SignInManager، مکانیزم‌هایی را برای مدیریت و تأیید هویت کاربران فراهم می‌کند.**
- ✓ **اتریبیوت [Authorize] در واقع یک بررسی احراز هویت را در اینجا انجام می‌دهد و هیچ‌گونه مجوزی را بررسی نخواهد کرد.**
- ✓ **شما می‌توانید از کلاس UserManager<T> برای ایجاد حساب کاربری جدید، لود اطلاعات از دیتابیس و تغییر رمزهای عبور خود استفاده کنید.**
- ✓ **SignInManager <T> برای ورود و خروج کاربر از اپلیکیشن استفاده می‌شود.**
- ✓ **متد PasswordSignIn فرآیند احراز هویت را انجام می‌دهد.**
- ✓ **متد SignOutAsync هر گونه Session کاربر را کنسل می‌کند.**
- ✓ **EF Core DbContext شما می‌تواند با ارث‌بری از IdentityDbContext <TUser> از Identity بهرمند شود.**
- ✓ **کلاس پایه IdentityDbContext شامل DbSet <T> لازم برای ذخیره کاربران با استفاده از EF Core است.**

کتاب‌های نوشته شده: