



ANY
Server

ASP.NET
Core

ANY
Client



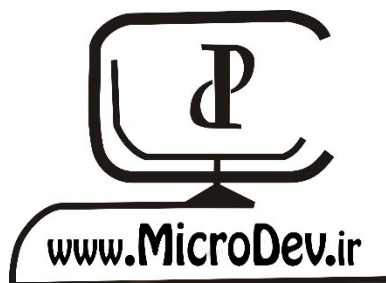
ASP.NET Core

به زبان ساده

مولفین:

زهرا بیات قلی لاله

علی بیات قلی لاله



ASP.NET Core

به زبان ساده

زهرا بیات قلی لاله

علی بیات قلی لاله

ASP.NET Core

به زبان ساده

مؤلفین : زهرا بیات قلی لاله – علی بیات قلی لاله

طراح جلد : زهرا بیات قلی لاله

ناشر : سایت کتاب سبز

مشخصات ظاهری : ۱۲۸ص

سال انتشار: اسفند ۹۷

قیمت : رایگان

فهرست

فصل اول: مقدمات ASP.NET Core

ASP.NET Core چیست؟

مزایای ASP.NET Core

ایجاد اولین اپلیکیشن ASP.NET Core

ساختار پروژه

کلاس Program

کلاس Startup

Middleware در ASP.NET Core چیست؟

قابلیت‌های Middleware :

موارد استفاده از Middleware

نحوه‌ی عملکرد Middleware

کاربرد متد ConfigureServices در کلاس Startup

چطور Requestها با Middleware، Handle می‌شوند؟

کاربرد متد Configure در کلاس Startup

wwwroot

فصل دوم: مدیریت Exceptionها

مدیریت Exceptionها

مدیریت Exception ها و محیط های اجرایی

Exception در محیط اجرایی

تعریف Environment

مدیریت Status code ها

فصل سوم : مقدمات EF Core

Routing چیست؟

مقدمه ای در مورد EF Core

اضافه کردن دیتابیس به پروژه

DataAnnotation چیست؟

تعریف DbContext

ConnectionString چیست؟

مقدمه ای در مورد Dependency Injection

مزایای DI

طول عمر یک سرویس ایجاد شده توسط DI

انواع طول عمر

فصل چهارم : ایجاد دیتابیس

ریجستر DbContext از طریق DI

نکاتی در مورد appsettings.json و IConfiguration service

Entity Framework Migration چیست؟

ایجاد Migration

نکاتی در مورد EF Core:

فصل پنجم: عملیات CRUD

مفهوم Separation Of Concerns

شروع عملیات CRUD

Controller چیست؟

Layout چیست؟

نمایش لیست کارمندان

Razor چیست؟

Tag Helper چیست؟

_ViewImports و _ViewStart چیست؟

جزئیات کارمند

ایجاد URL برای Action

Model Binding چیست؟

Validation بر روی Server

ValidateAntiForgeryToken چیست؟

متد حذف کارمند

درج کارمند جدید

تمام کدهای EmployeeController

ASP.NET Identity چیست؟

فصل ششم : ASP.NET Identity

Authentication و Authorization چیست؟

مدیریت User

ایجاد صفحه Login

Claim چیست؟

Principal چیست؟

ایجاد یک Principal

Claim برای کاربر

افزودن Claim Check

ایجاد Logout

ثبت نام کاربر

تقدیم به:

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

با تشکر:

از شرکت مدیریت رو شمند و همکار خوبم جناب آقای مهندس امین مژگانی، که در به سرانجام رساندن این کتاب همراه ما بودند بسیار سپاسگزاریم.

این کتاب مناسب کسانیست که می‌خواهند با یک عملکرد سریع، پاسخی برای بازار کار باشند.

داده‌ها، در همه جا با سرعت باورنکردنی در حال رشد هستند و میلیون‌ها برنامه کاربردی از این داده‌ها استفاده می‌کنند. اما خبر خوب برای شما این است، که هنوز میلیون‌ها برنامه کاربردی، تولید نشده و مهارت شما برای این شغل بسیار موردنیاز است.

متاسفانه فشار برای تولید و توسعه نرم‌افزار، بسیار بالاست و همین می‌تواند شروعی، برای حرکت دوباره باشد.

اما چطور این حرکت دوباره را شروع کنیم؟

برای این حرکت دوباره، باید به بازار کار نگاهی بیندازیم. همه چیز به سمت جدیدترین و بروزترین تکنولوژی‌ها در حرکت است. پس بیایید برای این حرکت دوباره، ما هم با بروزترین تکنولوژی‌ها شروع کنیم.

ماموریت من در زندگی، یادگیری جدیدترین تکنولوژی‌ها و آموزش آن به سایرین است. دوست دارم دانسته‌هایم را به اشتراک بگذارم و راه چندساله‌ام را برای دیگران آسان‌تر کنم. در این کتاب سعی شده، ایده‌ی یادگیری و تولید کدهای عالی با روشی کاملاً ساده و روان بیان شود.

تمرکز این کتاب، ساخت برنامه‌ای کاربردی و تنظیم عملکرد شما در برنامه‌نویسی است.

موضوع این کتاب، فریم ورک ASP.NET Core و هدف این کتاب، آموزش مفاهیم اساسی برای تولید یک وب اپلیکیشن است.

در این کتاب سعی شده، از جنبه‌های متفاوت، نکات و تکنیک‌هایی بیان، و گاهی با یک مثال، همه چیز ملموس‌تر شود.

شما خیلی سریع دست به کد خواهید شد و لذت برنامه‌نویسی، با رویکردهای جدید را، حس خواهید کرد.

بیایید با هم در **قلب** ASP.NET Core شیرجه بزنیم.

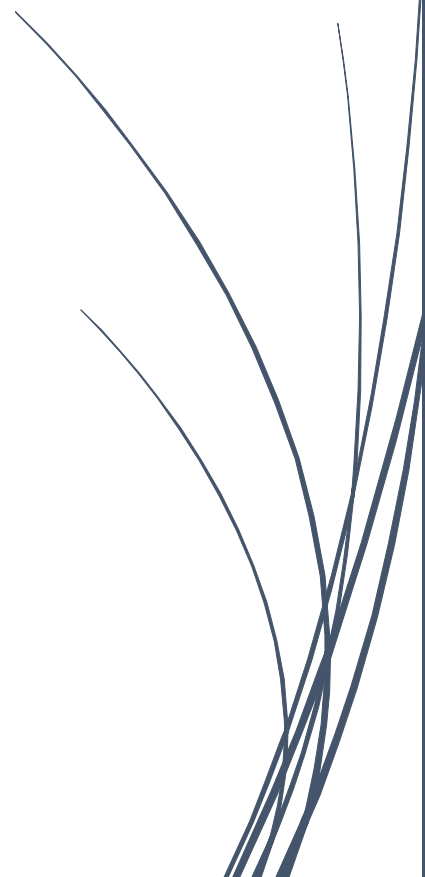
من برای این شروع دوباره بسیار هیجان‌زده‌ام.

فصل اول: مقدمات ASP.NET Core

➤ ASP.NET Core چیست؟

➤ ایجاد اولین اپلیکیشن ASP.NET Core

➤ ASP.NET Core در Middleware چیست؟



ASP.NET Core چیست؟

این روزها، وب اپلیکیشن‌ها، در همه جا دیده می‌شوند و انتظار می‌رود که این اپلیکیشن‌ها، بی‌نهایت Scalable، قابل توسعه در Cloud و دارای Performance بالا باشند.

خبر خوبی که می‌توانم به شما دهم، این است که ASP.NET Core دقیقاً برای این نیازمندی‌ها طراحی شده است.

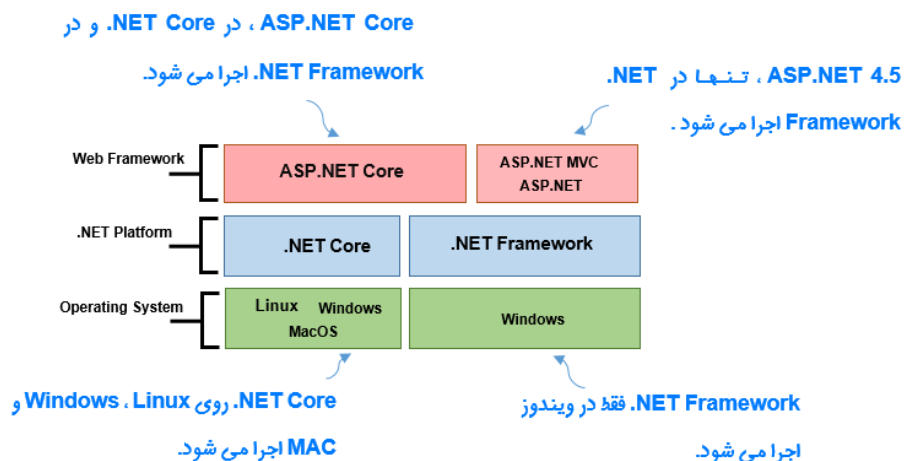
نکته بسیار مهم و جذاب ماجرای .NET Core این است که، این تکنولوژی می‌تواند هم در ویندوز، هم لینوکس و هم سیستم‌عامل اپل اجرا شود.

اینکه شما به یک وب‌سایت ساده نیاز دارید یا یک وب اپلیکیشن E-Commerce پیچیده، همه با این تکنولوژی قابل انجام است.

ASP.NET Core آخرین دستاورد ASP.NET می‌باشد و با هدف Highly Modular و Little Coupling، طراحی شده است. با این روند برنامه شما با ماژولاریتی بالا و کمترین وابستگی ایجاد می‌شود.

اگر قبلاً با ASP.NET MVC5 کار کرده باشید، می‌توان گفت که ASP.NET Core ایده‌ی خوبی برای شروع کار شماست.

ASP.NET Core کاملاً Open Source است و شما می‌توانید در GitHub آن را دانلود نمایید. علاوه بر این، قابل دسترس در Windows و Mac و Linux هم می‌باشد، که این دو مورد می‌تواند یک تضمین عالی برای ادامه‌ی راه ما باشد.



نکته خیلی مهم، در مورد ASP.NET Core :

کدهای اپلیکیشن در ASP.NET Core، در هر Hostی که باشد، می تواند به درخواست HTTP پاسخ دهد و اپلیکیشن شما در هر سروری که بتواند به Requestهای HTTP دسترسی داشته باشد، اجرا خواهد شد. این یعنی: شما دیگر به IIS وابسته نیستید.

به طور کلی Handle کردن درخواستهای HTTP به دو دسته مهم تقسیم می شوند:

- ۱) درخواست از یک Browser، که HTML برمی گرداند. این برای ASP.NET به عنوان Web UI است.
- ۲) یک درخواست از نرم افزار که داده ها را برمی گرداند. این برای ASP.NET Core به عنوان Web API یا Webservice است.

یک موضوع جالب:

Asp.net core یک بار دیگر:

- از اول نوشته شده است.
- ماژولار است.
- Multi-Platform است.
- و Performance بالایی دارد.

مزایای ASP.NET Core

برای دانستن اینکه، چرا مایکروسافت تصمیم گرفت تا یک فریم ورک جدید را ایجاد نماید، باید مزایای آن را دنبال نمایید.

ASP.NET Core امکانات زیادی را با خود به ارمغان آورده است:

- "Pipeline" Middleware برای تعریف رفتارهای اپلیکیشن شما.
- پشتیبانی توکار از Dependency Injection.
- ترکیب (MVC) UI و ساختار (Web API) API.
- سیستم پیکربندی بسیار گسترده.
- قابل Scalable شدن برای پلتفرم های Cloud، با استفاده از Asynchronous Programming.

شاید نسبت به بعضی از این واژگان بیگانه باشید.

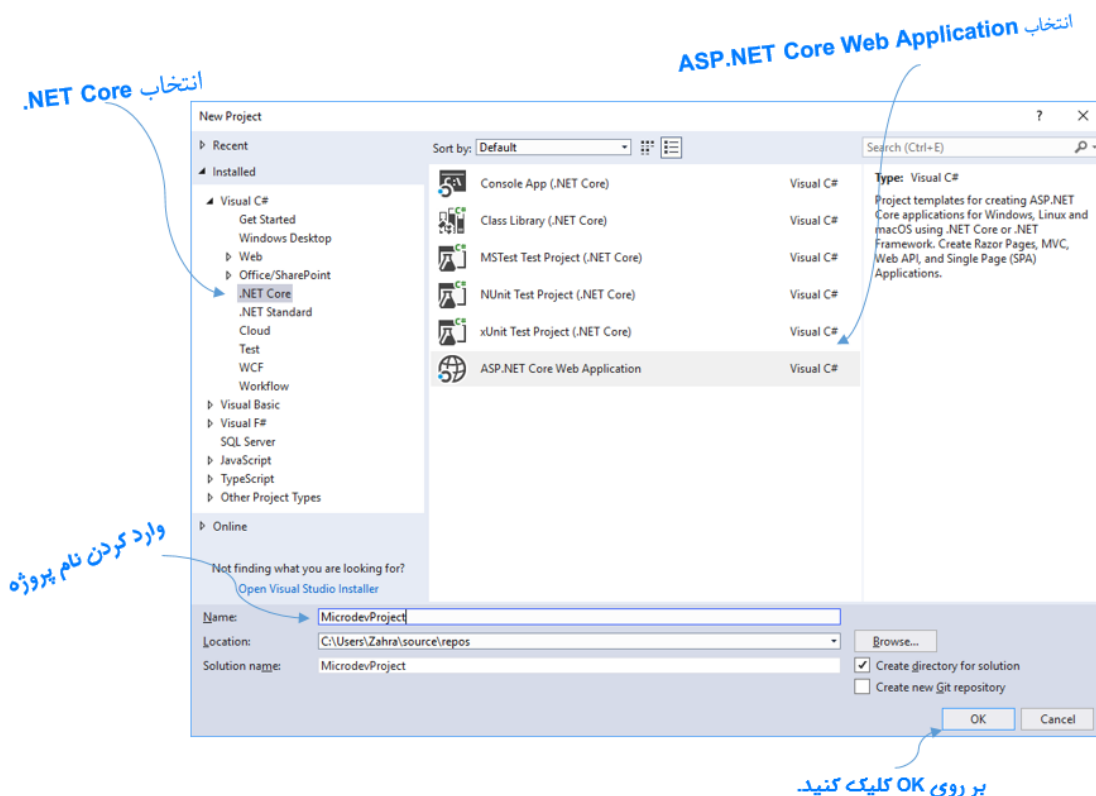
نگران نباشید!! در طول آموزش این مفاهیم توضیح داده شده و مدام تکرار می شود.

ایجاد اولین اپلیکیشن ASP.NET Core

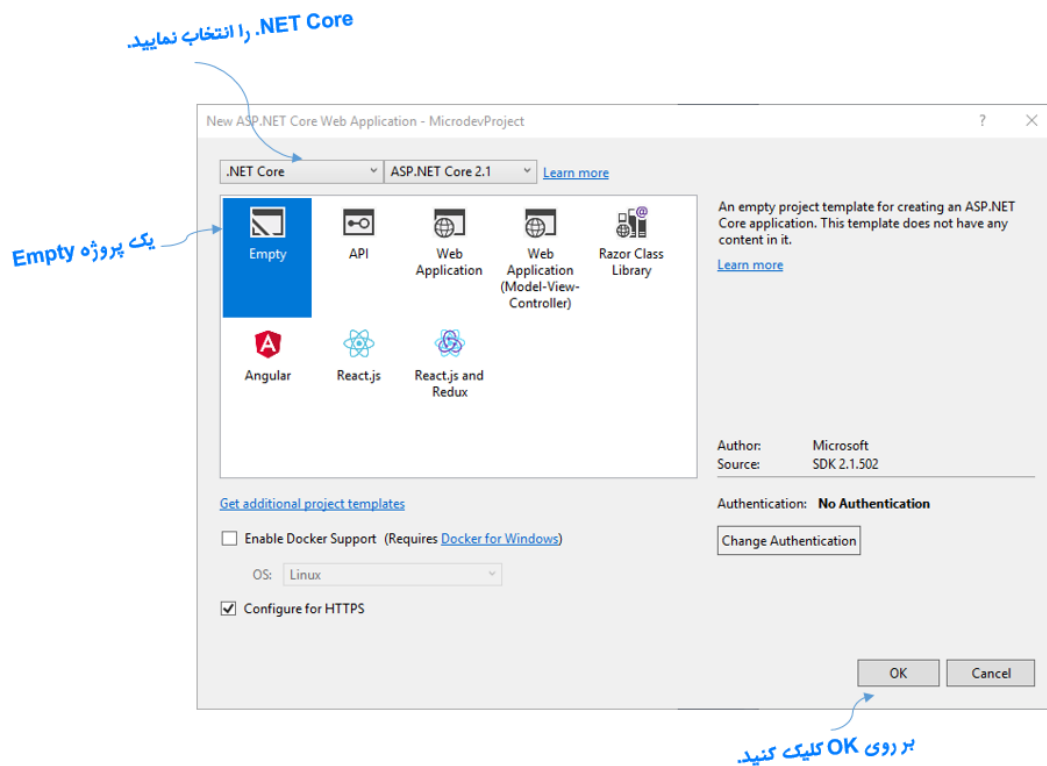
ASP.NET Core باعث می شود برنامه های کاربردی وب، سریع تر، آسان تر، راحت تر و امن تر ایجاد شوند.

پس بیایید ما هم از این امکانات استفاده کنیم و یک پروژه کوچک را با هم شروع کنیم.

- ویژوال استودیو را باز کنید و وارد منوی **File→New→Project** شوید.
- از پنل سمت چپ **.NET Core** و سپس **ASP.NET Core Web Application** را انتخاب نمایید.
- نام پروژه، مکان ذخیره سازی و نام **Solution** (اختیاری) را وارد و بر روی **OK** کلیک نمایید.



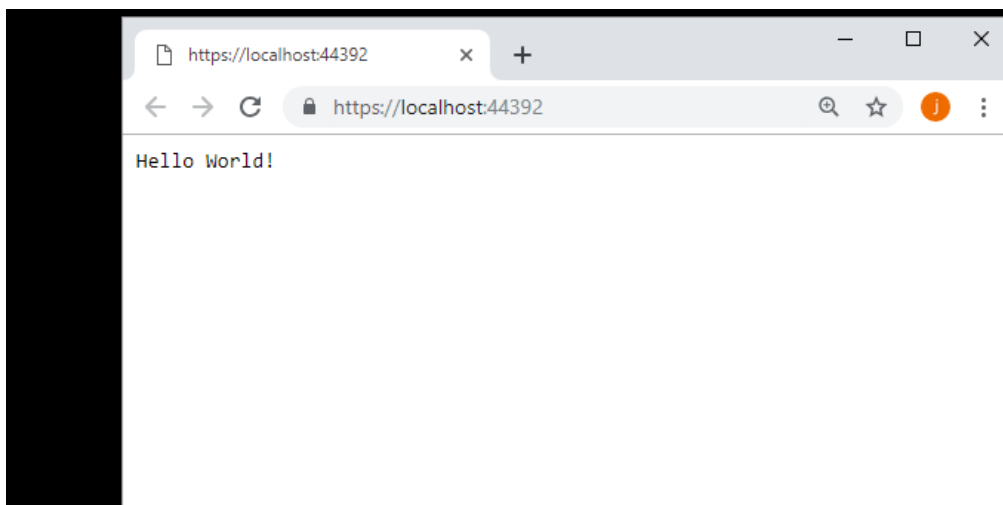
چون می خواهیم مباحث **ASP.Net** را از پایه شروع کنم، پس در کادر بعد، گزینه ی **Empty** را انتخاب نمایید و بر روی **OK** کلیک کنید.



منتظر بمانید تا پروژه‌ی شما ایجاد شود....

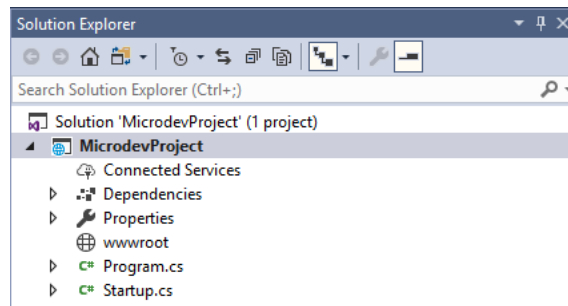
تبریک!! پروژه‌ی شما با موفقیت ایجاد شد.

حالا این پروژه را با زدن کلید (F5) اجرا کنید:



ساختار پروژه

در تصویر زیر ساختار پروژه ایجاد شده را، می‌بینید و در ادامه با مهم‌ترین اجزای این ساختار آشنا خواهید شد.



کلاس Program

تمام اپلیکیشن‌های ASP.NET Core همانند برنامه‌های Console، با یک فایل Program.cs شروع می‌شوند. این فایل یک متد Main دارد و زمانی که اپلیکیشن شما Start شود، این متد اجرا خواهد شد.

نمایی از کلاس Program:

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

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

برای توضیح روند عملکرد این کلاس، می‌خواهم کمی کدهای آن را ساده‌تر بنویسم.

```
public class Program
{
    public static void Main(string[] args)
    {
        IWebHostBuilder webHostBuilder = WebHost.CreateDefaultBuilder(args);
        webHostBuilder.UseStartup<Startup>();
        IWebHost webHost = webHostBuilder.Build();
        webHost.Run();
    }
}
```


هدف این متد ایجاد یک **IWebHost** و صدا زدن متد **Run** آن می باشد. ما در کدهای بالا از **IWebHostBuilder** برای ایجاد یک **IWebHost** استفاده کرده ایم.

متد **CreateDefaultBuilder** یک نمونه از **IWebHostBuilder** ایجاد کرده و تنظیمات زیر را انجام می دهد.

- پیکربندی سرور **Kestrel** که وب سرور پیش فرض است.
- مشخص کردن مسیر محتوای ایستا یا همان **Content root**
- بارگذاری **Configuration**
- پیکربندی **Logging**
- ...

بیشتر تنظیمات اپلیکیشن توسط **WebHostBuilder** انجام می شود، اما بعضی از کارها به فایل **Startup** واگذار خواهند شد، به همین منظور برای معرفی کلاس **Startup** باید از متد **UseStartup** استفاده نمایید و در نهایت با صدا زدن متد **Build** ← **IWebHost** را ایجاد کنید.

IWebHost، هسته‌ی اپلیکیشن شما در **ASP.NET Core** است که شامل **Configuration** و **Kestrel** می باشد. از **Kestrel** برای دریافت **Request**ها و ارسال **Response**ها استفاده می کنیم.

چرا Kestrel؟

Kestrel وب سروری است که به عنوان بخشی از **ASP.NET Core** عرضه شده. این وب سرور به صورت **Cross Platform** است، یعنی بر روی تمامی سیستم‌عامل‌ها پشتیبانی شده و توسط **.NET Core** قابل اجرا می باشد.

کلاس Startup

همانطور که دیدید، کلاس **Program**، برای پیکربندی ساختار اپلیکیشن شما بود، اما پیکربندی برخی از رفتارهای اپلیکیشن شما در کلاس **Startup** انجام می شود.

هرگونه **Configuration** که باید در زمان اجرای **ASP.NET Core** انجام شود، از این کلاس شروع خواهد شد و شامل چندین متد برای **Setup/Configure** وب اپلیکیشن شماست.

کلاس **Startup** مسئول پیکربندی دو جنبه مهم از اپلیکیشن است:

(۱) **رجیستر کردن سرویس‌ها**: هر کلاسی را که اپلیکیشن شما، به آن وابسته باشد باید در فایل **Startup** رجیستر شود. (چه در اپلیکیشن مشخص شود و چه توسط فریم‌ورک استفاده شود).

۲) **Middleware ها:** نحوه **Handle** کردن اپلیکیشن، برای **Request** ها به **Respond** ها از طریق **Middleware** هایی است، که اینجا مشخص می شود.

```
public void ConfigureServices(IServiceCollection services)
{
}

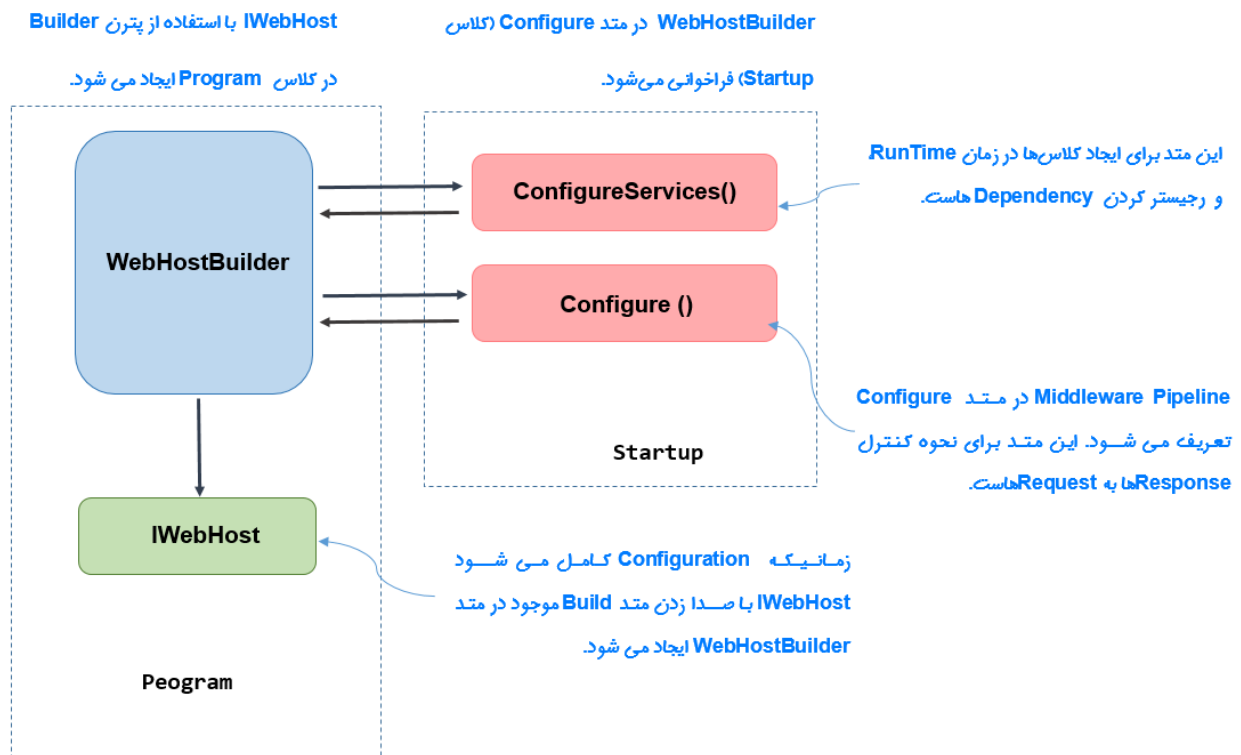
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
}
```

← رجیستر کردن سرویس ها با استفاده از
IServiceCollection

← پیگر بندی Middleware Pipeline برای
Handle کردن Request های HTTP

WebHostBuilder موجود در کلاس **Program**، متد **ConfigureServices** و **Configure** را فراخوانی می کند. هر فراخوانی یک بخش متفاوتی از درخواست شما را تنظیم می نماید، که آن تنظیم برای فراخوانی بعدی در دسترس است.

هر سرویس ثبت شده در متد **ConfigureServices**، برای متد **Configure** در دسترس است. هنگامی که پیگر بندی کامل شود، یک **IWebHost** توسط فراخوانی متد **Build()** موجود در **WebHostBuilder** ایجاد می شود.



Middleware در ASP.NET Core چیست؟

Middleware در ASP.NET Core ابزاری است که Request ها و Response ها را Handle می کند.

مهم ترین Middleware در اپلیکیشن های ASP.NET Core، MvcMiddleware است که می تواند HTML page، API، Response ها و.. را تولید کند.

قابلیت های Middleware :

مهم ترین قابلیت های Middleware را می توان به سه دسته تقسیم نمود:

- Middleware می تواند HTTP Request و HTTP Response را مدیریت کند.
- Middleware می تواند یک HTTP Request دریافتی را، پردازش، تغییر و به Middleware دیگر انتقال دهد.
- Middleware می تواند یک HTTP Response خروجی را، پردازش، تغییر و به Middleware دیگر منتقل کند.

با وجود این سه قابلیت نتیجه می گیریم: با استفاده از Middleware، می توان نحوه ی پاسخ گویی درخواست های HTTP را کنترل نمود.

موارد استفاده از Middleware

Middleware ها برای Handle کردن دو چیز بسیار مناسب هستند:

- Logging
- Error handling

اما مهم ترین موارد استفاده Middleware، رفع نگرانی های مدیریتی است.

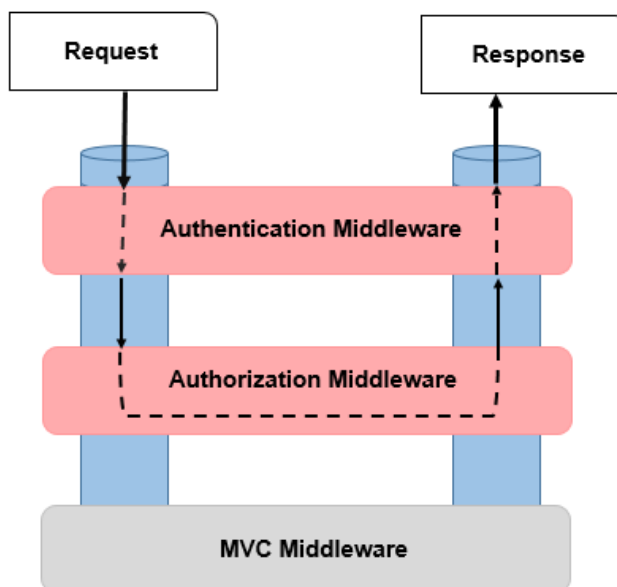
به طور مثال:

- Log کردن هر Request.
- افزودن Header های امنیتی برای Response ها
- Permission, Authorization
- تنظیمات زبان برای Request جاری.

نحوه عملکرد Middleware

نحوه عملکرد بدین صورت است که: **Middleware** یک درخواست را دریافت کرده، گاهی آن را تغییر داده و در نهایت به **Middleware** بعدی در **Pipeline** منتقل می‌کند. یا به عبارتی ساده‌تر می‌توان گفت: عملکرد یک **Middleware** بدین صورت است که، یک درخواست را دریافت و سپس آن را پردازش کند.

نکته کلیدی در اینجا، این است که **Pipeline**، دوطرفه است. تا زمانی که **Middleware** یک **Request** تولید کند، درخواست از یک طرف **Pipeline** در حال حرکت است. پس از تولید، **Request** از همان نقطه، از طریق **Pipeline** بازگشت داده می‌شود و **Middleware** را تا اولین **Middleware** طی می‌کند.



کاربرد متد ConfigureServices در کلاس Startup

در اپلیکیشن‌های **ASP.NET Core** هر زمان که می‌خواهید یک ویژگی جدید به برنامه اضافه کنید، باید آن را در متد **ConfigureServices** رجیستر نمایید.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

AddMvc، اکستنشن متدی بر روی اینترفیس **IServiceCollection** است که سرویس **MVC** را به اپلیکیشن شما اضافه می‌کند.

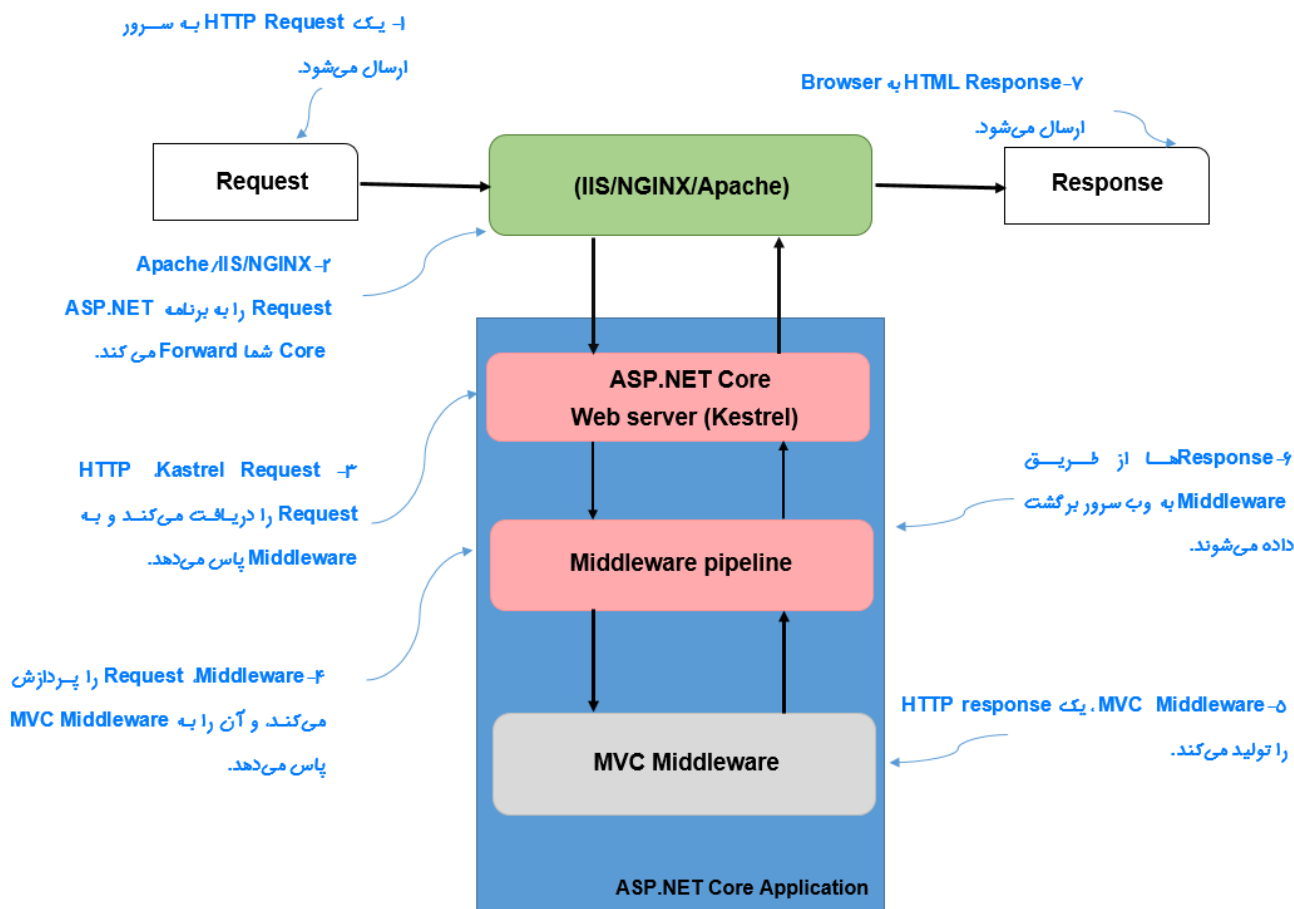
مطمئنا با دیدن این کد شگفت زده شدید!!

تنها با نوشتن یک متد `AddMVC()` یک برنامه کامل MVC در اپلیکیشن تان راه اندازی شد. با نوشتن این خط کد، پشت صحنه تمام سناریوهای مختلف، از جمله سرویس های `Razor`، `Render`، `HTML`، قالب بندی سرویس ها، سرویس های `Routing` و... انجام می شود.

چطور Request ها با Middleware، Handle می شوند؟

اپلیکیشن ASP.NET Core شامل بلوک هایی برای پردازش یک Request ورودی از Browser است.

- هر Request به طور پیش فرض به HTTP Server ما می رسد (Kestrel)،
- سپس HTTP server این درخواست را به Middleware Pipeline پاس می دهد و در اینجا، تغییراتی به آن Request داده خواهد شد.
- سپس این تغییرات به MVC Middleware پاس داده شده، تا Response تولید شود.
- Response تولید شده، از طریق Middleware به سرور برمی گردد و در Browser دیده خواهد شد.



نکته!!

Kestrel تنها HTTP Server در ASP.NET Core نیست، اما می‌توان به قطعیت گفت که این HTTP Server دارای Performance بالا و Cross-Platform است. و ما از IIS تنها به‌عنوان یک Reverse Proxy استفاده می‌کنیم تا بتوانیم برخی از قابلیت‌هایی که این HTTP server نوپا نمی‌تواند انجام دهد (مثلاً: امنیت، Load Balance و...) را به سرانجام رسانیم.

کاربرد متد Configure در کلاس Startup

Middlewareها، کامپوننت‌های کوچکی هستند که زمان دریافت یک HTTP Request، به صورت پشت سرهم اجرا می‌شوند. وظیفه‌ی متد Configure، تعریف رفتارهای اپلیکیشن شماست و باید هنگام دریافت این HTTP Requestها، Middlewareهای مورد نیاز را معرفی نماید.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseMvc(); ← MVC Middleware
}
```

در متد Configure، یک اینترفیس IApplicationBuilder تزریق شده است که از طریق آن می‌توانید ترتیب اجرای Middlewareها را مشخص کنید.

نکته!!

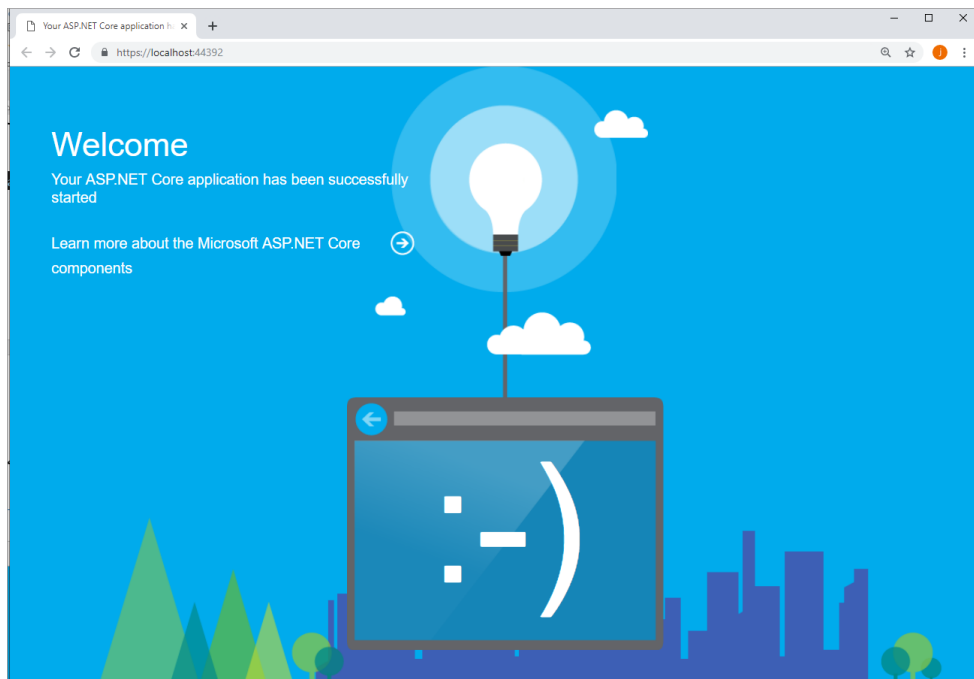
بیشتر Middlewareها به صورت Extension method برای اینترفیس IApplicationBuilder تعریف می‌شوند و معمولاً با عبارت Use شروع می‌شوند.

به مثال زیر دقت کنید:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    app.UseWelcomePage(); ← این Middleware برای نمایش صفحه خوشامدگویی است.
}
```

کد فوق یک **Middleware** ساده برای نمایش صفحه خوش آمدگویی است؛ که در حالت پیش فرض به هر درخواستی پاسخ خواهد داد:

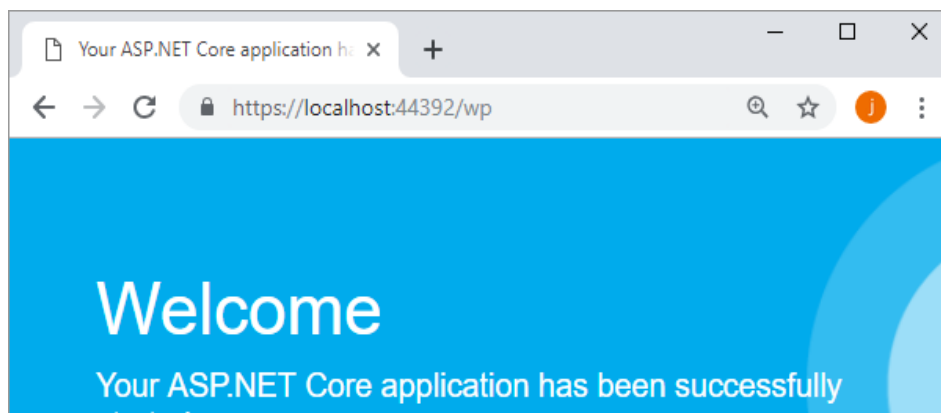
حالا با اجرای برنامه (F5):



شما می‌توانید **Option**هایی به این **Middleware** پاس دهید. به طور مثال در کد زیر مسیر نمایش تغییر داده شده است.

```
app.UseWelcomePage(new WelcomePageOptions  
{  
    Path = "/wp" ← مسیر نمایش صفحه خوشامدگویی.  
});
```

اکنون **Middleware** فوق زمانی نمایش داده خواهد شد که آدرس **https://localhost:44392 /wp** را وارد نمائید.

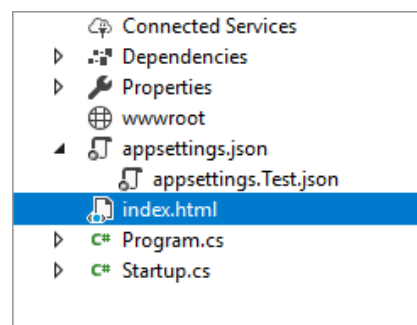


نکته!!

- ترتیب اجرای Middlewareها بسیار مهم است. به عنوان مثال Middlewareهایی که بعد از **Middleware** - **UseWelcomePage** نوشته شود، شانس نمایش را از دست خواهند داد.

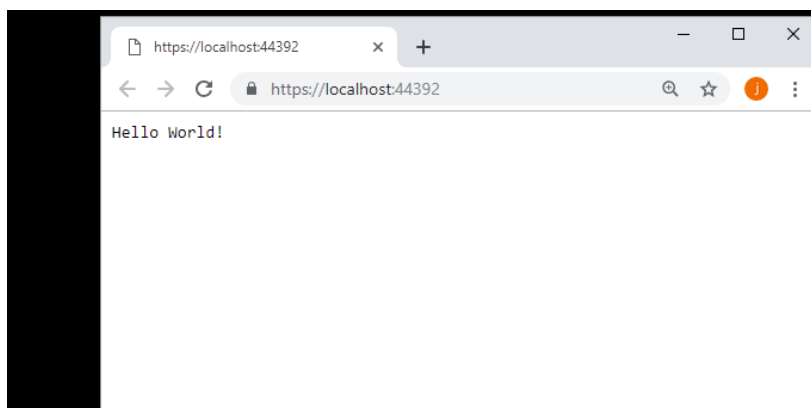
wwwroot

یکی از ویژگی‌های مهم هر وب‌اپلیکیشن، داشتن قابلیت استفاده از فایل‌های استاتیک است. سعی کنید به این پروژه یک فایل **HTML** اضافه نمایید و سپس برنامه را اجرا کنید.

فایل **Index.html**:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <h1>Wellcome!!</h1>
</body>
</html>
```

اجرای برنامه:



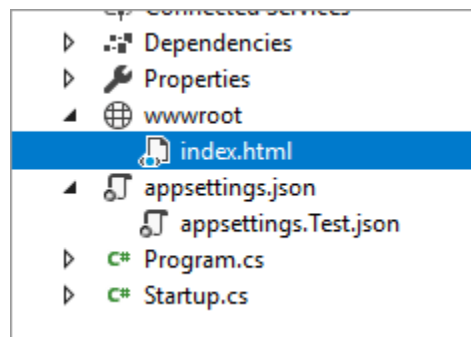
همانطور که می بینید محتوای فایل `index.html` نمایش داده نشد.

دو دلیل جهت عدم نمایش این فایل در خروجی وجود دارد:

۱. فایل های استاتیک باید درون **Folder** ← **wwwroot** قرار گیرند.

۲. درون **Pipeline** ما، هنوز **Middleware**ی جهت رسیدگی به درخواست نمایش فایل های استاتیک ثبت نشده است.

پس قدم اول: باید فایل `Index.html` را به درون **Folder** ← **wwwroot** انتقال دهید.



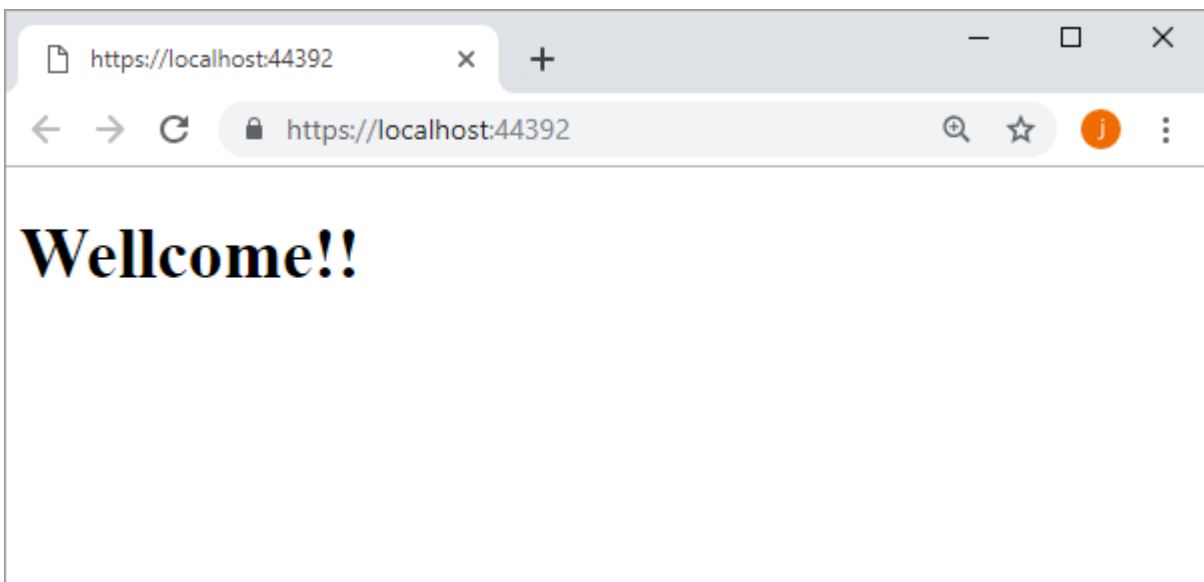
و قدم بعد افزودن **Middleware** هایی به نام `UseDefaultFiles` , `UseStaticFiles` در متد `Configure` کلاس `Startup` است.

در حالت پیش فرض `UseStaticFiles` به دنبال فایل های استاتیک درون پوشه ی `wwwroot` می گردد. اگر بخواهیم `index.html` به عنوان پیش فرض در هنگام اجرای پروژه در نظر گرفته شود، باید `Middleware` دیگری با نام `UseDefaultFiles` را قبل از `UseStaticFiles` قرار دهیم:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.Run(async (context) =>
    {
        var x = _configuration["TestSetting"];
        await context.Response.WriteAsync(x);
    });
}
```

این Middleware ها برای نمایش فایل `index.html` موجود در مسیر `wwwroot` است.

یکبار دیگر برنامه را اجرا کنید و خروجی را ببینید.



همانطور که قبلا گفتیم، ترتیب اجرای Middlewareها بسیار مهم است. اگر **UseDefaultFiles** را بعد از **UseStaticFiles** قرار دهیم، نتیجه مورد انتظار را دریافت نخواهیم کرد زیرا **UseDefaultFiles** در واقع عمل استفاده از فایل‌های استاتیک را انجام نمی‌دهد.

یک خبر خوب!!

Middleware دیگری با نام **UseFileServer** وجود دارد که کار هر دوی این **Middleware**ها را انجام می‌دهد:

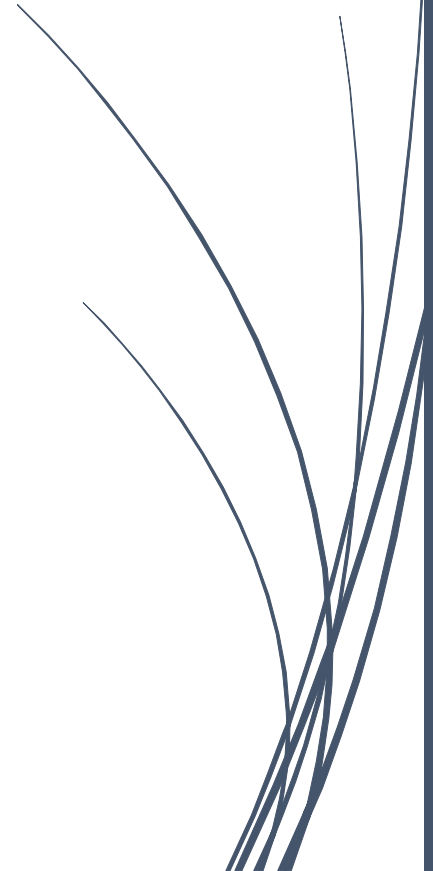
```
app.UseFileServer();
```

ما حتی می‌توانیم به این **Middleware** آپشن‌هایی مانند فعال‌سازی **Directory Browsing** هم ارسال نماییم.

فصل دوم : مدیریت Exception ها

➤ مدیریت Exception ها و Status Code ها

➤ تعریف Enviroment ها



مدیریت Exceptionها

یکی از جنبه‌های مهم یک اپلیکیشن، مدیریت Exceptionهاست. Exceptionها واقعیت زندگی، برای تمامی اپلیکیشن‌ها هستند. حتی اگر کدهای شما بسیار عالی هم نوشته شود، به محض اینکه اپلیکیشن خود را Release و Deploy نمایید، کاربران چه عمدا و چه سهوا، بالاخره راهی برای شکستن آن می‌یابند.

بنابراین در تولید اپلیکیشن، باید برای این‌گونه خطاها پاسخ مناسبی به کاربر ارائه دهید و با بهترین حالت این مسئله را مدیریت نمایید. این مسئله می‌تواند باعث شکست اپلیکیشن شما شود.

فلسفه‌ی طراحی ASP.NET Core این است که هر Feature، یک Option است. بنابراین مدیریت خطاها، یک Feature است که شما باید به‌صراحت آن را در اپلیکیشن خود فعال نمایید.

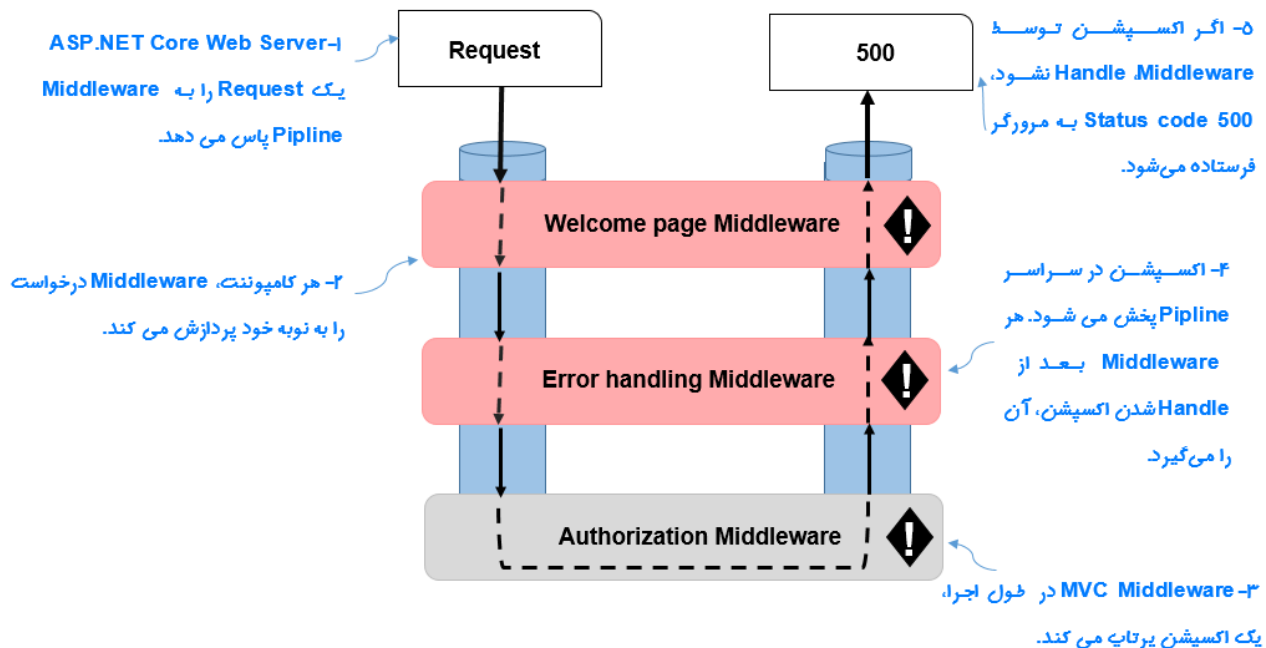
خطاهای متفاوتی می‌توانند در اپلیکیشن شما رخ دهند و راه‌های بسیاری برای بررسی آن وجود دارند، اما در اینجا می‌خواهم دو مورد از مرسوم‌ترین خطاها را به شما نشان دهم:

- Exceptionها

- و خطاهای Status code

Exceptionها: Exceptionها زمانی اتفاق می‌افتند که یک شرایط غیر منتظره یافت شود. **NullReferenceException** نمونه‌ای است که بدون شک همه‌ی شما آن را تجربه کرده‌اید و زمانی رخ می‌دهد که تلاش برای دسترسی به **Object**ی دارید، که هنوز **Initialized** نشده است.

اگر یک **Exception** در یک کامپوننت **Middleware** اتفاق بیفتد، (همانطور که در شکل زیر می‌بینید) در سراسر **Pipeline** منتشر می‌شود. اگر **Pipeline** آن را مدیریت نکند، وب سرور، **Status code 500** را به کاربر برمی‌گرداند.



گاهی اوقات هیچ **Exception** رخ نمی‌دهد اما **Middleware** ممکن است کد خطایی را ایجاد کند. یکی از مواردی که ممکن است دیده باشید، زمانیست که مسیر **Request**، **Handle** نمی‌شود. در این وضعیت **Pipeline** خطای **404** را برمی‌گرداند. در نتیجه صفحه خطای **404** به کاربر نمایش داده می‌شود.

گرچه این رفتار، صحیح است اما تجربه خوبی برای کاربران اپلیکیشن شما نخواهند بود.

برای حل این مشکلات، **Error handling middleware**، تلاش می‌کند قبل از اینکه اپلیکیشن، چیزی را به کاربر نمایش دهد، **Response** را تغییر دهد.

Error handling middleware، جزئیات خطای رخ داده را در یک صفحه کاربرپسند نمایش می‌دهد.

با توجه به اینکه هرگونه خطا می‌تواند اپلیکیشن شما را با شکست روبرو کند، پس عجله کنید و هرچه سریع‌تر **Error handling middleware** را به **Middleware Pipeline** اضافه نمایید.

شما می‌توانید **Exception**ها را به هر روشی که دوست دارید مدیریت کنید، اما استفاده از **Middleware**ها این کار را برای شما ساده‌تر کرده است.

با ما همراه باشید تا این قابلیت مهم را به پروژه اضافه نماییم...

مدیریت Exception ها و محیط های اجرایی

همانطور که بالاتر گفتیم، یکی از نگرانی های برنامه نویسیان، مدیریت Exception هاست. اما برای مدیریت Exception ها، یکی از اصلی ترین مسائل، موضوع محیط های اجرایی است.

اینترفیس دیگری که در متد **Configure** کلاس **Startup** تزریق شده، اینترفیس **IHostingEnvironment** می باشد. هدف این اینترفیس، ارائه رفتارهای متفاوت، در انواع محیط های توسعه می باشد.

به عنوان مثال: زمانیکه اپلیکیشن شما شروع به کار می کند اگر در محیط **Development** باشید، این متد باید Exception ها را طوری **Handle** کند که قابل **Track** توسط برنامه نویس باشد. و زمانیکه اپلیکیشن در محیط **Production** اجرا می شود، این **Exception** باید، با نمایش یک پیغام خطا همراه باشد.

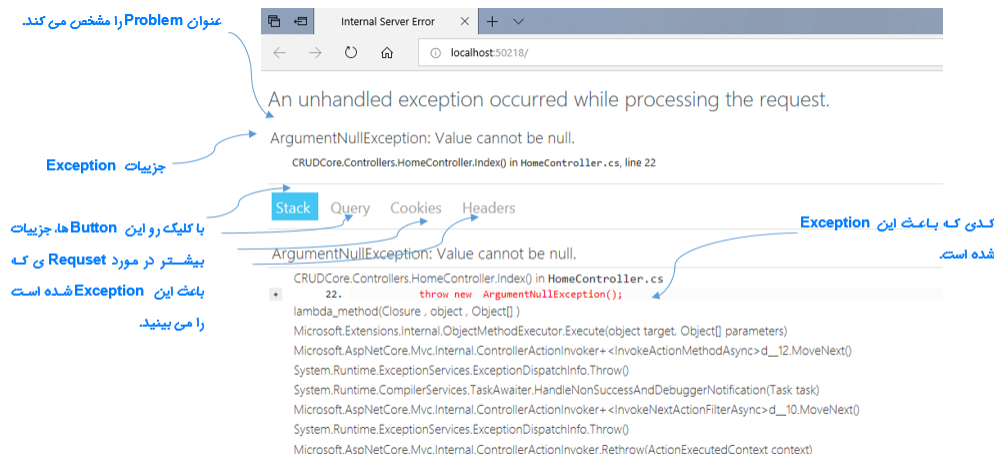
خوشبختانه با اینترفیس **IHostingEnvironment**، به این نگرانی پاسخ در ست داده شده است. پس می توان گفت: شی **IHostingEnvironment** حاوی اطلاعات کاملی در مورد محیط فعلی ما می باشد.

ما از طریق **IHostingEnvironment** به محیط های متفاوتی دسترسی خواهیم داشت و **Middleware Pipeline** ها بسته به اینکه در چه محیطی قرار داشته باشید، می توانند واکنش های متفاوتی را نشان دهند.

- **UseDeveloperExceptionPage**: با این **Middleware** تضمین می کنیم، اگر **Exception** رخ داد اطلاعات خطا در مرورگر نمایش داده شود. و همانند کد پایین باید بگوییم، اگر در محیط **Development** بودیم، اطلاعات **Exception** رخ داده شده را، برای **Track** برنامه نویس در مرورگر نمایش بدهد.

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

بعد از اجرای این خط کد، اگر **Exception** رخ دهد، این **Middleware** اجازه می دهد درخواست به **Middleware** های بعدی هدایت شود. اگر جایی دیگر در **Pipeline** خطایی رخ دهد، یک **UI** همراه با اطلاعات کاملی از وضعیت خطا به کاربر نمایش داده خواهد شد (تصویر زیر)



این صفحه شامل جزئیاتی در مورد **Request** و **Exception** است. و زمانی که یک خطا رخ می دهد، داشتن این اطلاعات به **Debug** برنامه کمک زیادی خواهد کرد.

Exception در محیط اجرایی

دانستن این اطلاعات بسیار ارزشمند است، اما تنها برای زمانی که در محیط **Development** قرار داریم. اگر این اطلاعات در محیط اجرایی پروژه هم، اینطور نمایش داده شود، **خطر امنیتی**، اپلیکیشن ما را تهدید خواهد کرد. زیرا جزئیات پروژه برای همه به نمایش درمی آید و هکرها راحت تر به اهدافشان می رسند.

به دلیل امنیت در محیط **Production**، دیگر نمی توانیم از **UseDeveloperExceptionPage** استفاده کنیم. پس در این محیط چطور **Exception** ها را **Handle** کنیم تا اپلیکیشن کاربر پسند باشد؟

خوشبختانه برای این موضوع، **Middleware** دیگری تعبیه شده که شما می توانید در صورت بروز **Exception**، نمایش پیغام را در صفحه ای سفارشی، به کاربر نمایش دهید.

با استفاده از **Middleware** بی به نام **UseExceptionHandler** این امکان به برنامه نویس داده می شود تا در صورت بروز یک **Exception**، کاربر با صفحه ای اختصاصی روبرو شود. این باعث می شود برنامه ما برای زمان های بحرانی کاربر پسندتر باشد.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
```

```
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else if (env.IsProduction())
    {

```

در صورتیکه در محیط **Development** باشیم و یک **Exception** رخ دهد، صفحه ای برای نمایش جزئیات **Exception** نمایش داده می شود.

در صورتیکه در محیط **Production** باشیم و یک **Exception** رخ دهد، صفحه ای سفارشی شما

نمایش داده می شود.

```
app.UseExceptionHandler("/home/error");
}
}
```



تعریف Environment

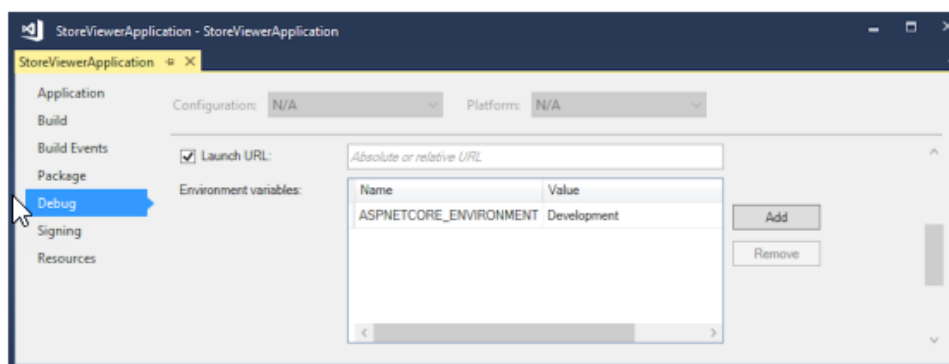
داشتن محیط‌های توسعه، باعث می‌شود شما برای هر هدفی، محیط اجرایی خود را **Handle** کنید؛ مثلاً برنامه باید در محیط تست با محیط تولید و... متفاوت عمل نماید.

اما یک سوال؟

کجا می‌توان این محیط‌های توسعه را تعریف یا تغییر داد؟

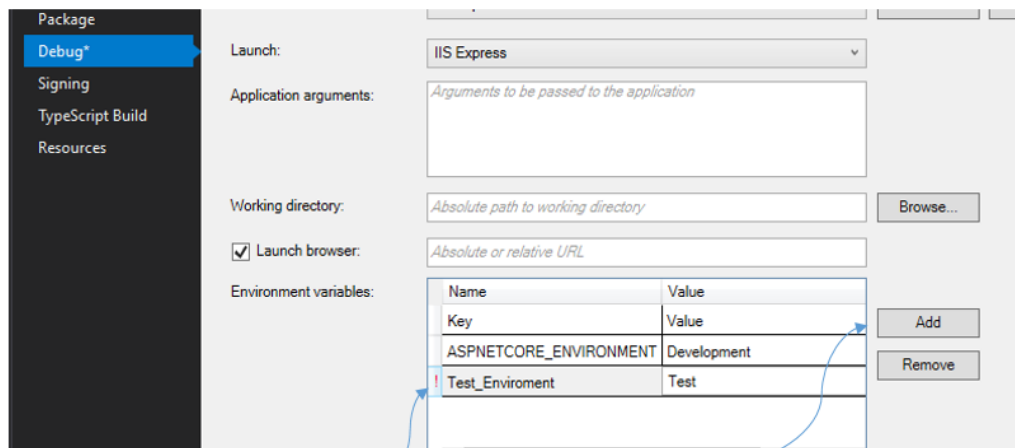
چند روش برای تنظیم **Hosting Environment** و تعریف **EnvironmentName** وجود دارد:

روش اول: روی پروژه راست کلیک کنید و با انتخاب گزینه **Properties** وارد کادر زیر شوید.



در تصویر بالا تنها یک محیط (**Development**) تعریف شده، اما می‌توانید محیط‌های دیگر را هم اضافه نمایید.

وارد تب **Debug** شوید و روی **Add** کلیک کنید. حالا می‌توانید نام محیط توسعه خود را اضافه نمایید.



بعد از زدن Add، در کادر Name، نام محیط توسعه و در کادر Value،

مقدار نمایشی در محیط برنامه نویسی را وارد نمایید.

نکته!!

خواهد کرد. اگر مقدار آن برابر با Development باشد، IsDevelopment برابر با true خواهد شد. اگر این Property وجود نداشته باشد ASP.NET Core فرض را بر این خواهد گذاشت که در محیط Production هستیم.

روش دوم: زمانی که یک پروژه از نوع ASP.NET Core ایجاد می کنیم، Visual Studio یک فایل launchSettings.json جهت پیکربندی اپلیکیشن اضافه خواهد کرد. شما می توانید با مراجعه به این فایل، محیط اجرایی جدیدی تعریف نمایید.



```

1  {
2  "iisSettings": {
3    "windowsAuthentication": false,
4    "anonymousAuthentication": true,
5    "iisExpress": {
6      "applicationUrl": "http://localhost:50218/",
7      "sslPort": 0
8    }
9  },
10 "profiles": {
11   "IIS Express": {
12     "commandName": "IISExpress",
13     "launchBrowser": true,
14     "environmentVariables": {
15       "Key": "Value",
16       "Test_Environment": "Test",
17       "ASPNETCORE_ENVIRONMENT": "Development"
18     }
19   },
20   "MicroDevCoreProject": {
21     "commandName": "Project",
22     "launchBrowser": true,
23     "environmentVariables": {
24       "Test_ENVIRONMENT": "Test"
25     },
26     "applicationUrl": "http://localhost:50219/"
27   },
28   "Docker": {
29     "commandName": "Docker",
30     "launchBrowser": true,
31     "launchUrl": "{Scheme}://{ServiceHost}:{ServicePort}"
32   }
33 }
34 }

```

می توانید نام محیط توسعه را تغییر دهید.

در پایان این پروسه، باید در کلاس **Service**، بگویید اگر در محیط **Test** بودید و یک **Exception** رخداد چطور **Handle** شود.

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else if (env.EnvironmentName == "Test")
    {
        app.UseExceptionHandler("/Employee/Error");
    }
    app.UseMvc(configureRoutes);
}

```

اگر در محیط **Test** باشید، صفحه ای سفارشی (در مسیر مشخص شده) برای نمایش خطا ظاهر خواهد شد.

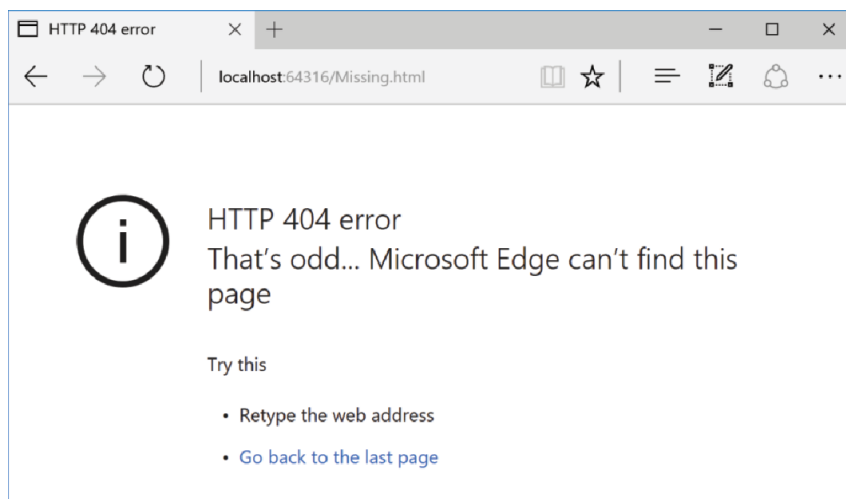
مدیریت Status code ها

ExceptionHandlerMiddleware و **DeveloperExceptionPageMiddleware** گزینه هایی مناسب، برای مدیریت **Exception** ها هستند، اما نکته ای که باید بدانید، این است که، **Exception** ها همیشه در قالب خطا ظاهر

نمی شوند؛ گاهی **Middleware Pipeline** یک **HTTP error status code** را در **Response** برمی گرداند که در این حالت، باید برای یک ارائه خوب به کاربر **Status code** ها را هم مدیریت نمایید.

برنامه ی شما برای نمایش انواع خاصی از وضعیت های خطا، می تواند طیف گسترده ای از **HTTP Status code** داشته باشد. به عنوان مثال: قبلا دیدید که عدد **500** یعنی **Server Error** و زمانی اتفاق می افتد، که یک **Exception** رخ داده و هنوز **Handle** نشده است. یا خطای **404** یعنی **file not found** و زمانی رخ می دهد که کاربر یک **URL** نامعتبر وارد می کند و این **URL**، **Handle** نمی شود.

بدون **Handle** کردن این **Status code** ها، کاربران صفحه ی خطای عمومی (مانند تصویر زیر) می بینند، که این می تواند کاملا گیج کننده باشد.

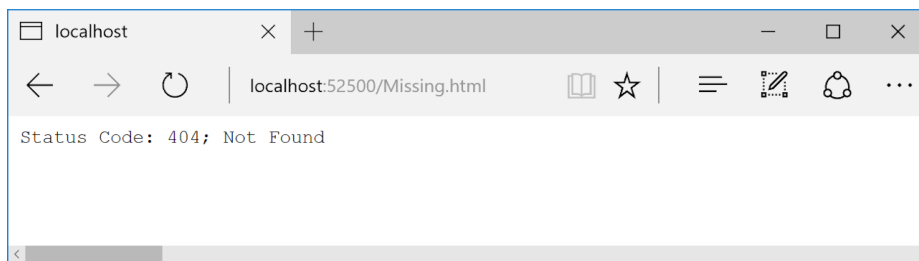


بهترین رویکرد برای حل این مسئله، مدیریت **Status code** ها و برگرداندن یک صفحه خطا با توجه به درخواست دریافتی می باشد. مایکروسافت هم با همین رویکرد **StatusCodePagesMiddleware** را تعریف کرده است.

```
app.UseStatusCodePages();
```

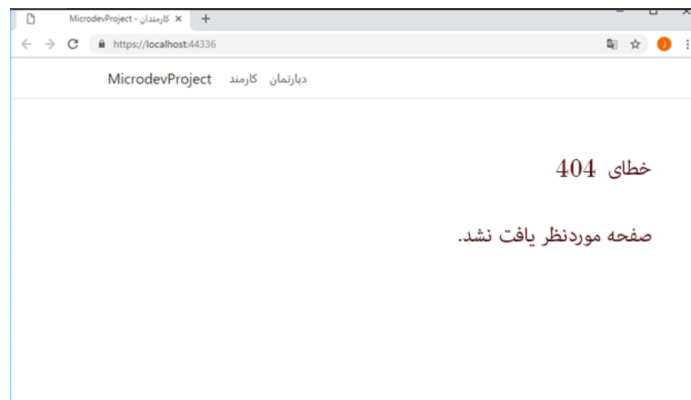
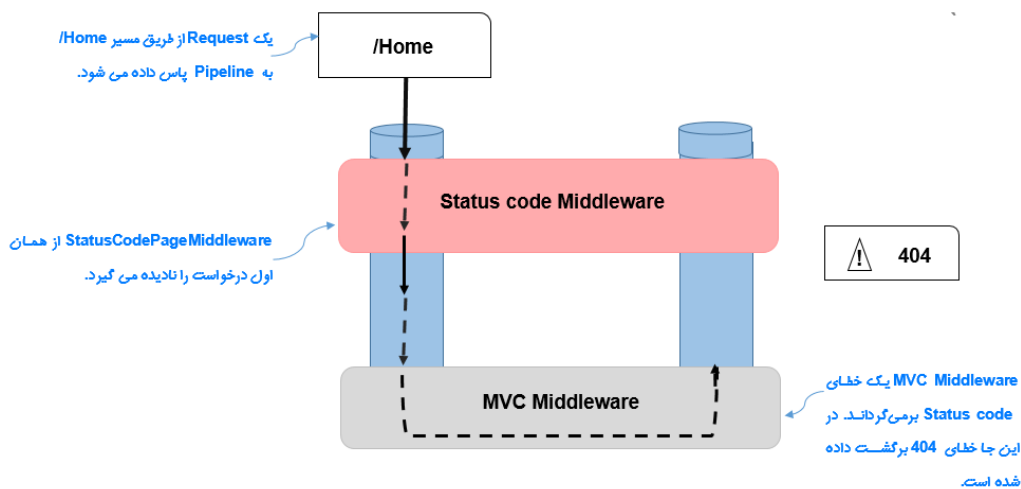
در این متد، **Middleware** هر **Response** را که **HTTP StatusCode** آن با **4xx** یا **5xx** شروع شده باشد و هیچ **Response body** نداشته باشد را **Intercept** می کند.

یک مثال ساده: جایی که شما هیچ گونه پیکربندی اضافه نمی کنید، **Middleware** (مانند تصویر زیر) یک متن **Response body** که شامل نوع و نام **Response** است را برمی گرداند.



شاید با دیدن تصویر بالا، بگویید: دیدن این صفحه، از خطای عمومی که بالاتر دیدیم هم، بدتر است. اما نظر من این است که این روش می‌تواند شروع خوبی برای یک ارثه‌ی بهتر باشد.

StatusCodePageMiddleware هم، شبیه **ExceptionHandlerMiddleware** استفاده می‌شود. پس در این محیط به جای استفاده از **UseStatusCodePages()** متد از **Extension** یک **method** بر روی **UseStatusCodePages** می‌باشد، که هر بار **Response code** بین **4xx** یا **5xx** باشد، از **Error HandlingPath** استفاده می‌کند.



- MVC و Routing چیست ؟
- مقدمه‌ای در مورد EF Core
- مقدمه‌ای در مورد Dependency Injection
- طول عمر یک سرویس ایجاد شده توسط DI

MVC چیست؟

MVC یک قالب عمومی برای طراحی اپلیکیشن‌های همراه با UI بوده و هدف این الگو، جدا سازی مدیریت و استفاده مجدد از داده‌ها، در نمایش می‌باشد.

این الگو یکی از الگوهای محبوب در طراحی UI Layer است.

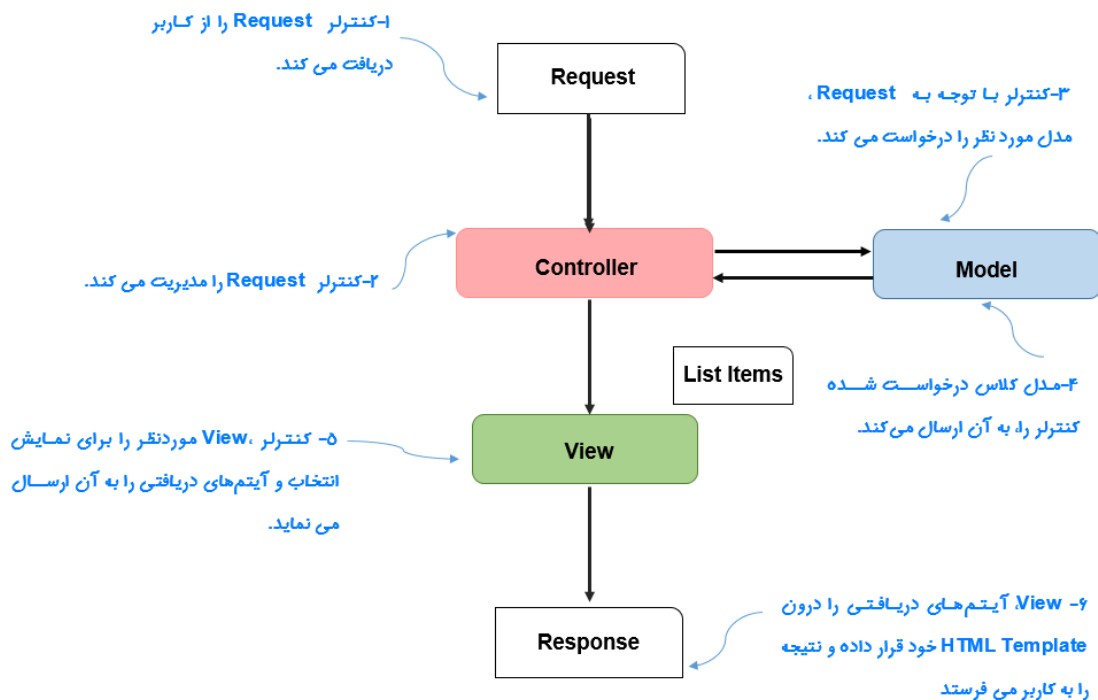
MVC بر مفهوم Separation Of Concern متمرکز است و فرض بر این است که با جدا سازی هر یک از این جنبه‌ها از هم:

۱- تمرکز بر روی Single Responsibility بیشتر می‌شود.

۲- وابستگی متقابل در سیستم کاهش می‌یابد.

به طور کلی سه مفهوم، MVC Design Pattern را می‌سازند:

- Model: داده‌هایی که باید نمایش داده شود.
- View: صفحه ای که داده‌های ارائه شده توسط مدل را نشان می‌دهد.
- Controller: مدل را Update و View موردنظر را نمایش می‌دهد.



مراحل انجام کار:

- Controller، HTTP Request را دریافت می کند.
- بسته به Request، Controller اطلاعات درخواست شده را از Model دریافت، یا اینکه Model را Update می نماید.
- Controller، View را برای نمایش و انتقال Model انتخاب می کند.
- View از اطلاعات موجود در Model برای UI استفاده می نماید.
- سپس این HTML به عنوان یک HTTP Response به کلاینت ارسال خواهد شد.

Routing چیست؟

MvcMiddleware برای مدیریت منطق‌های پیچیده‌ی اپلیکیشن شماسست و با Invoke کردن یک متد، یک Request مناسب را Handle می نماید. حالا یکی از سوال برانگیزترین جنبه‌های این Middleware، این است که چگونه یک متد برای اجرای یک Request انتخاب می شود؟

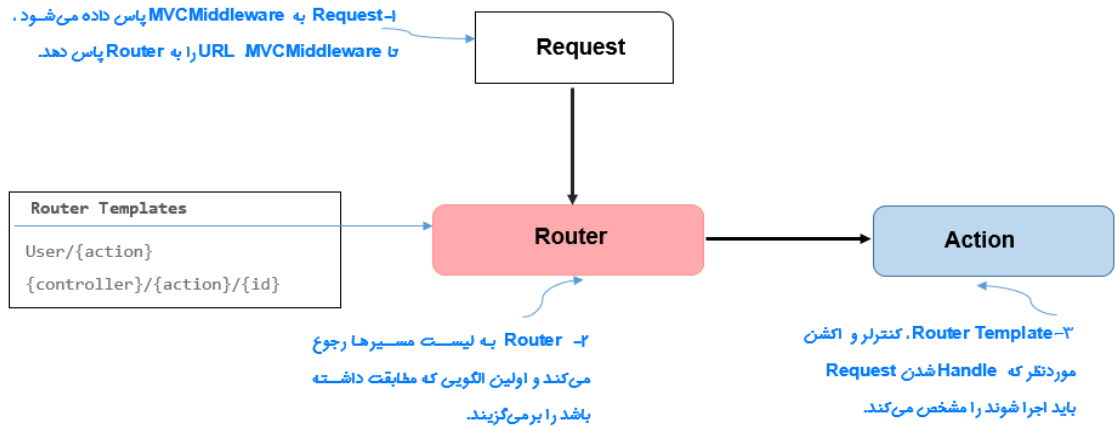
این جوابی است که Routing به ما می دهد:

Routing یا مسیریابی، قسمت کلیدی MVC Design Pattern در ASP.NET Core می باشد که Request ورودی را به یک Controller-Action خاص متصل می کند.

MVC با استفاده از Routing مسیریابی می کند. زمانی که، در اپلیکیشن خود از MvcMiddleware استفاده می نمایید، باید یک پیکربندی Routing برای Map کردن Request های ورودی به MVC Route انجام دهید. این Routing، URL را می گیرد و یک Controller و یک Action را مطابق با نیاز انتخاب می کند.

به عنوان مثال:

URL →/Employee/Create که به Route می رسد، دنبال EmployeeController و View ← Create آن می گردد.



حالا که با این مفهوم آشنا شدید، بیایید قابلیت Routing را در فایل Startup اضافه کنیم.

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

namespace MicrodevProject
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

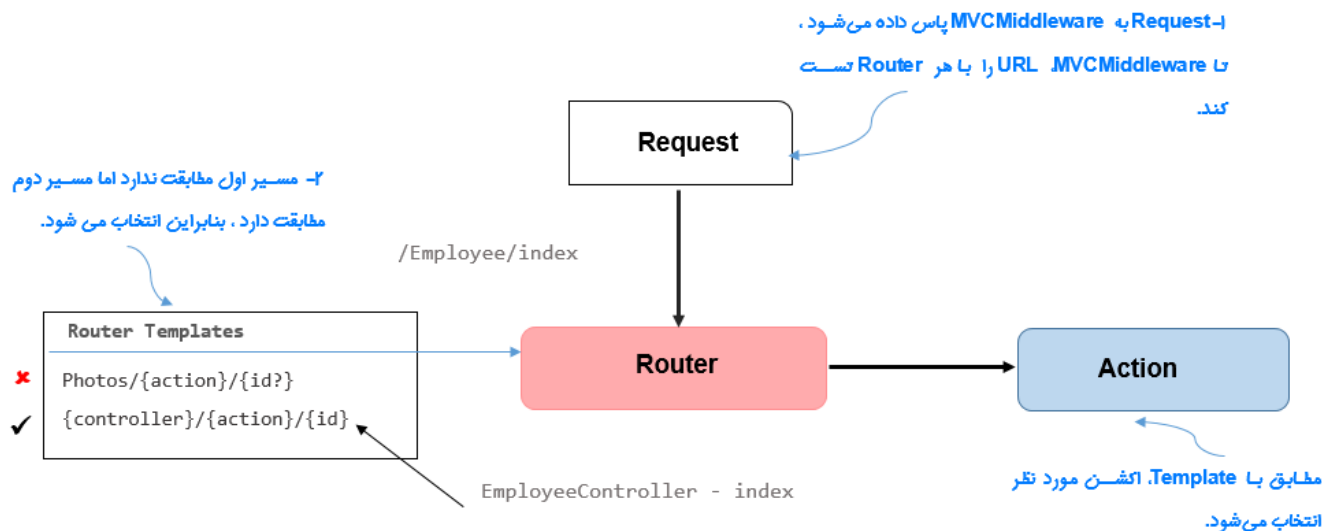
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Employee}/{action=Index}/{id?}");
            });
        }
    }
}
```

الگوی Route، ساختاری برای انطباق با URL تعریف می‌کند.

زمانی که شما متد UseMvc() را فراخوانی می‌کنید، باید برای مشخص کردن مسیرها در اپلیکیشن، یک عبارت Lambda بنویسید. با انجام این کار، در هر بار فراخوانی IRouteBuilder.MapRoute یک مسیر را تنظیم می‌کند.

شما می‌توانید با قالب گفته شده بالا، Routing Pattern‌های زیادی را اضافه نمایید که هر کدام می‌تواند یک عدد را برای Map کردن URL‌های مختلف به Action Method مورد نظر تنظیم کند.

قالب‌های مسیریابی، ساختار URL مشخص شده را، در برنامه شما تعریف می‌کنند. به طور مثال: URL‌های /Employee/index و /Employee/، هر دو با قالب Employee /{action=index} تطبیق داده می‌شوند.



```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Employee}/{action=Index}/{id?}");
        routes.MapRoute(
            name: "Login",
            template: "{controller=Account}/{action=Login}/{id?}");
    });
}
```

در MvcMiddleware نتیجه موفقیت مسیریابی، یک Request است که یک Action را در Controller مربوطه می‌یابد و سپس Middleware از این Action برای تولید Response مناسب استفاده خواهد کرد.

دو روش برای تعریف مسیریابی وجود دارد:

- مسیریابی از طریق MvcMiddleware (روشی که بالا گفتیم).
- استفاده از AttributeRouting.

به طور پیش فرض مسیریابی ذکر شده بالا، به عنوان یک مسیریابی عمومی در اپلیکیشن ها استفاده می شود. در این روش، باید Controller ها و Action های خود را تعریف نمایید تا Request دریافتی، مطابق با قرارداد حرکت و بالاخره مسیریابی مورد نظر خود را بیابد.

اما در AttributeRouting باید [Route] را بر روی Action Method های خود قرار دهید. این روش انعطاف پذیری بیشتری دارد و برای زمانی که از Web Api استفاده می کنید، بسیار کارآمد است.

```
[Route("[Employee]")]
public class EmployeeController: Controller
{
    private readonly IEmployeeService service;

    public EmployeeController(IEmployeeService service)
    {
        this.service = service;
    }
    [Route("")] ← عدم تعیین Template برای Route به معنای Default بودن آن است.
    public IActionResult Index()
    {
        var model = new EmployeeViewModel();
        model.Employees =service.GetAll();
        return View(model);
    }
    [Route("[Create]")] ← AttributeRouting برای مسیریابی
    [HttpGet]
    public IActionResult Create()
    {
        return View();
    }
}
```

توجه داشته باشید: در هر کدام از روش های ذکر شد، باید URL های مورد نظر خود را با استفاده از قالب مسیریابی تعریف نمایید. اگر در این قالب چیزی برای Route تعیین نشود، به معنای Default تعیین می شود. (Default یعنی: با اجرای شدن این Controller به صورت پیش فرض متدی که [Route("")] را داشته باشد اجرا خواهد شد).

حالت دیگری از AttributeRouting، استفاده از توکن است. در این حالت دیگر نیازی به نوشتن نام Controller یا Action Method نیست.

```

[Route("[controller]/[action]")] ← استفاده از توکن
public class EmployeeController: Controller
{
    private readonly IEmployeeService service;

    public EmployeeController(IEmployeeService service)
    {
        this.service = service;
    }

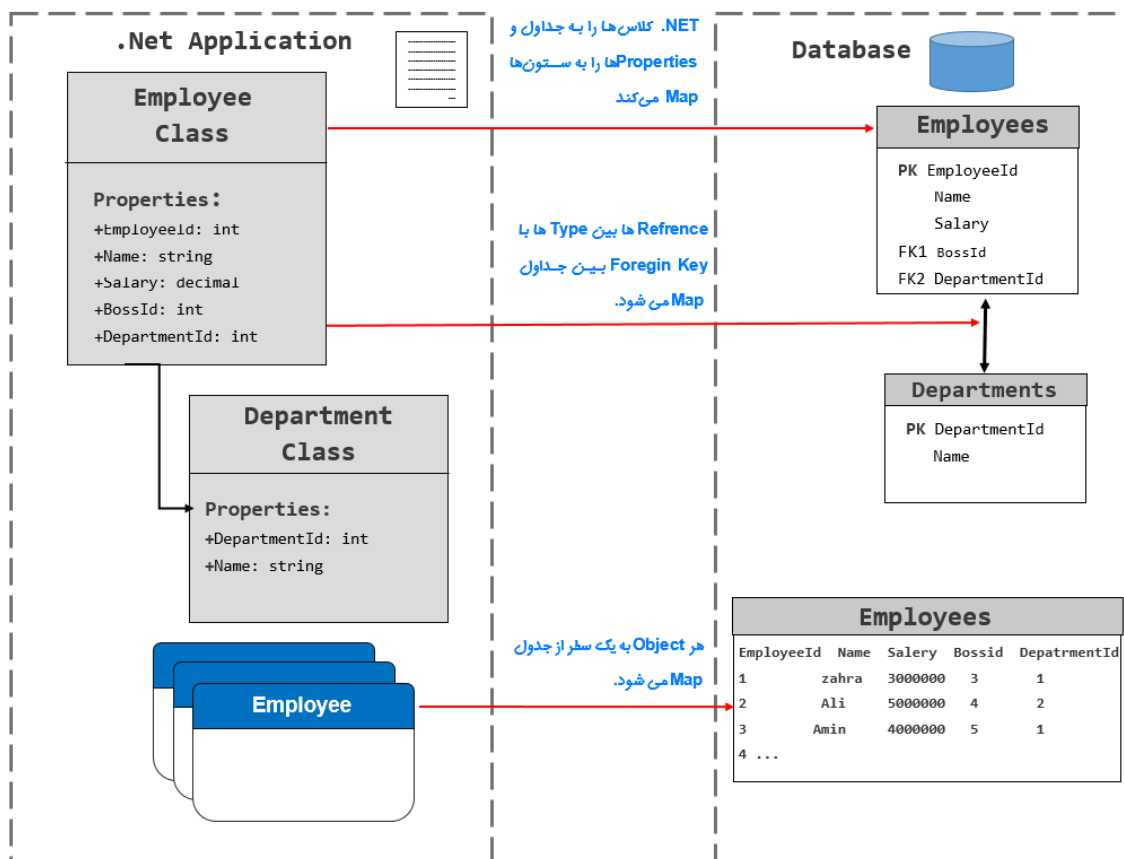
    public IActionResult Index()
    {
        var model = new EmployeeViewModel();
        model.Employees =service.GetAll();
        return View(model);
    }
    [HttpGet]
    public IActionResult Create()
    {
        return View();
    }
}

```

مزیت نوشتن توکن این است که، اگر بعداً نام **Controller** را تغییر دهیم؛ فریم‌ورک آن را تشخیص خواهد داد زیرا به صورت صریح نام کنترلر را ننوشته‌ایم.

مقدمه ای در مورد EF Core

EF core کتابخانه‌ای برای دسترسی به دیتابیس است. که به عنوان یک **ORM** عمل می‌نماید و با **Map** کردن کلاس‌ها و اشیاء داتانت به اجزای دیتابیس (مثل جداول و ردیف‌ها) بین آنها ارتباط برقرار می‌کند.



مهمترین ویژگی EF core برای Developer ها، نوشتن کدهای دسترسی به دیتابیس، در کمترین زمان و با کمترین هزینه است.

EF core یک کتابخانه بسیار کارآمد است که می‌تواند با طیف وسیعی از دیتابیس‌ها ارتباط برقرار کند. EF Core در سال ۲۰۱۶ توسط مایکروسافت منتشر شد و دارای قابلیت Multiplatform است. بدین منظور می‌تواند بر روی ویندوز، لینوکس و اپل اجرا شود.

این کار به عنوان بخشی از ابتکار .NET است و به همین دلیل، Core بخشی از نام EF Core می‌باشد. البته در همین جا بگویم که یکی از مهم‌ترین ویژگی‌های .NET Core، Open Source بودن آن است.

EF Core نسخه اولی‌ی Entity Framework نیست، EF Core با سال‌ها تجربه و بازخورد از نسخه‌های قبلی شروع و ایجاد شده است.

البته تغییرات عمده‌ای هم در آن وجود دارد، از جمله: Handle کردن دیتابیس‌های غیر رابطه‌ای که EF6 برای آن طراحی نشده بود.

نکته!!

یکی از بزرگترین مزایا که ORMها به ارمغان آورده‌اند، این است که با آنها می‌توان سرعت توسعه برنامه، را ارتقا داد. شما می‌توانید در بیشتر موارد، تنها با آشنایی مفاهیم شی‌گرا و بدون نیاز به دستکاری دیتابیس، تغییرات سفارشی خود را به دیتابیس اعمال نمایید.

تا اینجا با مفاهیم پایه آشنا شدید، حالا می‌خواهیم وارد بخش عملی پروژه شویم.

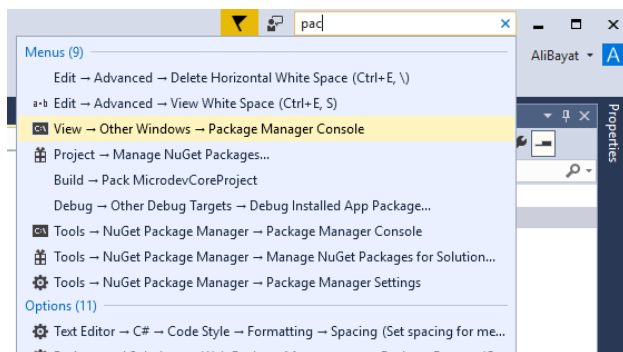
اضافه کردن دیتابیس به پروژه

برای سادگی این پروژه، می‌خواهم دیتابیس‌ی به نام **MicrodevDb** که دارای جداول **Employee** و **Department** می‌باشد را راه‌اندازی؛ و سپس با استفاده از **ASP.Net Core** عمل **CRUD** بر روی داده‌های این جداول اعمال نمایم.

برای ارتباط با دیتابیس، نیاز به یک **ORM** داریم و ما در **ASP.NET Core** از **EF Core** استفاده می‌کنیم.

EF Core بسیار انعطاف‌پذیر است، و به شما این امکان را می‌دهد تا **Entity**های خود را به هر شکلی که دوست دارید تعریف نمایید.

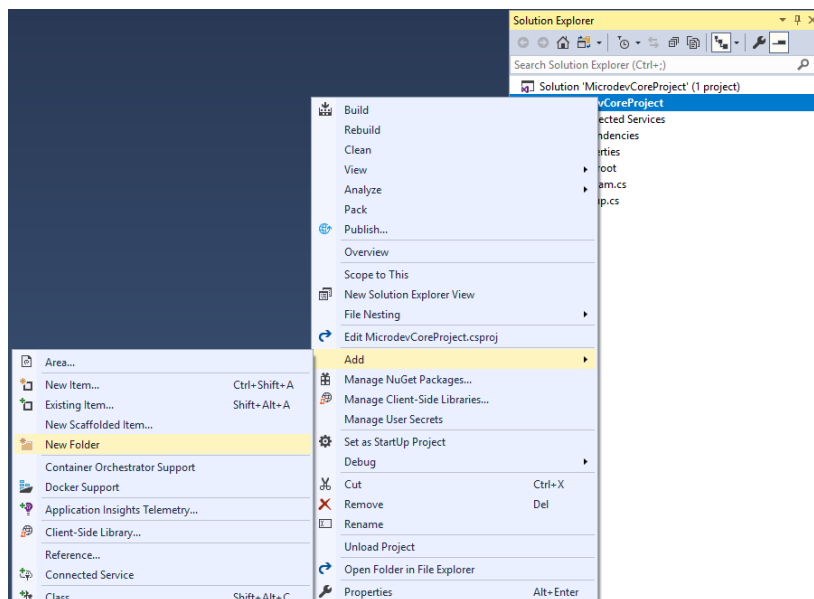
- قبل از هرکاری باید **EF Core** را به پروژه اضافه کنید. برای انجام این کار، مانند تصویر زیر **Package Manger Console** را باز نمایید.



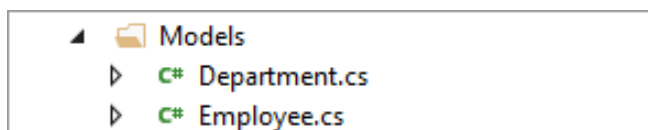
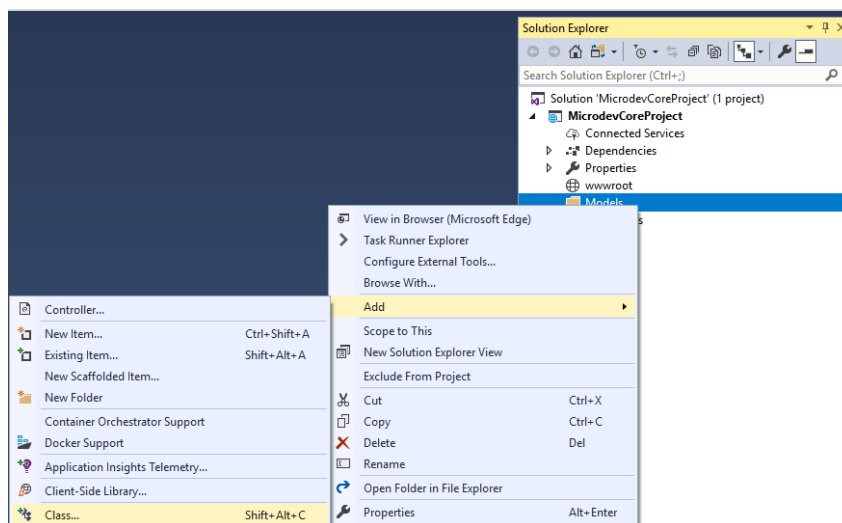
- حالا دستور زیر را در آن تایپ و در نهایت **Enter** را بزنید.

- **Install-Package Microsoft.EntityFrameworkCore.Tools**

- در سمت راست بر روی **Solution** راست کلیک نمایید و در منوی مربوطه با انتخاب گزینه‌ی **Add→Folder**، پوشه‌ای با نام **Models** ایجاد نمایید.



- در این پوشه کلاس‌هایی به نام **Employee** و **Department** بسازید. این دو کلاس **Entity** های دیتابیس ما هستند. (**Entity** شی است که در دیتابیس ذخیره می‌شود).



DataAnnotation چیست؟

داده‌ها قبل از ارسال باید اعتبار سنجی شوند و این، موضوع خیلی مهمی در ذخیره سازی اطلاعات است. برای حل این مسئله استفاده از **DataAnnotation** ها مطرح شده. **DataAnnotation** ها به شما این امکان را می‌دهند تا قوانینی مشخص کنید و **Properties** ها در **Model** مطابق با این قوانین عمل نمایند.

شیوهی کار بدین صورت است که، **DataAnnotation** ها **Metadata** هایی، برای توصیف داده تهیه می کنند و داده ها، باید از قوانین این **Metadata** ها پیروی نمایند.

به عنوان مثال:

به جای اینکه **null** بودن داده ها به صورت دستی چک شود، **[Required]** را بالای **Property** های خود بگذارید و دیگر خیالتان راحت باشد که نمی توان داده ی **Null** وارد نمود.

برخی **DataAnnotation** ها:

- **[EmailAddress]** : برای اعتبارسنجی فرمت ایمیل.
 - **[MinLength(min)]** : بررسی حداقل کاراکتر وارد شده.
 - **[Phone]** : برای اعتبارسنجی فرمت تلفن.
 - **[Required]** : ورود مقدار برای **Property** اجباریست.
 - **[Range(min, max)]** : بررسی مقدار بین حداقل و حداکثر.
 - **[Display(Name = "")]** : نام سفارشی شما برای نمایش در **View**.
- حالا وارد کلاس **Employee** شوید و **Property** های زیر را به آن اضافه کنید.

```
using System.ComponentModel.DataAnnotations;

namespace MicrodevProject.Models
{
    public class Employee
    {
        public int EmployeeId { get; set; }
        [Required]
        [Display(Name = "نام و نام خانوادگی")]
        public string Name { get; set; }
        [Required]
        [Display(Name = "حقوق دریافتی")]
        public decimal Salary { get; set; }
        public int? BossId { get; set; }
        public Employee Boss { get; set; }
        public int DepartmentId { get; set; }
        public Department Department { get; set; }
    }
}
```

- سپس وارد کلاس **Department** شوید و مقادیر زیر را به آن بیفزایید.

```
using System.ComponentModel.DataAnnotations;
```

```
namespace MicrodevProject.Models
{
    public class Department
    {
        public int DepartmentId { get; set; }
        [Required]
        [Display(Name = "نام شرکت")]
        public string Name { get; set; }
    }
}
```

هر کارمند می‌تواند در یک دپارتمان (شرکت) و یک رئیس داشته باشد. پس:

- **DepartmentId** به عنوان کلید خارجی برای ارتباط **Employee** با **Department** است.
- **BossId** هم، یک کلید خارجی از جدول **Employee** به خودش است.

تعریف DbContext

برای ارتباط با دیتابیس و استفاده از EF، باید یک کلاس ایجاد کنیم که از کلاس **DbContext** ارث‌بری کند. **DbContext** قلب EF Core است.

در **DbContext** کلاس‌های شما به صورت **DbSet<T>** تعریف می‌شود. **T** در واقع **Entity** ما می‌باشد. (**Employee** یا **Department**)

هر **DbSet** به یک جدول در دیتابیس **Map** خواهد شد. پس شما باید بابت هر **Entity** یک **DbSet** داشته باشید.

ایجاد DbContext :

یک **Folder** به نام **Data** در این **Solution** ایجاد و کلاسی با نام **MicrodevDbContext** را به آن اضافه نمایید.

این کلاس باید از کلاس **DbContext** ارث‌بری نماید و جهت تعریف جداول‌های شما در دیتابیس، باید شامل مجموعه **Property**‌هایی از نوع **DbSet** باشد.

DbContext در **ASP.NET Core** یک **Constructor** دارد که پارامتر **DbContextOptions<T>** را می‌گیرد و **option**‌های دیتابیس را تعریف می‌کند.

```
namespace MicrodevProject.Data
{
```



```

public class MicrodevDbContext : DbContext
{
    public MicrodevDbContext(DbContextOptions<MicrodevDbContext> options)
        : base(options) { }
    public DbSet<Employee> Employees { get; set; }
    public DbSet<Department> Departments { get; set; }
}

```

options در Constructor شامل جزئیاتی از جمله:
 ارث بری از DbContext
 Connection String می باشد.
 تعریف جداول شما در دیتابیس

ConnectionString چیست؟

زمانیکه می‌خواهید اپلیکیشن‌تان را توسعه دهید و در ماشین‌های مختلف مستقر نمایید، موضوع مشخص کردن مکان دیتابیس مطرح می‌شود.

نوع دیتابیس با توجه به بیزنس شما تعریف می‌شود، اما مکان دیتابیس باید روی سیستم شما یا هر جایی در سرور دیتابیس قرار گیرد.

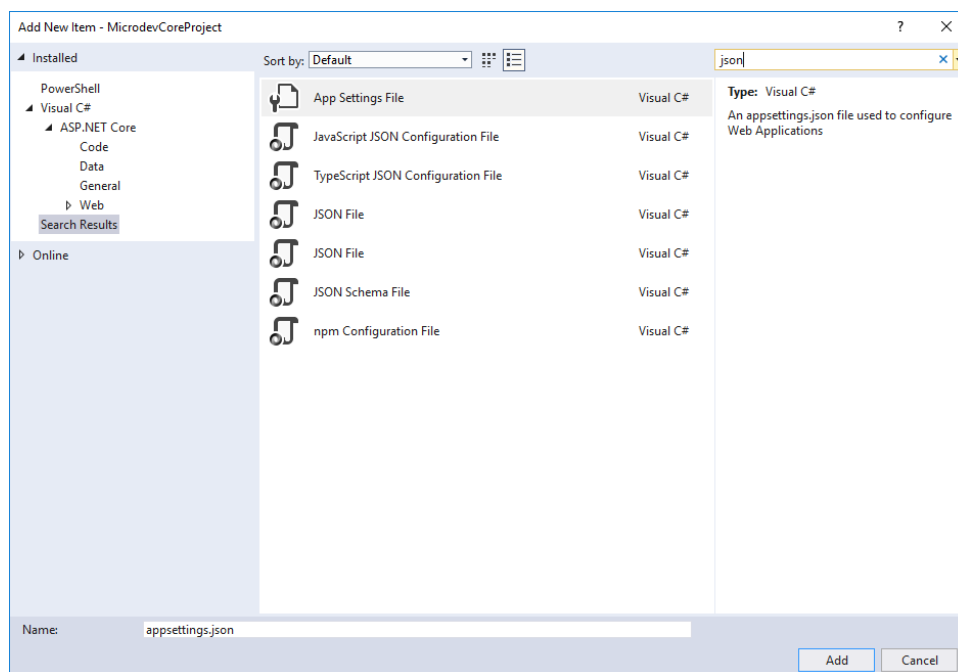
برای مثال:

در اپلیکیشن‌های وب، به‌طور معمول محل دیتابیس، بر روی یک Host قرار دارد (جایی که کاربران واقعی به آن دسترسی داشته باشند) و درون سخت افزار شما نیست. بنابراین مکان و تنظیمات مختلف دیتابیس معمولاً در یک ConnectionString ذخیره می‌شود.

ConnectionString به فریم‌ورک می‌گوید، دیتابیس روی چه سروری قرار دارد، پس بهتر است آن را درون فایل appsettings.json قرار دهیم تا بتوانیم بدون کامپایل مجدد، محل دیتابیس را در کامپیوترهای مختلف مشخص کنیم.

بیا به پروژه‌ی خود، فایل‌ی به نام appsettings.json اضافه و در آن ConnectionString را مشخص کنیم.

- روی پروژه راست کلیک نمایید و گزینه‌ی Add→NewItem را انتخاب کنید.
- حالا در کادر باز شده و در قسمت Search، کلمه json را تایپ و سپس App Setting File را انتخاب و در پایان OK نمایید.



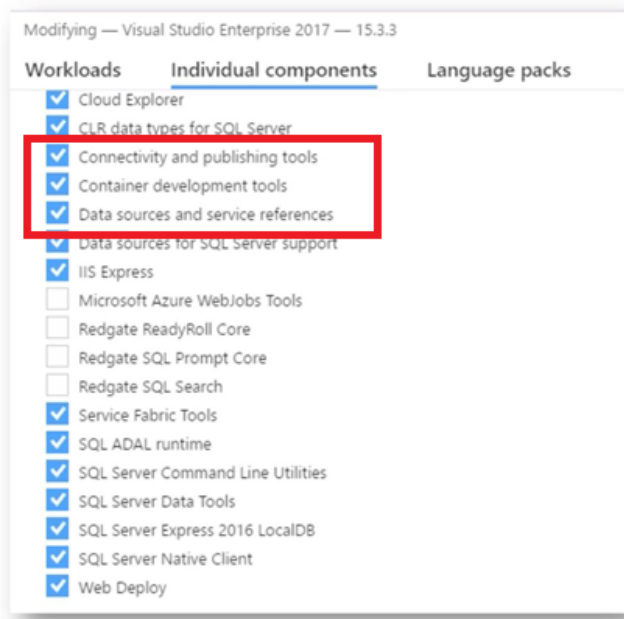
شما می‌توانید **ConnectionString** را به سلیقه خود **سفارشی** نمایید. البته باید **توجه** داشته باشید، نام این **ConnectionString** بعداً در معرفی **دیتابیس** موردنیاز است.

در تصویر بالا من نام **ConnectionString** را **MicrodevConnection** و نام دیتابیس را به **MicrodevDB** تغییر دادم. (به بقیه تنظیمات تغییری داده نشد).

نکته!!

LocalDB یک نسخه از **SQL Server** است. این نسخه از **SQL Server** همراه با **Visual Studio 2017** نصب خواهد شد.

اگر **LocalDB** بر روی سیستم شما نصب نشده است؛ می‌توانید **Visual Studio Installer** را اجرا و آن را نصب کنید:



حالا نوبت به ریجستر کردن **DbContext**، (در متد **ConfigureServices** کلاس **Startup**) می‌رسد.

اما قبل از انجام این کار، می‌خواهم کمی در مورد **DI** صحبت کنم.

مقدمه ای در مورد Dependency Injection

ممکن است قبلا در مورد **DI** شنیده، و شاید حتی در اپلیکیشن‌های خود از آن استفاده کرده باشید؛ اما اجازه دهید کمی در مورد این موضوع صحبت کنیم.

درک **DI** بسیار مهم است، زیرا **Asp.net core** به طور گسترده برای دریافت هر سرویسی از **DI** استفاده می‌کند.

DI یک **Design Pattern** است که این قابلیت را فراهم می‌کند تا به راحتی کدهای **Loosely Coupled** بنویسید.

شاید بپرسید، کدهای **Loosely Coupled** چیست و به چه دردی می‌خورد؟

در دنیای نرم‌افزار، تغییرات، بخش جدایی ناپذیر در اپلیکیشن‌های ماست. این تغییرات به مرور زمان باعث خراب شدن پایه‌های اپلیکیشن می‌شود. آقای رابرت سسیل مارتین معروف به **عمو باب**، اصولی به نام **SOLID** طراحی کرده که به ما کمک می‌کند تا نرم‌افزار را طوری طراحی کنیم که کمتر به شکست بینجامد.

▪ **اصل اول** **S - SRP - Single Responsibility Principle** : هر ماژول نرم‌افزاری می‌بایست تنها یک

دلیل برای تغییر داشته باشد.

- **اصل دوم O - OCP – Open Closed Principle** : ماژول‌های نرم‌افزار باید برای تغییرات بسته و برای توسعه باز باشند.
 - **اصل سوم L - LSP – Liskov Substitution Principle** : زیر کلاس‌ها باید بتوانند جایگزین نوع پایه‌ی خود باشند.
 - **اصل چهارم I - ISP – Interface Segregation Principle** : کلاینت‌ها نباید وابسته به متدهایی باشند که آنها را پیاده‌سازی نمی‌کنند.
 - **اصل پنجم D - DIP – Dependency Inversion principle** : مهم‌ترین اصل که می‌گوید : ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند، هر دو باید به انتزاعات وابسته باشند. انتزاعات نباید وابسته به جزئیات باشند، بلکه جزئیات باید وابسته به انتزاعات باشند.
- قلب این اصول، اصل D است که با استفاده از DI پیاده‌سازی می‌شود. همین مفهوم باعث ایجاد کدهای Loosely Coupled می‌شود. کدهای Loosely Coupled کدهایی هستند که وابستگی بین ماژول‌ها را کمتر می‌کنند. با کم شدن این وابستگی‌ها، تغییر یک ماژول تمام برنامه را تحت تاثیر قرار نمی‌دهد.
- فریم‌ورک ASP.NET Core برای ماژولار بودن از بهترین شیوه‌ی مهندسی نرم‌افزار یعنی اصول SOLID پیروی می‌کند. اصول SOLID یکی از بهترین تجربه‌ها در طول دوران برنامه‌نویسی شی‌گرا است.

مزایای DI

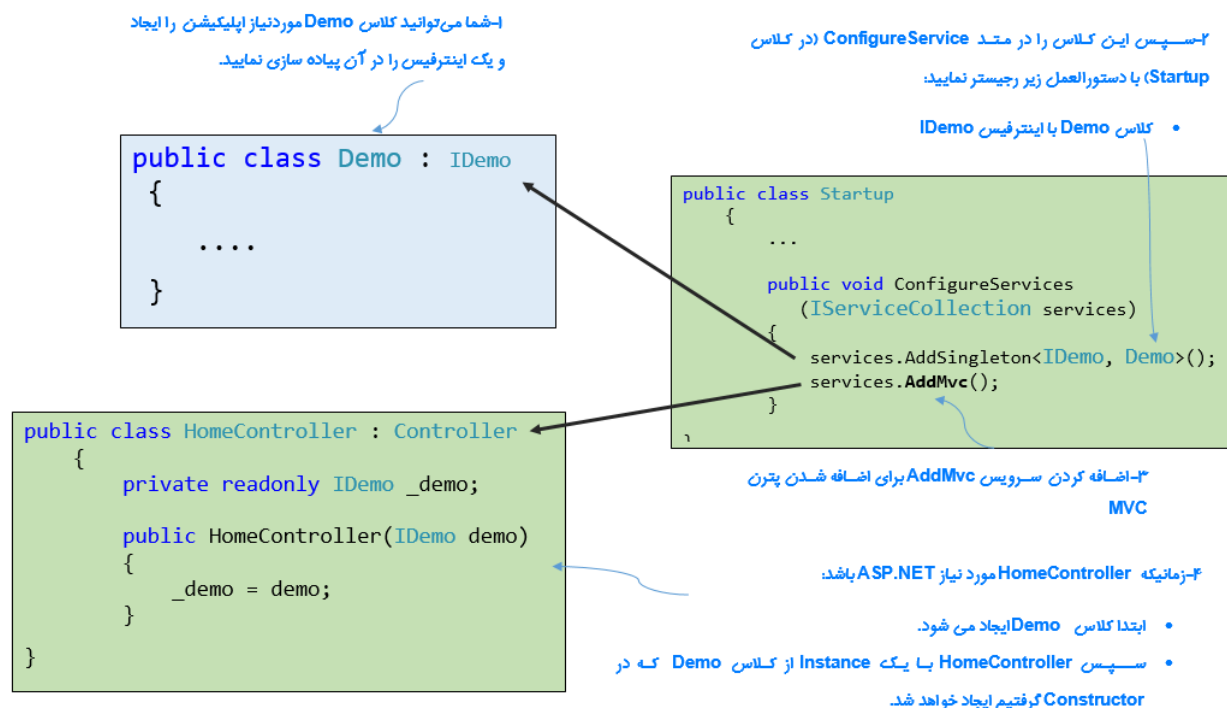
- DI به اپلیکیشن شما این امکان را می‌دهد، تا به صورت داینامیک با خودش پیوند داشته باشد. DI Provider به این صورت عمل می‌نماید که درخواست شما را دریافت و سپس با توجه به این درخواست، کلاس موردنظر را ایجاد می‌نماید.
- به طور مثال: اگر یکی از کلاس‌ها، نیاز به DbContext اپلیکیشن داشته باشد، DI آن را آماده می‌کند و دیگر نیازی نیست که در هر کلاسی به صورت دستی این کلاس را new کنید و کدهای Tightly Coupled ایجاد نمایید.
- استفاده از DI و اینترفیس‌ها باعث می‌شود اپلیکیشن، Loosely Coupled شود. Coupling مفهوم مهمی در برنامه‌نویسی شی‌گرا است، که در آن، عملکرد یک کلاس به کلاس دیگر وابسته است. شما می‌توانید یک اینترفیس داشته باشید که دو کلاس آن را پیاده‌سازی کرده باشند و هر زمان، هر کدام از

کلاس‌ها را که نیاز داشته باشید، با این اینترفیس **Match** کنید. این روش در **Unit Test** بسیار کاربرد دارد. (شما می‌توانید یک سرویس را با یک ورژن دیگر آن جایگزین کنید)

ما می‌توانیم با استفاده از **DI**، **تنظیمات پیشرفته** داشته باشیم و با توجه به درخواست، کلاس موردنظر را بازگشت دهیم.

مثلاً: در برنامه تجاری خود بخواهید یک کارت اعتباری **ساختگی** به جای کارت اعتباری واقعی داشته باشید و این، از طریق تنظیمات **Handle** شود.

اصل **DI** سیم کشی را به برنامه شما اضافه می‌کند که از طریق آن می‌توانید پیچیدگی را کمتر کنید.



در تصویر بالا ما توانستیم با استفاده از **DI** در **Asp.net core**، کلاس **IDemo/Demo** را رجیستر کنیم. سپس می‌توانیم در **HomeController** به آن دسترسی داشته باشید. کلاس‌هایی که توسط **DI** رجیستر می‌شوند اشاره به سرویس‌های ما دارند.

هر سرویس **DI**، می‌تواند به هر سرویس **DI** دیگر **Refrence** یا **Inject** شود.

در تصویر بالا: متد **AddMvc** می‌تواند، سرویس **IDemo/Demo** که در کلاس **Startup** رجیستر شده را در کلاس **HomeController** صدا بزند.

در این مثال شما اینترفیس **IDemo** را در کلاس **Constructor** کلاس **HomeController** تزریق کردید و **DI** یک **Instance** از کلاس **Demo** را در اختیار شما قرار داد.

طول عمر یک سرویس ایجاد شده توسط DI

طول عمر یک سرویس یعنی، قبل از ایجاد شدن **Instance** از سرویس، مدت زمان زنده ماندن سرویس در **Container** مشخص کنید.

یکی از ویژگی‌های مهم **DI**، بررسی طول عمر یک **Instance** ایجاد شده است.

چقدر طول می کشد که این Instance ایجاد شده **Dispose** شود؟

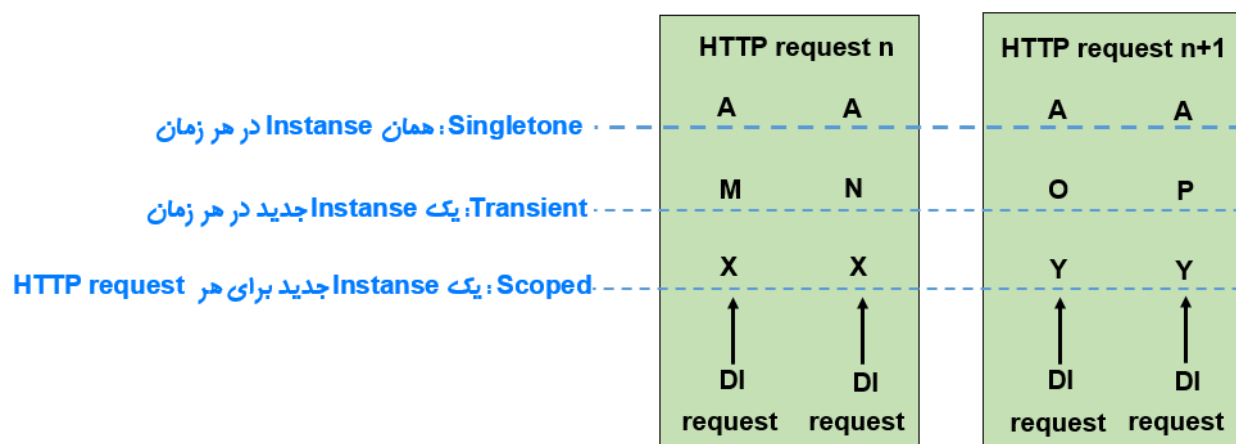
هر گاه از **DI**، یک سرویس رجیستر شده، درخواست شود یکی از دو مورد زیر اتفاق می افتد:

- یک **Instance** جدید از آن سرویس ایجاد و بازگشت داده می شود.
- یک **Instance** موجود از سرویس بازگشت داده می شود.

طول عمر یک سرویس با توجه به این دو مورد و در طول رجیستر شدن سرویس، تعریف خواهد شد.

این سخن تصدیق می کند، زمانی که یک سرویس ایجاد می شود، **DI Container**، می تواند **Instance** موجود از سرویس را، استفاده مجدد نماید.

یک سرویس می تواند در سه طول عمر در **DI Container** زندگی کند.



انواع طول عمر

- **AddTransient**: در این طول عمر، هر بار که یک سرویس درخواست شود، یک نمونه جدید ایجاد خواهد شد. این یعنی، شما می‌توانید نمونه‌های مختلفی از همان کلاس را در همان گراف وابستگی داشته باشید.
- **AddScoped**: در این طول عمر، به ازای هر درخواست HTTP یک نمونه از سرویس ایجاد می‌شود.
- **AddSingleton**: در این طول عمر، شما در طول کل حیات اپلیکیشن تنها یک نمونه از سرویس را دریافت خواهید کرد.

در مثال بالا شما یک **Instance** را به صورت **Transient** رجیستر کردید، پس هر زمان که یک **Instance** از این کلاس **Demo** را درخواست کنید، یک نمونه جدید ایجاد خواهد شد.

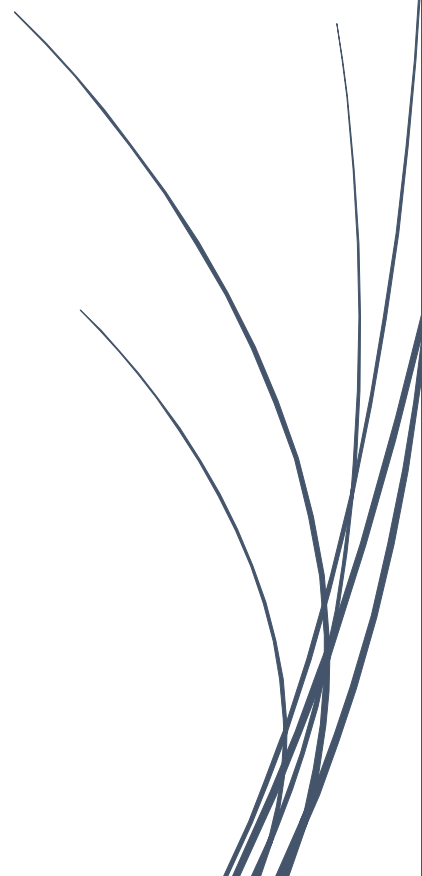
فراموش نکنید، اگر بخواهید کلاس‌هایتان را با **DI** استفاده کنید، حتما باید **طول عمر** آن را مشخص کنید.

تا اینجا با مفاهیم و مزایای **DI** آشنا شدید، با ما همراه باشید تا در فصل بعد، از این ابزار به صورت عملی استفاده نماییم.

➤ ریجستر کردن DbContext از طریق DI

➤ ایجاد Migration و Upadat دیتابیس

➤ تعریف Seed



ریجستر DbContext از طریق DI

در فصل قبل در مورد DI صحبت کردیم و شما با مزایای آن آشنا شدید. در این فصل قصد داریم DbContext (AppDbContext) را به عنوان یک سرویس در DI ريجستر نماییم.

قبل از ريجستر شدن این سرویس، باید نوع دیتابیس خود را مشخص کنید. EF Core از طیف وسیعی از دیتابیس‌ها پشتیبانی می‌کند، شما می‌توانید با توجه به مهارت خود، یکی را برگزینید:

- PostgreSQL—Npgsql.EntityFrameworkCore.PostgreSQL
- Microsoft SQL Server—Microsoft.EntityFrameworkCore.SqlServer
- MySQL—MySql.Data.EntityFrameworkCore
- SQLite—Microsoft.EntityFrameworkCore.SQLite

من در این آموزش از SQL Server استفاده کردم. پس باید پکیج `Microsoft.EntityFrameworkCore.SqlServer` در اپلیکیشن اضافه شود.

بنابراین دستور `Install-Package Microsoft.EntityFrameworkCore.SqlServer` را در `Package Manager Console` اجرا کنید.

```
Package Manager Console
Package source: All | Default project: MicrodevCoreProject
PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer
Restoring packages for C:\Users\Ali\source\repos\MicrodevCoreProject\MicrodevCoreProject\MicrodevCoreProject.csproj...
Installing NuGet package Microsoft.EntityFrameworkCore.SqlServer 2.1.4.
Committing restore...
Writing lock file to disk. Path: C:\Users\Ali\source\repos\MicrodevCoreProject\MicrodevCoreProject\obj\project.assets.json
Restore completed in 713.65 ms for C:\Users\Ali\source\repos\MicrodevCoreProject\MicrodevCoreProject\MicrodevCoreProject.csproj.
Successfully uninstalled 'Microsoft.EntityFrameworkCore.SqlServer 2.1.1' from MicrodevCoreProject
Successfully installed 'Microsoft.EntityFrameworkCore.SqlServer 2.1.4' to MicrodevCoreProject
Executing nuget actions took 711.89 ms
Time Elapsed: 00:00:02.0719015
PM>
```

حالا وارد فایل `Startup.cs` شوید و مراحل زیر را قدم به قدم دنبال نمایید.

۱- برای ارتباط و ایجاد دیتابیس، ابتدا باید بتوانیم `Connection String` را بخوانیم، اما چطور می‌توان

یک فایل `Json` را در بدنه برنامه بخوانیم؟

پاسخ این سوال را `IConfiguration service` می‌دهد.

یکی از قابلیت‌هایی که **WebHostBuilder** به ارمغان آورده است، سرویس **IConfiguration** می‌باشد. این سرویس، دسترسی به فایل‌های پیکربندی اپلیکیشن را، در اختیار شما می‌گذارد.

سورس‌های پیش‌فرض این سرویس:

- **فایلی با نام appsettings.json**
- **User secrets**
- **Environment variables**
- **Command line arguments**

خواندن محتوای **appsettings.json** توسط سرویس **IConfiguration**:

در کلاس **Startup.cs** یک **Constructor** ایجاد نمایید و اینترفیس **IConfiguration** را به آن **Inject** نمایید و در پایان، متغیر ورودی **Constructor** را در یک فیلد **readonly** قرار دهید.

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

namespace MicrodevProject
{
    public class Startup
    {
        private readonly IConfiguration configuration;

        public Startup(IConfiguration configuration)
        {
            this.configuration = configuration;
        }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

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

            app.UseMvc(routes =>
            {
                routes.MapRoute(
```

```
        name: "default",  
        template: "{controller=Employee}/{action=Index}/{id?}");  
    });  
}  
  
}  
}
```

نکاتی در مورد appsettings.json و Configuration service

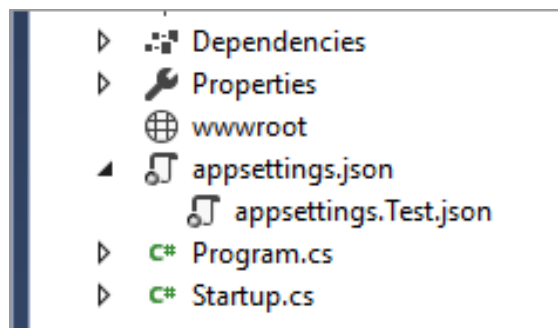
- اولین نکته: این سرویس به طور پیش فرض جستجوی خود را در فایل appsettings.json آغاز می کند، اگر در این فایل به نتیجه ای نرسد به دنبال سایر سورس های ذکر شده خواهد رفت.
- دومین نکته: ما می توانیم درون پروژه چندین فایل appsetting داشته باشیم. در حالت پیش فرض اپلیکیشن از فایل appsettings.json استفاده می کند؛ اما ASP.NET Core فایل دیگری با الگوی نامگذاری زیر را هم جستجو خواهد کرد:

appsettings.EnvironmentName.json

ما با استفاده از الگوی بالا می توانیم تنظیمات را برای محیط های مختلف سفارشی کنیم. شاید بپرسد چگونه؟

یک فایل با نام appsettings.Test.json ایجاد کنید و تنظیمات Connection String را درون آن قرار دهید. حالا اگر برنامه را برای محیط Test تنظیم نمایید، برای یافتن ConnectionString دنبال فایل appsettings.Test.json خواهد رفت.

مزیت این روش این است که در هر محیطی تنظیمات همان محیط اعمال می شود.



- دومین نکته: علاوه بر Connection String می توانید، کلیدهای دیگری هم در فایل appsettings.json داشته باشید.

➤ سومین نکته: شما می‌توانید در هر جایی از پروژه که باشید این اینترفیس را استفاده نمایید. به مثال زیر توجه داشته باشید:

من در فایل `appsettings.json` یک کلید `TestSetting` اضافه می‌کنم:

```
{
  "TestSetting": "DefaultSetting"
  //.....
}
```

و در کلاس `Startup` مقدار این کلید را درون متغیری قرار می‌دهم و به خروجی می‌فرستم.

```
public class Startup
{
    private readonly IConfiguration _configuration;

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

    public void ConfigureServices(IServiceCollection services)
    {
    }

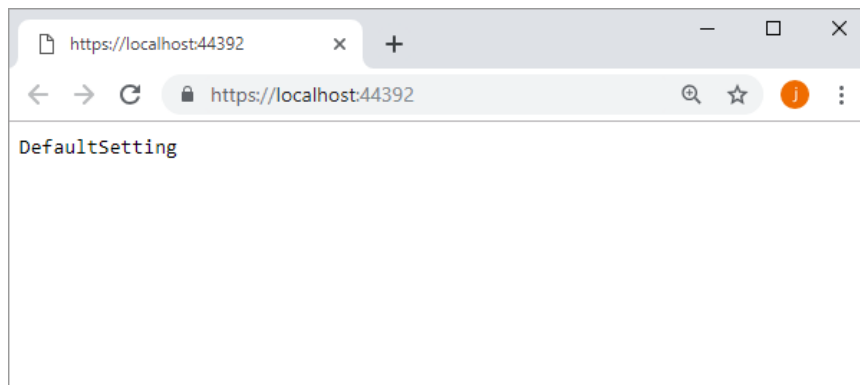
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.Run(async (context) =>
        {
            var setting = _configuration["TestSetting"];
            await context.Response.WriteAsync(setting);
        });
    }
}
```

خواندن مقدار کلید `TestSetting`

فرستادن متغیر `setting` به خروجی.

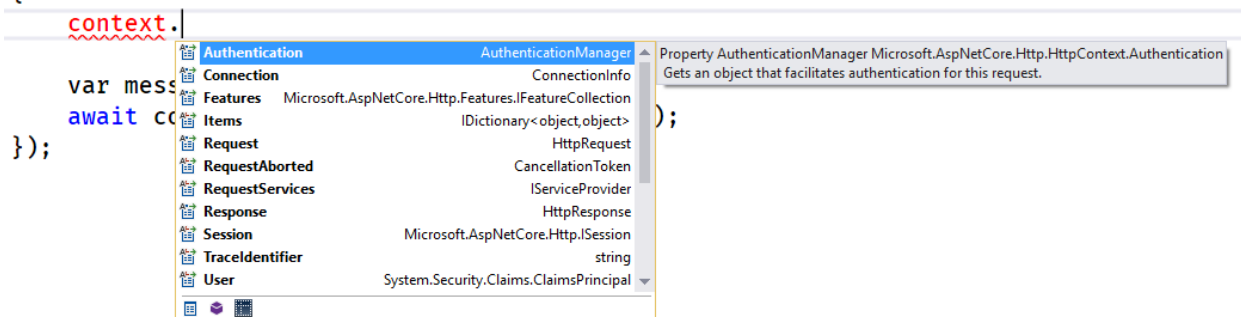
بعد از اجرای برنامه (F5):



نکته!!

متد Run امکان اجرای Middleware های سطح پایین را فراهم می آورد. درون این متد می توان با استفاده از شیء context به تمامی قسمت های یک درخواست دسترسی داشت. این متد یک تابع از ورودی دریافت خواهد کرد که ورودی و خروجی این تابع یک RequestDelegate می باشد. یک RequestDelegate نیز تابعی است که یک HttpContext از ورودی دریافت کرده و در نهایت یک Task به عنوان خروجی خواهد داشت.

```
app.Run(async (context) =>
{
```



۲- مرحله بعد، استفاده از **DbContext** برای معرفی دیتابیس به برنامه.

باز هم در اینجا، از **DI** استفاده می کنیم زیرا این روش برای انجام این مورد بسیار مناسب است.

ابتدا باید **ConnectionString** که بالاتر تعریف نمودیم را، به **DbContext** معرفی نموده و دیتابیس موردنظر را تعیین نماییم.

```
using MicrodevProject.Data;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

```

namespace MicrodevProject
{
    public class Startup
    {
        private readonly IConfiguration configuration;

        public Startup(IConfiguration configuration)
        {
            this.configuration = configuration;
        }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<MicrodevDbContext>(option =>
            {
                option.UseSqlServer(configuration.GetConnectionString("MicrodevCo
                nnection"));
                services.AddMvc();
            });
        }
    }
}

```

مشخص کردن یک Database Provider

از پارامترهای جنریک رجیستر کردن DbContext با استفاده

ConnectionString یک Token از تنظیمات می باشد، که در بخش appsetting فایل ConnectionString تنظیم می شود.

```

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

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Employee}/{action=Index}/{id?}");
    });
}
}
}

```

در کد بالا به **DI Provider**، نحوه ایجاد **DbContext** اپلیکیشن (با استفاده از پروسهی رجیستر کردن سرویس) گفته شد و حالا برنامه‌ی شما در هر نقطه از سیستم، می‌تواند به دیتابیس دسترسی داشته باشد.

نکته!!

اگر از DI استفاده نکنیم:

۱. برای هر بار دسترسی به دیتابیس، باید کد دسترسی را تکرار کنیم.
۲. رشته دسترسی به پایگاه داده ثابت می شود و اگر زمانی بخواهید سایت خود را جایی دیگر Host نمایید، به مشکل برخوردید، چون مکان دیتابیس برای Host دیتابیس، متفاوت از دیتابیس است که در محیط توسعه استفاده می کنید.

از لحاظ فنی چون `DbContext`، `IDisposable` را پیاده سازی کرده است، به صورت خودکار `Dispose` می شود و شما در هر بار درخواست `HTTP`، یک نمونه جدید را دریافت خواهید کرد. اینجا دیگر از طول عمر خبری نیست و ما از سه طول عمری که بالاتر گفتیم استفاده نکردیم.

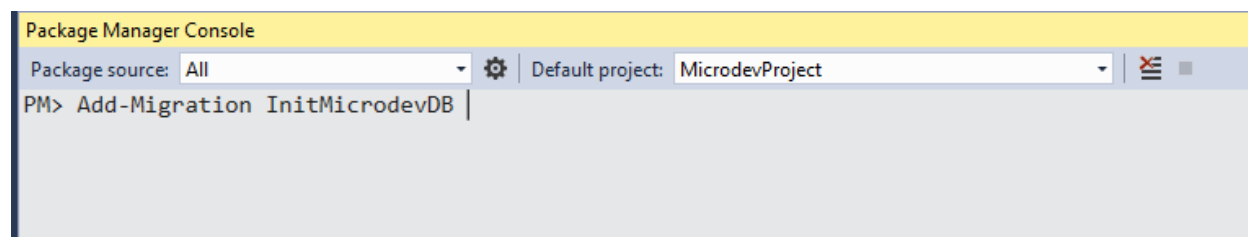
Entity Framework Migration چیست؟

بعد از انجام موفقیت آمیز مراحل بالا، نوبت به ایجاد دیتابیس می رسد. اما دیتابیس چطور ایجاد می شود؟ یک روش خوب برای ایجاد دیتابیس، وادار کردن EF به ساخت دیتابیس است. ساده ترین رویکرد EF برای انجام این کار، استفاده از `Migration` است. `Migration` راه حلی برای مدیریت جداول در دیتابیس می باشد. با `Migration` می توانید بدون هیچ گونه دردسری، تغییرات را به جداول دیتابیس اعمال نمایید.

ایجاد Migration

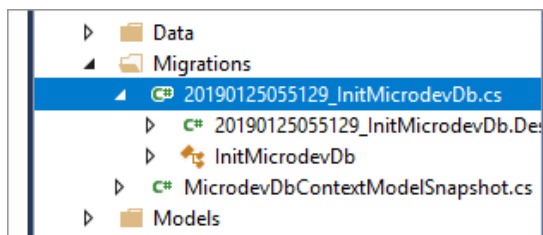
برای ایجاد `Migration` و تولید ساختار دیتابیس باید دستور `Add-Migration` را در `Package Manager Console` اجرا کنید. این دستور، ایجاد و آپدیت ساختار دیتابیس در اپلیکیشن را برعهده دارد.

پس `Package Manager Console` را باز نمایید و دستور `Add-Migration InitMicrodevDB` را تایپ و `Enter` را بزنید.



```
Package Manager Console
Package source: All | Default project: MicrodevProject
PM> Add-Migration InitMicrodevDB
```

حالا یک `Folder` به نام `Migrations` به `Solution` اضافه شده است. در این `Folder` کلاسی وجود دارد که کد ایجاد دیتابیس و جداول های ما در آن قرار گرفته است.



```

1 using Microsoft.EntityFrameworkCore.Metadata;
2 using Microsoft.EntityFrameworkCore.Migrations;
3
4 namespace MicrodevProject.Migrations
5 {
6     1 reference
7     public partial class InitMicrodevDb : Migration
8     {
9         0 references | 0 exceptions
10    protected override void Up(MigrationBuilder migrationBuilder)
11    {
12        migrationBuilder.CreateTable(
13            name: "Departments",
14            columns: table => new
15            {
16                DepartmentId = table.Column<int>(nullable: false)
17                    .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
18                Name = table.Column<string>(nullable: true)
19            },
20            constraints: table =>
21            {
22                table.PrimaryKey("PK_Departments", x => x.DepartmentId);
23            });
24        migrationBuilder.CreateTable(
25            name: "Employees",
26            columns: table => new
27            {
28                EmployeeId = table.Column<int>(nullable: false)
29                    .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
30                Name = table.Column<string>(nullable: true),
31                Salary = table.Column<decimal>(nullable: false),
32                BossId = table.Column<int>(nullable: true),
33                DepartmentId = table.Column<int>(nullable: false)
34            },
35            constraints: table =>
36            {
37                table.PrimaryKey("PK_Employees", x => x.EmployeeId);
38                table.ForeignKey(
39                    name: "FK_Employees_Departments_DepartmentId",

```

یک هشدار:

تغییر ساختار دیتابیس در وب سایت، باید به دقت مورد توجه قرار گیرد، بع ضی مواقع ب بسیاری از چیزها می توانند به اشتباه باشند و تاثیر آن می تواند باعث از دست رفتن اطلاعات یا خراب شدن وبسایت شود.

حالا نوبت به اعمال کدهای Migration به دیتابیس می رسد. سه روش برای اعمال این کدها وجود دارد:

- (۱) اپلیکیشن شما می تواند در طول اجرا شدن Startup، دیتابیس را چک و Migrate کند.
- (۲) می توانید یک اپلیکیشن مستقل برای Migrate دیتابیس داشته باشید.

۳) می‌توانید از دستورات SQL برای Update دیتابیس استفاده کنید.

ساده‌ترین روش، گزینه سوم است. شما می‌توانید، تنها با نوشتن Update-Database در Package Manager Console، این کدها را به دیتابیس اعمال نمایید.

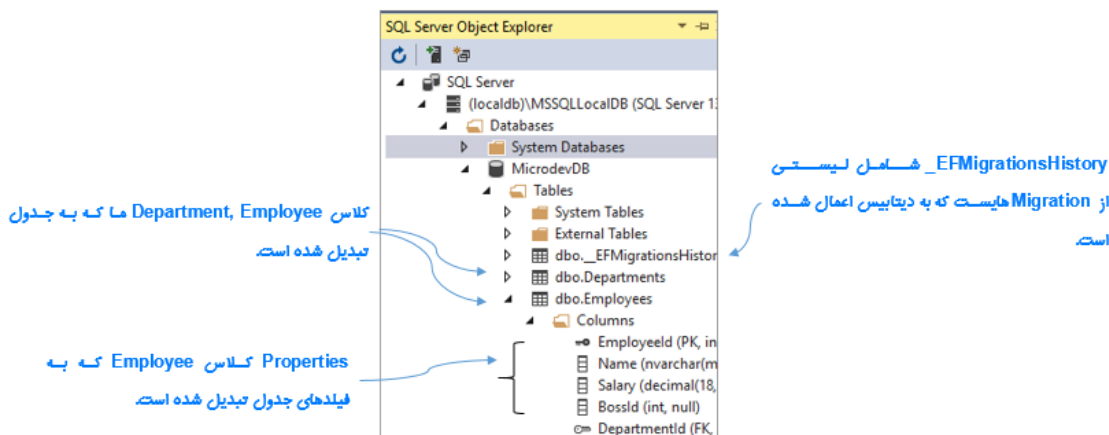
- Update-Database: این دستور برای اعمال کد Migration به دیتابیس است که در DbContext اپلیکیشن به آن اشاره شده بود. اجرای این دستور در دفعات بعد تنها دیتابیس شما را آپدیت می‌نماید.

```

Package Manager Console
Package source: All | Default project: MicrodevProject
PM> Update-Database
The EF Core tools version '2.1.1-rtm-30846' is older than that of the runtime '2.1.4-rtm-31024'. Update the tools for the latest features and bug fixes.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No type was specified for the decimal column 'Salary' on entity type 'Employee'. This will cause values to be silently truncated if they do not fit in
server column type that can accommodate all the values using 'ForHasColumnType()'.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 2.1.4-rtm-31024 initialized 'MicrodevDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (198ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[_EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']

```

به مسیر: **View → SQL Server Object Explorer** مراجعه کنید تا دیتابیس ایجاد شده را ببینید.



Seed چیست؟

در بسیاری از مواقع با اجرای اپلیکیشن، نیاز است تا برخی از جداول دیتابیس با اطلاعات پیش فرضی مقداردهی اولیه شوند. راه حل این مسأله، **Seeding** دیتابیس است.

Seeding دیتابیس، این امکان را به ما می‌دهد تا در حین اجرای برنامه، اطلاعاتی را به جداولی از دیتابیس اضافه نماییم.

برای اعمال قابلیت **Seeding** به اپلیکیشن:

➤ **مرحله اول:** باید کلاسی به نام **Seeder** داشته باشیم. این کلاس وظیفه‌ی مقداردهی اولیه به جداول را برعهده دارد.

- در پوشه **Data** یک کلاس به نام **Seeder** ایجاد نمایید.
- در این کلاس یک **Constructor** ایجاد کرده و **MicrodevDbContext** را به آن **Inject** نمایید و در پایان مقدار آن را درون یک متغیر **ReadOnly** قرار دهید.
- حالا یک متد **Seed()** تعریف کنید و با دستور **_context.Database.EnsureCreated();** دیتابیس را ایجاد نمایید.
- درون این متد، دو عبارت شرطی خواهیم داشت که چک می‌کنند آیا جدول **Employee** و **Department** رکوردی دارد یا خیر؟ در صورت منفی بودن لیستی از **Employee**ها و **Department**ها را ایجاد و در این جدول درج می‌نماید.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using MicrodevProject.Models;

namespace MicrodevProject.Data
{
    public class Seeder
    {
        private readonly MicrodevDbContext _context;

        public Seeder(MicrodevDbContext context)
        {
            _context = context;
        }
        public async Task Seed()
        {
            _context.Database.EnsureCreated();

            if (!_context.Departments.Any())
            {
                List<Department> departments = new List<Department>
                {
                    new Department{Name="روشنند مدیریت"},
                    new Department{Name="باز دکان"},
                    new Department{Name="پرداز ایده"},
                };
                _context.Departments.AddRange(departments);
            }
            if (!_context.Employees.Any())
            {
                // اگر در جدول Department هیچ شرکتی ثبت نشده باشد
                // لیست شرکت‌های بالا به این جدول اضافه می‌شود.
            }
        }
    }
}
```

```

List<Employee> employees = new List<Employee>
{
    new Employee {Name="بیات زهرا", Salary=1000 ,BossId=3,
    DepartmentId=1 },
    new Employee {Name="بیات علی", Salary=3000 ,BossId=5,
    DepartmentId=2 },
    new Employee {Name="مژگانی امین", Salary=1000, BossId=4,
    DepartmentId=1 },
    new Employee {Name="سهیلی محمد", Salary = 4000,BossId = 1,
    DepartmentId = 1 },
    new Employee {Name="احمدی سارا", Salary=1000, BossId=4,
    DepartmentId=2 },
    new Employee {Name="سعیدی محمد", Salary=1000, BossId=4,
    DepartmentId=2 },
    new Employee {Name="افتخاری سهیلا", Salary=2000, BossId=4,
    DepartmentId=2 },
    new Employee {Name="حسینی سعید", Salary=5000, BossId=4,
    DepartmentId=3},
    new Employee {Name="محمدی زهرا", Salary=1000, BossId=8,
    DepartmentId=3 }
};
_context.Employees.AddRange(employees);
await _context.SaveChangesAsync();
}
}
}
}

```

اگر در جدول Employee هیچ کارمندی ثبت نشده باشد لیست کارمندان بالا به این جدول اضافه می شود.

ثبت تغییرات در دیتابیس

➤ **مرحله دوم:** باید این کلاس را در متد **ConfigureServices** ریجستر نماییم.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MicrodevDbContext>(c =>
    c.UseSqlServer(configuration.GetConnectionString("MicrodevConnection"
    )));
    services.AddTransient<Seeder>();
    services.AddMvc();
}

```

ریجستر کردن کلاس Seeder

➤ **مرحله سوم:** تعریف یک **Scope** در متد **Configure** و پاس دادن سرویس بالا به آن است.

در این مرحله ما یک **Scope** ایجاد می کنیم و سپس کلاس **Seeder** را به متد **GetService** پاس داده و در پایان متد **Seed** را صدا می زنیم. عبارت **Wait** به این خاطر است که متد **Seed** در کلاس **Seeder** به صورت **Task** تعریف شده است.

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{

```

```

if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    using (var scope = app.ApplicationServices.CreateScope())
    {
        scope.ServiceProvider.GetService<Seeder>().Seed().Wait();
    }
}

app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Employee}/{action=Index}/{id?}");
});
}

```

حالا برنامه را اجرا کنید تا برای اولین بار داده‌های موردنظر شما در دیتابیس ذخیره شود.

برای دیدن این داده‌ها وارد مسیر **SQL Server Object Explorer** → **View** شوید.

داده‌های جدول **Department**:

| DepartmentId | Name |
|--------------|---------------|
| 1 | مدیریت روشمند |
| 2 | دکان یاز |
| 3 | ایده پرداز |
| NULL | NULL |

داده‌های جدول **Employee**:

| EmployeeId | Name | Salary | BossId | DepartmentId |
|------------|---------------|---------|--------|--------------|
| 1 | زهرا بیات | 1000.00 | 3 | 1 |
| 2 | علی بیات | 3000.00 | 5 | 2 |
| 3 | امین مزگانی | 1000.00 | 4 | 1 |
| 4 | محمد سهیلی | 4000.00 | 1 | 1 |
| 5 | سارا احمدی | 1000.00 | 4 | 2 |
| 6 | محمد سعیدی | 1000.00 | 4 | 2 |
| 7 | سهیلا افتخاری | 2000.00 | 4 | 2 |
| 8 | سعید حسینی | 5000.00 | 4 | 3 |
| 9 | زهرا محمدی | 1000.00 | 8 | 3 |
| NULL | NULL | NULL | NULL | NULL |

نکاتی در مورد EF Core:

- ✓ EF Core یک ORM است که از طریق آن می‌توانید با دیتابیس تعامل داشته باشید. این کار با دستکاری کلاس‌های استاندارد POCO که به آن Entity گفته می‌شود انجام می‌پذیرد.
- ✓ EF Core کلاس‌های Entity را به جدول‌های متناظر و Property‌های Entity را به Column‌های Table و هر Instance از شی Entity به یک ردیف از جدول Map می‌کند.
- ✓ EF Core از Database Provider استفاده می‌کند، که از طریق آن می‌توانید، بدون دستکاری کدهای خود، دیتابیس را تغییر دهید. EF Core دارای Providerهای Microsoft SQL Server, SQLite, PostgreSQL, MySQL و ... است.
- ✓ EF Core یک ORM چند سکویی یا همان Cross Platform با Performance بالا است.
- ✓ EF Core بر اساس Property‌های DbSet<T> در DbContext، یک نمای داخلی از Entity‌های برنامه شما و چگونگی Map شدن آنها به دیتابیس را ذخیره می‌کند. EF Core مدلی را مبتنی بر کلاس‌های Entity و ارتباط بین آنها ایجاد می‌کند.
- ✓ برای اضافه کردن EF Core باید پکیج Database provider را توسط NuGet نصب کنید.
- ✓ برای درست کردن و استفاده از Migration‌ها باید پکیج Microsoft.EntityFrameworkCore.Tools را نیز نصب نمایید.
- ✓ EF Core دارای تعدادی قرارداد پیش‌فرض برای چگونگی تعریف Entity‌هاست. مثلاً: کلید اصلی و کلید خارجی که می‌توانید آنها را با استفاده از DataAnnotation‌ها یا Fluent API سفارشی کنید.
- ✓ برنامه شما از DbContext و با استفاده از DI Container با دیتابیس تعامل برقرار می‌کند. دستور AddDbContext<T> برای رجستر شدن DbContext است و شما همین جا می‌توانید Database provider و ConnectionString را هم تعریف کنید.
- ✓ EF Core از Migration برای رهگیری تغییرات تعریف Entity‌ها استفاده می‌کند تا مطمئن شود، تعریف Entity‌های شما، مدل‌های داخلی EF Core و اسکیمای دیتابیس، همگی با هم یکی هستند.
- ✓ بعد از تغییر یک Entity، شما می‌توانید یک Migration از طریق ابزار NET CLI یا cmdlet‌های PowerShell ویزوال استودیو درست کنید.
- ✓ برای درست کردن یک Migration جدید از طریق NET CLI، می‌توانید دستور dotnet ef migrations add Name را در folder پروژه خود وارد نمایید. (Name نامی است که می‌خواهید برای نامگذاری Migration خود استفاده کنید.)

- ✓ برای اعمال **Migration** روی دیتابیس، می‌توانید از دستور **dotnet ef database update** استفاده کنید. این دستور اگر دیتابیس موجود نباشد، یک دیتابیس جدید ایجاد می‌کند، در غیر این صورت **Migration** را به دیتابیس شما اعمال خواهد کرد.
- ✓ وقتی شما **Migration** ایجاد می‌کنید، **EF Core** هیچ کاری روی دیتابیس انجام نمی‌دهد. فقط زمانی که دستور **dotnet ef database update** را صدا می‌زنید تغییرات **Migration** به دیتابیس اعمال می‌شوند.
- ✓ شما می‌توانید با درست کردن یک **Instance** از یک **Entity** و فرستادن آن به **DbContext** مثلاً **context.Add(e)** و سپس صدا زدن دستور **context.SaveChanges()** از **EF** بخواهید که دستور **Insert** در **SQL** را اجرا کند.
- ✓ شما می‌توانید رکوردهای دیتابیس را توسط پراپرتی **DbSet<T>** تعریف شده در **DbContext** را **load** کنید. این یک اینترفیس **IQueryable** را نشان می‌دهد، بنابراین شما می‌توانید از عبارتهای **LINQ** برای فیلتر و تغییر شکل داده‌های ورودی، قبل از بازگشت دادن آنها به خروجی استفاده کنید. اگر از عبارتهای سی شاربی استفاده کنید که **EF** نتواند به **SQL** ترجمه کند. ابتدا دیتاهای لازم را برای انجام آن کار در حافظه **load** می‌کند.
- ✓ **Update** کردن **Entity** ها در ۳ مرحله انجام می‌پذیرد : ۱- خواندن **Entity** از دیتابیس، ۲- تغییر دادن **Entity**، ۳- ذخیره کردن تغییرات در دیتابیس. **EF Core** تغییرات انجام شده در **Property** ها را **Track** می‌کند، بنابراین می‌تواند دستورات **SQL** ایجاد شده را بهینه کند.
- ✓ با استفاده از دستور **Remove** می‌توانید **Entity** ها را حذف کنید. اما برای استفاده از این دستور باید دقت کنید که دیتاهای ناخواسته حذف نشوند، گاهی استفاده از یک فلگ **IsDeleted** راه حل امن‌تری برای این منظور است.

فصل پنجم : عملیات CRUD

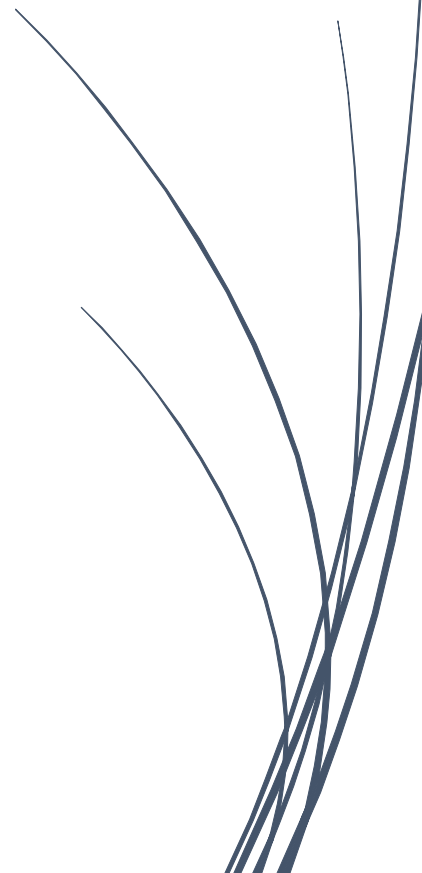
➤ مفهوم Separation Of Concerns

➤ Layout و Controller چیست؟

➤ افزودن TagHelper ، ViewStart و ViewImport به پروژه

➤ مفاهیم Validation و Model binding

➤ انجام عملیات CRUD



مفهوم Separation Of Concerns

برای اینکه بتوانیم کد مستقل داشته باشیم، باید محدوده **مسئولیت و تقسیم وظایف** صحیحی از برنامه داشته باشیم. **Separation Of Concern** یعنی هر شیء باید تنها یک کار را (وظیفه اش را) به بهترین وجه انجام دهد. این موضوع با دو اصل **Loosely coupling** و **Cohesion** ارتباط دارد. **Loosely coupling** یعنی تا جایی که امکان دارد باید هر قسمت از برنامه مستقل از قسمت‌های دیگر باشد، **Cohesion** یعنی هر قسمت از برنامه باید شامل کدهایی مرتبط با همان بخش باشد.

تفکیک برنامه به بخش‌های متمایز در سراسر برنامه سودمند است. در این قسمت می‌خواهیم مزایای **SOC** در پروژه **ASP.NET Core** را بیان کنیم:

Frontend، تماماً درباره نمایش اطلاعات است و برای انجام درست این وظیفه نیاز به تمرکز زیادی دارد. بنابراین شما از لایه **Service** برای **Handle** کردن دستورات **EF Core** و انتقال اطلاعات (به شکلی که به راحتی قابل استفاده در **Frontend** باشد) استفاده می‌کنید. (اغلب از طریق **DTO**ها یا همان **ViewModel**ها)

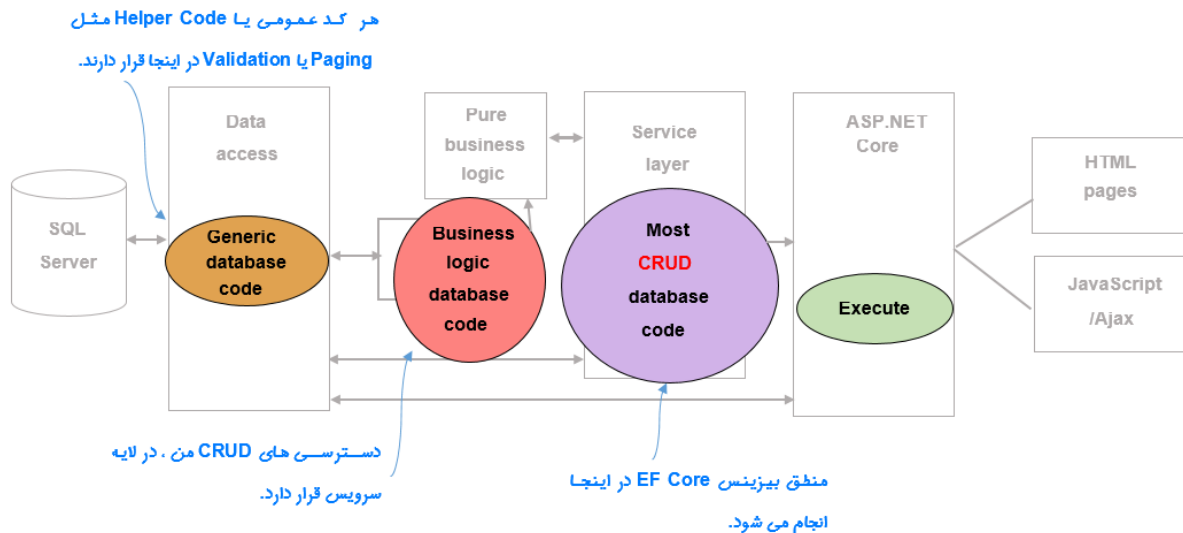
با این کار می‌توانید به جای فکر کردن به درستی **Query**های دیتابیس، بر روی ایجاد بهترین **User Experience** تمرکز کنید.

Controllerهای **Asp.net** معمولاً دارای چندین **Action** هستند (مثلاً یکی برای لیست اقلام، یکی برای اضافه کردن آیتم جدید، یکی برای ویرایش و...) که هر **Action**، نیاز به کدهای دسترسی به دیتابیس دارد. با انتقال کد دیتابیس به لایه **سرویس**، می‌توانید به جای اینکه کدها را در **Controller**ها پخش کنید، برای هر دسترسی به دیتابیس، یک کلاس جدا درست کنید.

و در نهایت نوشتن **Unit Test** برای کدهای دیتابیسی که در لایه **Service** قرار داده شده‌اند، بسیار ساده‌تر از زمانی است که این کدها را در **Controller**ها قرار دهیم. شما می‌توانید کدهای **Controller**ها را تست کنید، اما اگر در **Controller** به **Property**هایی مثل **HttpRequest** دسترسی داشته باشید (که همین طور هم هست) تست کردن بسیار پیچیده خواهد بود.

شروع عملیات CRUD

تا به اینجا با مفاهیم زیادی در **ASP.NET Core** آشنا شدید، و با هدفی که تاکنون پیش بردیم، می‌خواهیم یک پروژه **CRUD** را شروع کنیم.



نیازمندی های پروژه:

- ۱- لیستی از نام کارمندان را نمایش دهیم.
- ۲- بتوانیم جزئیات هر کارمند و دپارتمان آن را ببینیم و ویرایش کنیم.
- ۳- به لیست کارمندان اضافه و حذف نماییم.

برای انجام عملیات بالا، ابتدا سرویس های موردنظرمان را می نویسیم و سپس آن را به پروژه اضافه می کنیم.

روش استاندارد برای انجام عملیات بالا به شرح زیر است:

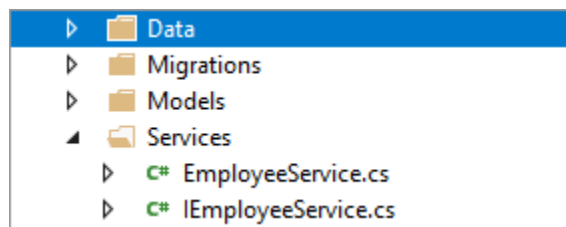
- ابتدا یک interface تعریف نمایید و هر آنچه به آن نیاز دارید را در آن مشخص کنید.
- سپس این interface را در یک کلاس پیاده سازی نمایید.

از لحاظ فنی این گونه برنامه نویسی، یک **Best Practice** است، زیرا می توان با استفاده از این اینترفیس **Unit test** را به آسانی در پروژه پیاده سازی نمود.

نکته!!

Controller تنها با این interface در ارتباط است پس مطمئن شوید تمام متدها و **Properties** های موردنیاز را در آن قرار داده اید.

در پروژه ی خود، پوشه ای به نام **Services** ایجاد و در آن اینترفیس **IEmployeeService** و کلاس **EmployeeService** اضافه نمایید.



توجه داشته باشید: باید برای **Department** هم، تمام این مراحل تکرار شود اما برای کاهش مطالب تکراری، در اینجا گفته نشده است.

ما باید عملیات زیر را در این **Interface** قرار دهیم:

- واکنشی اطلاعات تمام کارمندان
- واکنشی اطلاعات یک کارمند براساس کلید اصلی کارمند
- افزودن یک کارمند جدید
- ویرایش اطلاعات کارمند
- حذف یک کارمند

کد زیر را به اینترفیس **IEmployeeService** اضافه نمایید.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using MicrodevProject.Models;
using Microsoft.AspNetCore.Mvc.Rendering;

namespace MicrodevProject.Services
{
    public interface IEmployeeService
    {
        Task<IEnumerable<Employee>> GetAllEmploeeysAsync();
        Task<Employee> GetEmployeeAsync(int? id);
        IQueryable<SelectListItem> GetDropDownEmployees();
        IQueryable<SelectListItem> GetDropDownDepartments();

        Task<Employee> AddAsync (Employee employee);
        Task<Employee> UpdateAsync(Employee employee);
    }
}
```

دریافت لیست تمام Employee ها

دریافت یک Employee براساس Id

دریافت لیست Employee و Department

افزودن یک Employee

Update یک Employee

```
Task DeleteConfirmedAsync(int id); ← | عملیات حذف یک Employee
}
}
```

حالا (همانند کد زیر) در کلاس **EmployeeService**، اینترفیس **IEmployeeService** را پیاده‌سازی می‌کنیم.

موضوع اصلی مطرح شده در اینجا، این است که برای ارتباط با دیتابیس، باید یک نمونه از **DbContext** را در **Constructor** سرویس خود **Inject** نمایید.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using MicrodevProject.Data;
using MicrodevProject.Models;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;

namespace MicrodevProject.Services
{
    public class EmployeeService : IEmployeeService
    {
        private readonly MicrodevDbContext _context; ← | دریافت Context از Constructor
        public EmployeeService(MicrodevDbContext context)
        {
            _context = context; ← | قرار دادن مقدار متغیر Context در یک متغیر ReadOnly
        }

        public async Task<IEnumerable<Employee>> GetAllEmployeesAsync()
        {
            return await _context.Employees.Include(x => ← | برگرداندن تمام Employee ها.
                x.Department).ToListAsync();
        }

        public async Task<Employee> GetEmployeeAsync(int? id)
        {
            return await _context.Employees.FirstOrDefaultAsync(m => m.EmployeeId ← | یافتن اولین Employee که با Id
                == id); ← | ورودی ما برابر باشد.
        }

        public IQueryable<SelectListItem> GetDropDownEmployees()
        {
            return _context.Employees.Select(x => new SelectListItem { Value = ← | برگرداندن لیستی از نام کارمندان
                x.DepartmentId.ToString(), Text = x.Name });
        }

        public IQueryable<SelectListItem> GetDropDownDepartments()
    }
}
```

```

{
    return _context.Departments.Select(x => new SelectListItem { Value =
        x.DepartmentId.ToString(), Text = x.Name });
}
}

```

برگرداندن لیستی از نام شرکت‌ها

```

public async Task<Employee> AddAsync(Employee employee)
{
    _context.Employees.Add(employee);
    await _context.SaveChangesAsync();
    return employee;
}

```

افزودن Employee در یافتی به جدول Employees ذخیره تغییرات در دیتابیس

```

public async Task<Employee> UpdateAsync(Employee employee)
{
    _context.Update(employee);
    await _context.SaveChangesAsync();
    return employee;
}

```

Update Employee در یافتی، ذخیره تغییرات در دیتابیس یافتن Employee که با id ورودی ما برابر باشد

```

public async Task DeleteConfirmedAsync(int id)
{
    var employee = await _context.Employees.FindAsync(id);
    _context.Employees.Remove(employee);
    await _context.SaveChangesAsync();
}
}
}

```

حذف Employee از جدول Employees ذخیره تغییرات در دیتابیس

ما، جهت استفاده از سرویس‌های **EmployeeService**، نباید به طور مستقیم از این کلاس **Instance Controller** ایجاد کند زیرا کدهای **Tightly Coupled** به وجود می‌آید. برای حل این مسئله بهتر است **IEmployeeService/EmployeeService** را با سرویس **DI** در **Startup** رجیستر نماییم تا از این پس **Controller** تنها با اینترفیس **IEmployeeService** در ارتباط باشد.

کلاس **Startup.cs** را باز کنید و در متد **ConfigureServices** دستور زیر را وارد نمایید.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MicrodevDbContext>(c =>
        c.UseSqlServer(configuration.GetConnectionString("MicrodevConnection")));
}

```

رجیستر کردن سرویس EmployeeService با interface IEmployeeService

```

services.AddTransient<Seeder>();
services.AddScoped<IEmployeeService, EmployeeService>();
services.AddMvc();

```

من در بالا سرویس **EmployeeService** را به صورت **Scoped** رجیستر نمودم، زیرا **DbContext** در واقع به صورت **Thread-safe** نیست؛ در اینحالت مطمئن خواهیم شد که **DbContext** تنها درون یک **Thread** در دسترس قرار خواهد گرفت.

نکته!!

یکی از مزیت‌های رجیستر توسط DI این است که، اگر بعدها بخواهیم پیاده‌سازی دیگری از این اینترفیس داشته باشیم، و در **Controller** از این پیاده‌سازی جدید استفاده نماییم، **Controller** دیگر درگیر تغییر کدهای خود نخواهد بود و در اینجا پیاده‌سازی دوم برای این اینترفیس رجیستر می‌شود.

برای مثال:

به جای رجیستر پایین:

```
services.AddScoped<IEmployeeService, EmployeeService>();
```

پیاده‌سازی دوم این اینترفیس نوشته می‌شود.

```
services.AddScoped<IEmployeeService, TestService>();
```

حالا **Controller** با تزریق **IEmployeeService**، دیگر با کلاس **EmployeeService** کاری ندارد و از سرویس-های **TestService** را استفاده می‌کند.

Controller چیست؟

Controller در **ASP.NET.Core** کلاسی است که می‌تواند، تعدادی **Action Method** داشته باشد و این **Action Method**ها می‌توانند به صورت منطقی با هم در ارتباط باشند.

هنگامی که یک درخواست به **MVC Middleware** فرستاده می‌شود، یک **Instance** جدید از **Controller** ساخته شده و **Action Method** آن صدا زده می‌شود.

قراردادهای استفاده از **Controller**ها به شرح زیر می‌باشد:

- **Controller**ها باید با نام **Controller** خاتمه یابند.
- و از کلاس **Controller** یا **ControllerBase** ارث‌بری کنند.(یا هر کلاسی که از این دو کلاس ارث‌بری کرده است).

MvcMiddleware هر کلاسی را که مطابق این شرایط باشد، در زمان اجرا شناسایی می‌کند و در زمان **Handle** شدن **Request**ها، آن را در دسترس قرار می‌دهد.

توجه داشته باشید ارث‌بری از کلاس **Controller** اجباری نیست، ولی بهتر است که تمام **Controller**های ما از این کلاس ارث‌بری کنند، زیرا این کلاس، متدهای کمکی زیادی در اختیار ما قرار می‌دهد.

برای داشتن **Controller** در پروژه، باید پوشه‌ای به نام **Controllers** ایجاد نمایید، اما قبل از نوشتن اولین **Controller** اجازه دهید تا کمی در مورد **Layout** صحبت کنم.

Layout چیست؟

هر **HTML Document**، دارای المنت‌های تکراری **<Head>**، **<body>**، **<HTML>** و در برخی صفحات **Header**، **Footer** می‌باشد. و احتمالاً هر صفحه از برنامه شما به فایل‌های **jQuery**، **Bootstrap** هم نیاز دارد.

اگر این عناصر را در هر صفحه به صورت جداگانه اضافه نمایید، خطایابی و نگهداری پروژه به یک کابوس تبدیل خواهد شد، زیرا تغییرات، دشوار و **خطایابی**، سخت خواهد شد.

استفاده از **Layout** این مشکل را برای ما حل کرده است. **Layout** یک صفحه **HTML** است که می‌توانید عناصر مشترک تمام صفحات را در آن قرار دهید و از تکرار این عناصر در هر صفحه به صورت جداگانه خودداری نمایید.

با **Layout**، مدیریت برنامه ساده‌تر می‌شود و شاید بخواهید، برای هر بخش، قالب متفاوتی در نظر بگیرید، در این صورت می‌توانید **Layout**های متفاوتی داشته باشید. به صورت قراردادی **Layout**ها در یک پوشه‌ای به نام **Shared** قرار می‌گیرند و نام **Layout** با کاراکتر _ شروع می‌شود.

یک **Layout** شبیه یک **Razor Template** است، با این تفاوت که هر **Layout** باید یک تابع **@RenderBody()** داشته باشد تا **Child View**ها را در بدنه قالب ما نمایش دهد.

پوشه‌ای به نام **Sharad** در **Folder→View** ایجاد و یک **View** به نام **Layout_** در آن قرار دهید. حالا کدهای زیر را در آن کپی نمایید.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>@ViewData["Title"] - MicrodevProject</title>
  <link rel="stylesheet"
  href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
  >

```

MetaTagهای موردنیاز

ViewData مکانیزم استاندارد است، برای پاس دادن داده، از View به Layout

Bootstrap پروژه که در سراسر پروژه استفاده می‌شود.

```

integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPM0"
crossorigin="anonymous">

</head>
<body>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"
  integrity="sha384-ChfqquxZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy"
  crossorigin="anonymous"></script>
  <header>
    <nav class="navbar navbar-expand-sm navbar-togglerable-sm navbar-light bg-white
    border-bottom box-shadow mb-3">
      <div class="container">
        <a class="navbar-brand" asp-area="" asp-controller="Employee" asp-
        action="Index">MicrodevProject</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
        data-target=".navbar-collapse" aria-
        controls="navbarSupportedContent"
        aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-
        reverse">
          <ul class="navbar-nav flex-grow-1">
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-
              controller="Employee" asp-action="Index">کارمند</a>
            </li>
            <li class="nav-item">
              <a class="nav-link text-dark" asp-area="" asp-
              controller="Department" asp-
              action="Privacy">دپارتمان</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </header>
  <div class="container">
    <main role="main" class="pb-3">
      @RenderBody()
    </main>
  </div>
  <footer class="border-top footer text-muted">
    <div class="container">
      &copy; 2019 - MicrodevProject - Zahra Bayat
    </div>
  </footer>
</body>
</html>

```

افزودن Bootstrap

navbar مشترک صفحات

به Template Engine می گوید که محتوای @RenderBody() Child View ها باید کجا قرار گیرد.

Footer مشترک صفحات

بعد از اضافه کردن این Layout در Folder ← Controller، یک کلاس EmployeeController ایجاد نمایید.

```
using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc;

namespace MicrodevProject.Controllers
{
    public class EmployeeController: Controller
    {
    }
}
```

ارث‌بری از کلاس Controller
قابلیت‌های زیادی به Controller اضافه می‌کند.

اکنون نیاز است بعد از رجیستر شدن **IEmployeeService** به عنوان سرویسی که می‌تواند از طریق **DI** ← **Inject** شود، این سرویس را به **Controller** خود **Inject** نمایید.

در این **Controller** ابتدا باید یک **Constructor** ایجاد نمایید و سپس اینترفیس **IEmployeeService** را به آن **Inject** کنید.

نکته!!

در این حالت **Controller** هیچ اطلاعی از کلاس **EmployeeService** ندارد و تنها با اینترفیس **IEmployeeService** سروکار دارد.

```
using MicrodevProject.Services;
using Microsoft.AspNetCore.Mvc;

namespace MicrodevProject.Controllers
{
    public class EmployeeController: Controller
    {
        private readonly IEmployeeService service;

        public EmployeeController(IEmployeeService service)
        {
            this.service = service;
        }
    }
}
```

به **IEmployeeService**
EmployeeController تزریق و
سپس مقدار آن درون یک متغیر
ReadOnly ریخته شده است.

نمایش لیست کارمندان

اولین قدم نمایش لیست تمام کارمندان است، بنابراین در کلاس **EmployeeController**، متدی با خروجی از نوع **ActionResult**، برای نمایش اطلاعات کارمندان می‌نویسیم.

```
using MicrodevProject.Services;
using MicrodevProject.ViewModels;
```



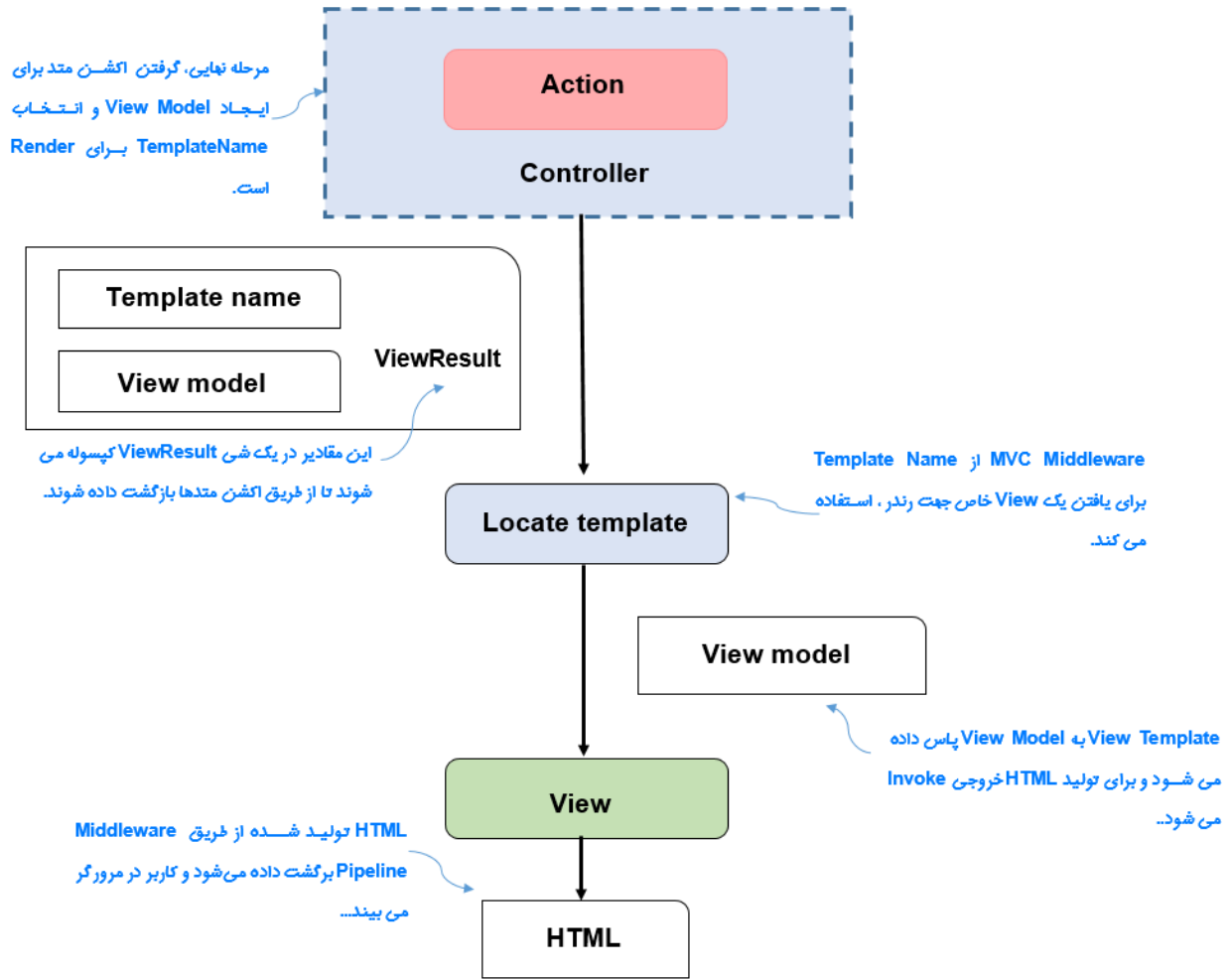
```
using Microsoft.AspNetCore.Mvc;

namespace MicrodevProject.Controllers
{
    public class EmployeeController: Controller
    {
        private readonly IEmployeeService service;

        public EmployeeController(IEmployeeService service)
        {
            this.service = service;
        }
        public async Task<IActionResult> Index()
        {
            var employees = await _service.GetAllEmployeeysAsync();
            return View(employees);
        }
    }
}
```

پاس دادن نتیجه به متد View

دریافت لیست کارمندان از سرویس



نکته!!

شما می‌توانید خروجی متد را هر آنچه که نیاز دارید قرار دهید، اما در MVC مرسوم است که خروجی Action Method ها از نوع IActionResult باشد.

Razor چیست؟

در MVC یکی از رایج‌ترین روش‌های تولید HTML، استفاده از Razor view engine است. در ASP.NET Core، معمولاً View با استفاده از Razor ایجاد می‌شود. Razor ترکیبی از HTML و C# است، که در آن نتیجه نهایی را تولید می‌کند و HTML آن، این امکان را به شما می‌دهد تا به سادگی چیزی که باید به مرورگر ارسال شود را دقیقاً مشخص کنید.

به طور مثال:

```
@{
    var names = new List<string> {"Zahra", "Ali", "Sara"}
}
<h1>Names: </h1>
<ul>
    @for (int i = 0; i < names.Count; i++)
    {
        var name = names[i];
        <li>@i - @name</li>
    }
</ul>
```

در اینجا لیستی از نام اشخاص را درون متغیری قرار دهیم.

ترکیب HTML، C# این امکان را می‌دهد که به صورت داینامیک در زمان Runtime بتوانیم HTML تولید کنیم.

در مثال بالا Razor مستقیماً، HTML را همانطور که هست به خروجی کپی می‌کند. کدهای HTML در مثال بالا بین <> قرار دارند و تعدادی عبارات سی‌شارپ می‌بینید که با کدهای HTML، نتیجه موردنظر شما را تولید می‌کند.

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

(۱) آن‌هایی که تنها، داده‌های نمایش داده شده در برنامه شما را می‌خوانند.

(۲) آن‌هایی که داده‌هایی را به برنامه شما ارسال می‌کنند.

زبان Razor حاوی تعدادی سازنده است که ساخت هر دو نوع اپلیکیشن را ساده‌تر می‌کند. با استفاده از Razor، می‌توانید HTML را با مقداری از ViewModel ترکیب کنید و داده‌ها را نمایش دهید.

Razor می تواند از C# به عنوان مکانیزم کنترلی استفاده کند. بنابراین اضافه کردن، المنت های شرطی و حلقه ها، بسیار ساده می شود.

چیزهای که با HTML نمی توان به آن رسید، با Razor به راحتی می تواند به واقعیت پیوندد.

Tag Helper چیست؟

یک روش معمول برای ارسال داده ها به وب اپلیکیشن، استفاده از HTML Form ها است. تقریباً هر برنامه دینامیکی که ایجاد می شود از فرم ها استفاده می کند.

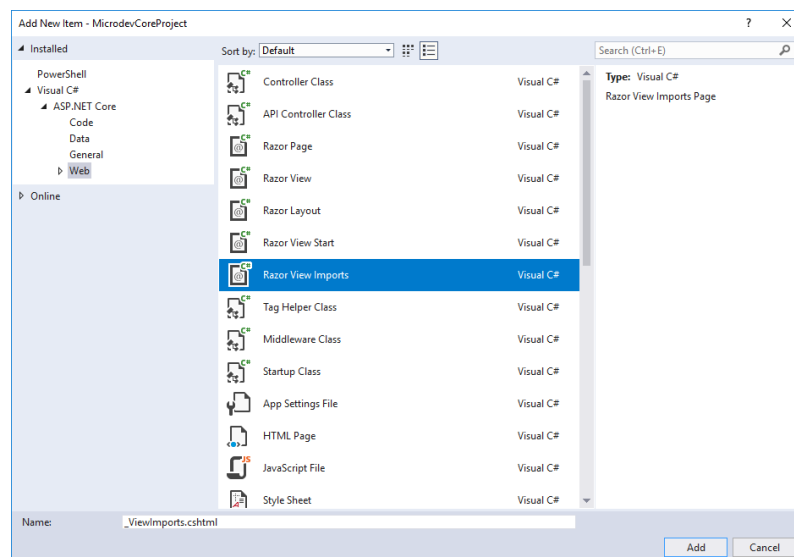
فرم ها یک روش کلیدی، برای برقراری ارتباط کاربران با اپلیکیشن، در مرورگر است. پس مهم است که فرم ها هم برای اپلیکیشن شما، درست و کاربرپسند تعریف شوند.

ASP.NET Core برای رسیدن به این هدف، قابلیتی به نام Tag Helper اضافه کرده است. Tag Helper ها، Razor Component هایی هستند که می توانید از آنها برای سفارشی کردن کدهای HTML استفاده نمایید.

Tag Helper ها می توانند با هدف سفارشی کردن ویژگی های HTML، به یک المنت (مانند <Input>) اضافه شده و باعث صرفه جویی زمان شوند.

برای لینک بین صفحات، یکی از بهترین راهکارها، استفاده از Tag Helper است.

جهت اضافه کردن Tag Helper روی Folder Views ← راست کلیک کنید و Add → New Item را انتخاب نموده و سپس Razor View Import را برگزینید.

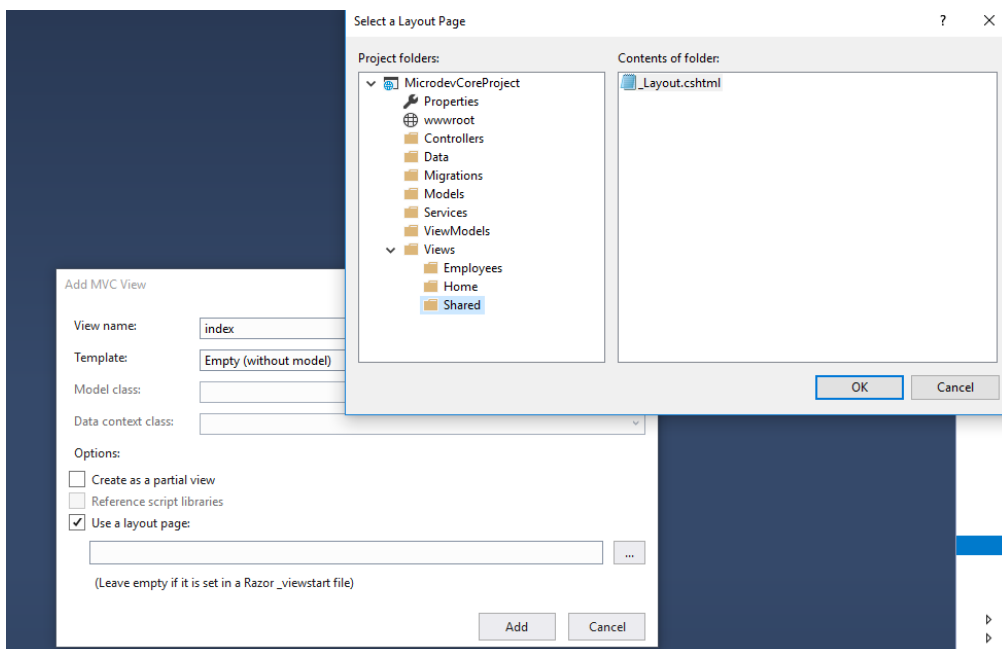
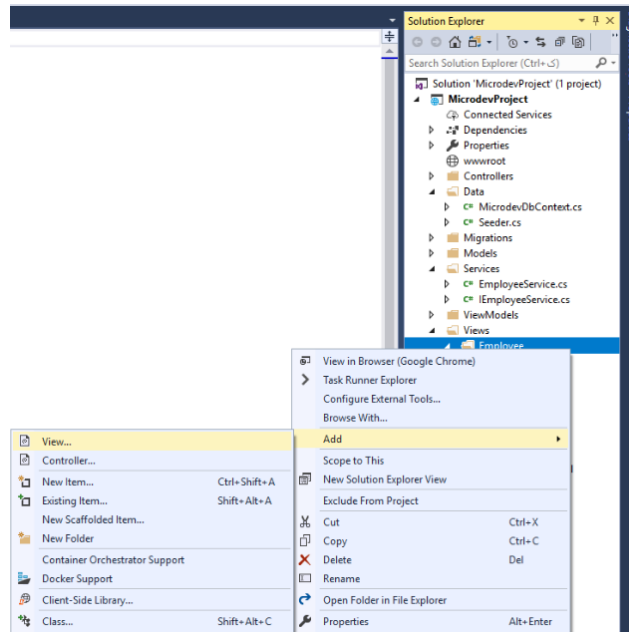


حالا، کد Tag Helper زیر را به این فایل اضافه نمایید.

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

خوشحالم که با هم توانستیم Tag Helper را به این پروژه اضافه نمایم بیاید به پروژه CRUD برگردیم...

پوشه‌ای به نام Views ایجاد کنید و درون آن پوشه‌ی دیگری به نام Employee و درون این پوشه آیتم index.cshtml را اضافه نمایید.



برای نمایش اطلاعات کارمندان، کد زیر را در این View قرار دهید.

```

@model MicrodevProject.ViewModels.EmployeeViewModel ;
@{
    ViewData["Title"] = "کارمندان";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<div dir="rtl">
    <h1 class="d-flex justify-content-center">لیست کارمندان</h1>
</div>
<p>
    <a class="btn btn-success" asp-action="Create">کارمند افزودن</a>
</p>
<table class="table" dir="rtl">
    <thead >
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Salary)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.BossId)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.DepartmentId)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Salary)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
            </tr>
        }
    </tbody>
</table>

```

استفاده از @model جهت توصیف ViewModel Type برای TagHelper است.

Layout برای این View در نظر گرفته می شود.

نمایش عناوین Property ها در کلاس Employee

ساختار foreach سی شارپ می تواند با Razor همراه باشد.

می توانید مقادیر Employee را در المنت‌های سی شارپ نمایش دهید.

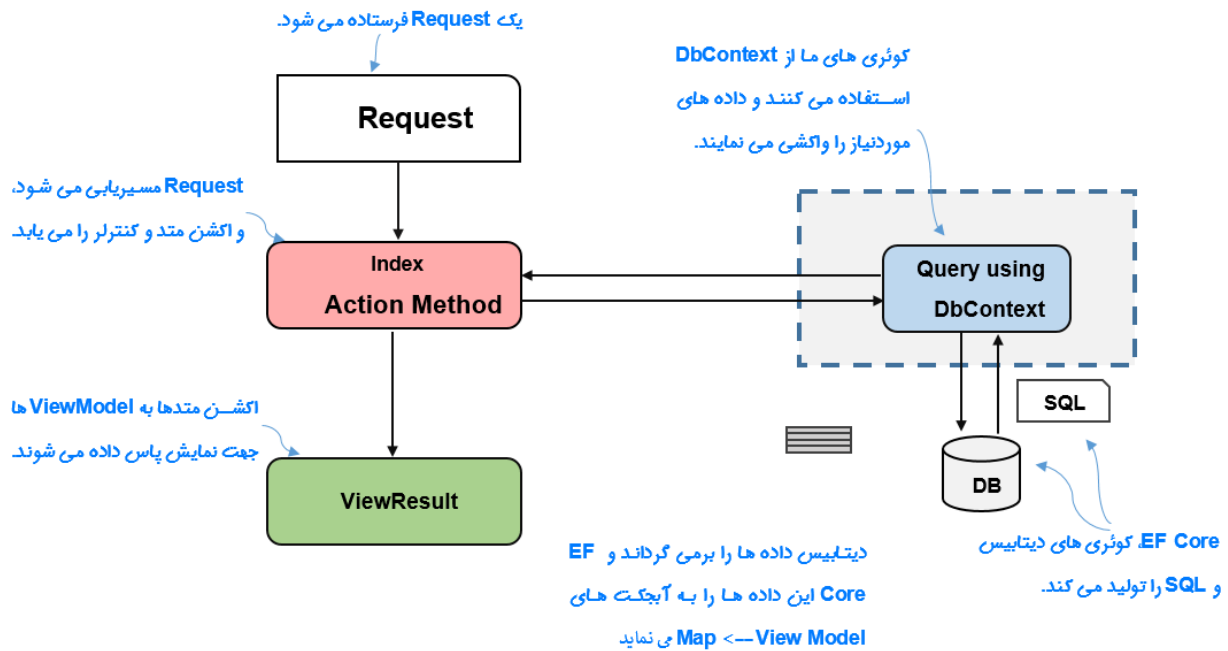
```

<td>
<a class="btn btn-primary" asp-action="Details" asp-route-
id="@item.EmployeeId">جزئیات</a> |
<a class="btn btn-info" asp-action="Edit" asp-route-
id="@item.EmployeeId">ویرایش</a> |
<a class="btn btn-danger" asp-action="Delete" asp-route-
id="@item.EmployeeId">حذف</a>
</td>
</tr>
}
</tbody>
</table>

```

استفاده از TagHelper برای مسیریابی و تولید کدهای HTML

بعد از زدن کلید F5 لیستی از اطلاعات کارمندان را می بینید.



| لیست کارمندان | | | |
|--------------------|--------------|---------------|-----------------------|
| نام و نام خانوادگی | حقوق دریافتی | BossId | افزودن کارمند |
| زهرا بیات | 1,000,000 | مدیریت روشمند | حذف ویرایش جزئیات |
| علی بیات | 3,000,000 | دکان باز | حذف ویرایش جزئیات |
| امین مزگانی | 1,000,000 | مدیریت روشمند | حذف ویرایش جزئیات |
| محمد سهیلی | 4,000,000 | مدیریت روشمند | حذف ویرایش جزئیات |
| سارا احمدی | 1,000,000 | دکان باز | حذف ویرایش جزئیات |
| محمد سعیدی | 1,000,000 | دکان باز | حذف ویرایش جزئیات |
| سهیلا افتخاری | 2,000,000 | دکان باز | حذف ویرایش جزئیات |
| سعید حسینی | 5,000,000 | ایده پرداز | حذف ویرایش جزئیات |
| زهرا محمدی | 1,000,000 | ایده پرداز | حذف ویرایش جزئیات |

© 2019 - MicrodevProject - Zahra Bayat

ViewImports و ViewStart چیست؟

با توجه به ماهیت **View**، گاهی مجبور هستیم مثلاً آدرس یک **View** را در **View**های دیگر تکرار کنیم. (به طور مثال تعریف **Layout** در هر **View**) یا دائماً باید در صفحات مختلف یک **Namespace** تکراری، وارد نماییم.

خوشبختانه **NET Core** دو مکانیزم برای حل این مسائل بیان کرده است:

- **_ViewImports.cshtml**
- **_ViewStart.cshtml**

هر **View** شامل عباراتی تکراری مانند **@model** , **@using** می‌باشد، فایل **ViewImports.cshtml** برای پرهیز از نوشتن **Using** تکراری در بالای هر **View**، مورد استفاده قرار می‌گیرد.

این فایل را می‌توان در هر **Folder**ای قرار داد (معمولاً در **Folder** ریشه قرار می‌گیرد) و به تمام **View**ها اعمال نمود.

توجه داشته باشید : تنها دستورات رزرو شده را در این فایل قرار دهید. این فایل به **@using** و **@addTagHelper** محدود می‌شود و مجاز به نوشتن هرگونه عبارات سی‌شارپی نخواهید بود.

_ViewImports.cshtml

- **@using MicrodevProject** ← Namespace پیش‌فرض اپلیکیشن
- **@using MicrodevProject.Models** ← نوشتن این Namespace در فایل، یعنی در هر **View**، این Namespace در دسترس است.
- **@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers** ← اضافه شدن **TagHelper** برای تمام **View**ها

اما اگر می‌خواهید **Layout** پیش‌فرض را برای هر **View**، تکرار نکنید، می‌توانید یک فایل **_ViewStart** به پروژه اضافه نمایید و مانند تصویر زیر **Layout** را در آن قرار دهید.

_ViewStart.cshtml

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

فریم‌ورک **MVC** قبل از اجرای کدهای درون **View**ها، ابتدا محتویات **View**یی با نام **_ViewStart.cshtml** را اجرا خواهد کرد، پس هر کدی که در این فایل باشد، قبل از اجرای تمام **View**ها اجرا خواهد شد.

بعد از اضافه نمودن **Layout** در این فایل، دیگر نیازی به تعیین **Layout** در **View**ها نخواهد داشت.

نکته!!

اگر فایل **_ViewStart.cshtml** را درون پوشه‌ی **Home** قرار دهیم؛ **Layout** تنها برای **View**های **Home** در نظر گرفته خواهد شد. اما با قرار دادن این فایل درون پوشه‌ی **Views**، **Layout** برای تمامی **View**ها تنظیم خواهد شد.

به پروژه برگردیم....

جزئیات کارمند

برای نمایش جزئیات کارمند، ابتدا یک **View** به نام **Detail** ایجاد کنید.

```
@model MicrodevProject.Models.Employee
```

```
@{
    ViewBag.Title = "کارمند جزئیات";
}
```

```
<div class="offset-4">
<h1 class="d-flex justify-content-center">کارمند جزئیات</h1>
<table dir="rtl" class="table">
<tr>
<th> @Html.DisplayNameFor(model => model.Name) </th>
<td> @Html.DisplayFor(model => model.Name)</td>
</tr>
<tr>
<th> @Html.DisplayNameFor(model => model.Department.Name)</th>
<td> @Html.DisplayFor(model => model.Department.Name)</td>
</tr>
<tr>
<th> رئیس نام </th>
<td> @Html.DisplayFor(model => model.Boss.Name)</td>
</tr>
```

نمایش جزئیات کارمند →


```

        <tr>
            <th> @Html.DisplayNameFor(model => model.Salary) </th>
            <td> @Html.DisplayFor(model => model.Salary)</td>
        </tr>
    </table>
    </div>

```

مسیریابی بین صفحات
TagHelper ها برای

برای ادامه کار باید با نحوه ایجاد URL برای Action آشنا شوید.

ایجاد URL برای Action

گاهی اوقات پیش می آید که بخواهید یک URL را به عنوان بخشی از نتیجه ی Action Method برگردانید. این اتفاق، معمولاً زمانی می افتد که بخواهید یک مرورگر را به یک Action یا Route خاص، هدایت کنید. در کد زیر دو روش برای تولید URL نوشته شده است.

▪ روش اول:

```
RedirectToAction("Details", "Employee", new { id = 5 });
```

این متد یک RedirectToActionResult با یک URL تولید می کند. RedirectToAction نام Action Method، Controller، و پارامترهای Route را می گیرد و یک URL را به همان روش Url.Action تولید می کند.

▪ روش دوم:

```
RedirectToRoute("Details_Employee", new { id = 5 });
```

این متد یک RedirectToRouteResult با یک URL تولید می کند. RedirectToRoute هم معادل Url.RouteUrl است.

دانستن این مطالب به ادامه بحث ما کمک بسیاری می کند....

حالا نوبت به نوشتن متد Details می رسد. در EmployeeController متد زیر را اضافه نمایید.

```

public async Task<IActionResult> Details(int? id)
{
    if (id == null)
        return NotFound();

    Employee employee = await _service.GetEmployeeAsync(id);
    if (employee == null)
        return NotFound();
}

```

گرفتن Id کارمند از ورودی

اگر Id=null باشد، صفحه ی NotFound نمایش داده خواهد شد.

اگر Id= null نبود، این Id را به متد GetEmployeeAsync سرویس ما می دهد تا کارمند موردنظر را از دیتابیس واکنش کند.

اگر کارمندی با این Id وجود نداشته، صفحه NotFound نمایش داده می شود

```
return NotFound();

return View(employee);
}
```

وگرنه کارمند واکنشی شده را به View موردنظر می‌فرستد.

بعد از اجرا، اگر در صفحه‌ی لیست کارمندان بر روی جزئیات کارمند کلیک کنید وارد صفحه‌ی جزئیات کارمند خواهید شد.



همانطور که در تصویر بالا می‌بینید، ما می‌توانیم یک کارمند را ویرایش یا حذف نماییم.

نوبت به نوشتن متد ویرایش می‌رسد.

دوباره به **EmployeeController** برمی‌گردیم و اینبار متد **Edit** را مانند کد زیر می‌نویسیم.

باید دو متد **Edit** ایجاد شود یکی برای فعل **Get** (زمانیکه وارد صفحه ویرایش کارمند می‌شویم) و دیگری برای

فعل **Post** (و زمانیکه بر روی ذخیره کارمند کلیک می‌کنیم این متد صدا زده می‌شود).

```
متد Get: گرفتن Id کارمند از ورودی
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
        return NotFound();
    Employee employee = await _service.GetEmployeeAsync(id);
}
```

اگر Id=null نبود، این Id را به متد **GetEmployeeAsync** سرویس ما می‌دهد تا کارمند موردنظر را از دیتابیس واکنشی کند.

اگر این Id=null باشد، صفحه **NotFound** نمایش داده می‌شود.

```

    if (employee == null)
    {
        return NotFound();
    }
    ViewBag.Bosses = _service.GetDropDownEmployees();
    ViewBag.Departments = _service.GetDropDownDepartments();
    return View(employee);
}

```

دریافت لیست Employee و Department از سرویس و قرار دادن نتیجه در دو متغیر ViewBag.Bosses, ViewBag.Department

اگر کارمندی با این Id وجود نداشته باشد، صفحه NotFound نمایش داده می شود.

کارمند واکنشی شده را به View موردنظر می فرستد.

بعد از نوشتن متد بالا در EmployeeController، باید در Folder → View → Employee یک View به نام Edit ایجاد نمایید و کدهای نمایش جزئیات کارمند را در آن قرار دهید.

@model MicrodevProject.Models.Employee

```

@{
    ViewData["Title"] = "کارمند ویرایش";
}

```

```

<h1 class="d-flex justify-content-center">کارمند ویرایش</h1>
<hr />
<div class="row" dir="rtl">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="EmployeeId" />

            <div class="form-group">
                <label asp-for="Name" class="control-label" dir="rtl"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Salary" class="control-label"></label>
                <input asp-for="Salary" class="form-control" />
                <span asp-validation-for="Salary" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label class="control-label">نام رئیس</label>
                <select asp-for="BossId" asp-items="@ViewBag.Bosses" class="form-control"></select>
                <span asp-validation-for="BossId" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label class="control-label">نام شرکت</label>

```

TagHelperها در Formها برای تولید Action URLها استفاده می شود.

asp-for در Label برای تولید برچسب هایی بر مبنای Model View می باشد.

asp-for در input برای تولید Typeها، Value و Validation Attribute برای Model است.

TagHelperها در یک Span نوشته می شوند.

```

    <select asp-for="DepartmentId" asp-items=@ViewBag.Departments
    class="form-control"></select>
    <span asp-validation-for="DepartmentId" class="text-
    danger"></span>
    </div>
    <div class="form-group">
    <input type="submit" value="ذخیره" class="btn btn-success" />
    </div>
</form>
</div>
</div>
</div>
<div>
    <a class="btn btn-info" asp-action="Index">اصلی صفحه</a>
</div>

```

حالا اگر بر روی ویرایش کارمند کلیک کنید (مانند تصویر پایین) وارد فرمی خواهید شد، که می‌توانید مشخصات کارمند را تغییر دهید.

توجه داشته باشید: هنوز Button ذخیره، هیچ اطلاعاتی را بروزسانی نمی‌کند.

با ما همراه باشید تا متد ویرایش اطلاعات را با هم بنویسیم.

The screenshot shows a web application interface for editing an employee. At the top, there's a breadcrumb: "MicrodevProject" > "کارمند" > "دبارتمان". The main heading is "ویرایش کارمند". The form contains the following fields:

- نام و نام خانوادگی: زهرا بیات
- حقوق دریافتی: 1000.00
- نام رئیس: امین مزگانی
- نام شرکت: مدیریت روشمند

At the bottom of the form is a green "ذخیره" (Save) button. In the bottom left corner, there is a blue button labeled "صفحه اصلی" (Home page). The footer contains the text: "© 2019 - MicrodevProject - Zahra Bayat".

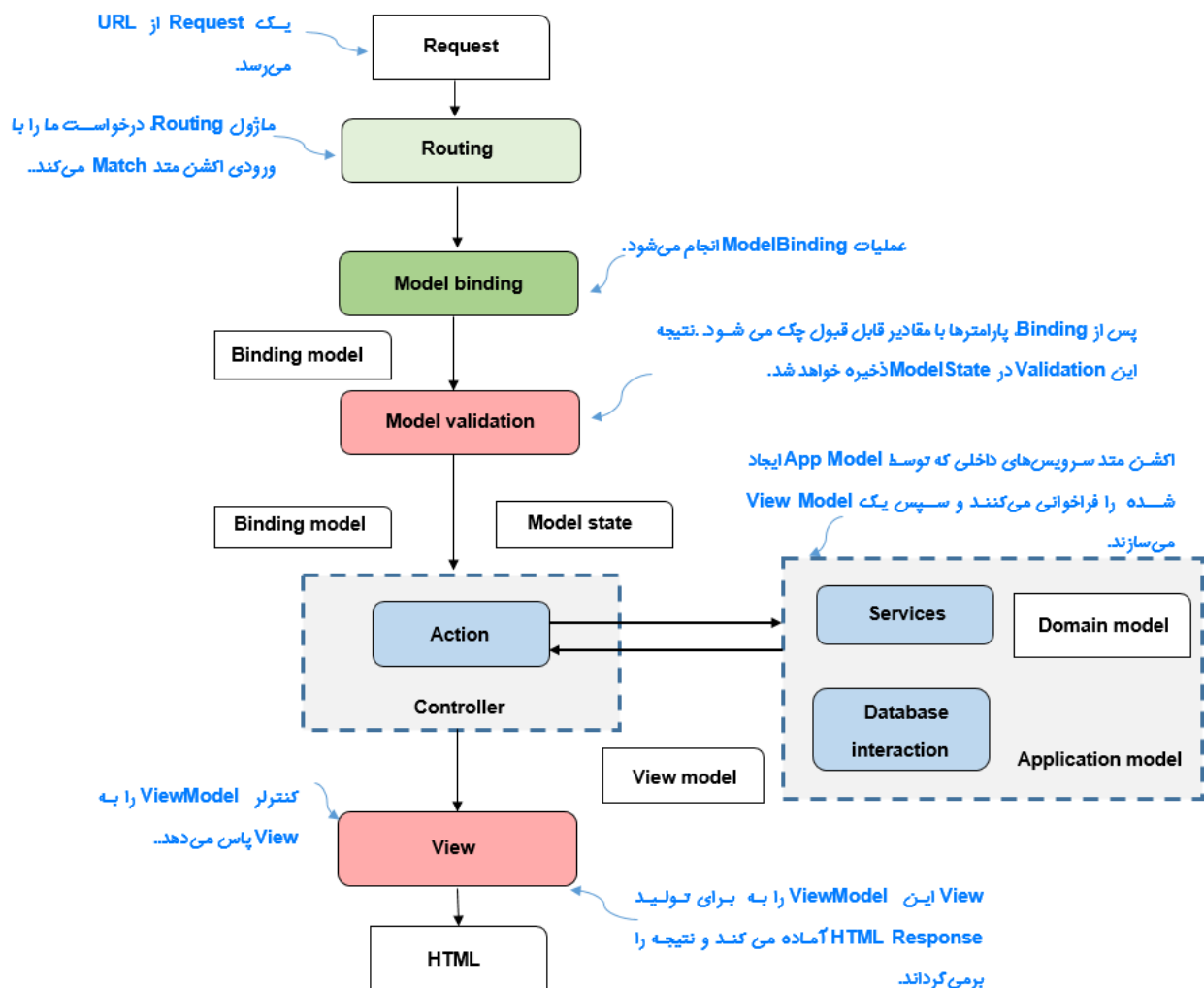
Model Binding چیست؟

در فصل‌های قبل با Route و ارسال پارامتر آشنا شدید، شاید این سوال در ذهنتان مطرح شود که با کلیک بر روی ذخیره اطلاعات، چگونه اطلاعات فرم به آبجکت ما فرستاده می‌شود؟

اطلاعاتی که در هنگام ایجاد یک Request توسط کاربر، به صراحت درخواست می شود، از طریق Model Binding به آبجکت های شما در Action Method بایند می شود.

این همان چگونگی پذیرش داده های ارسال شده، توسط کاربر و تبدیل آن به یک شی سی شارپ است. شما می توانید از Model Binding در Action Method ها استفاده نمایید.

Model Binding قبل از Action Method اجرا می شود و بررسی می کند: آیا مقادیر اتصالی معتبر است یا خیر؟



Validation بر روی Server

اعتبارسنجی Model Binding، قبل از اجرای Action رخ می دهد اما توجه داشته باشید، که Action ها در هر صورت (چه Validate موفق باشد چه نباشد) اجرا خواهند شد.

اعتبارسنجی به صورت خودکار اتفاق می افتد اما مدیریت داده های معتبر بر عهده Action Method است.

MvcMiddleware خروجی نتیجه اعتبارسنجی را، در یک **Property** به نام **ModelState** (موجود در کلاس **ControllerBase**) ذخیره می‌کند. این شی یک **ModelStateDictionary** است که حاوی لیستی از تمام خطاهای اعتبارسنجی، پس از اتصال به مدل می‌باشد.

حالا نوبت به نوشتن کدی برای ذخیره اطلاعات بروز شده‌ی کارمند می‌رسد.

ما یک متد **Edit** دیگر با فعل **[HttpPost]** نیازمندیم. به **EmployeeController** برمی‌گردیم و کدهای ذخیره اطلاعات را در متد **Edit** قرار می‌دهیم.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, Employee employee)
{
    if (id != employee.EmployeeId)
        return NotFound();

    if (ModelState.IsValid)
    {
        await _service.UpdateAsync(employee);
        return View(employee);
    }
    else
    {
        return RedirectToAction(nameof(Index));
    }
}
```

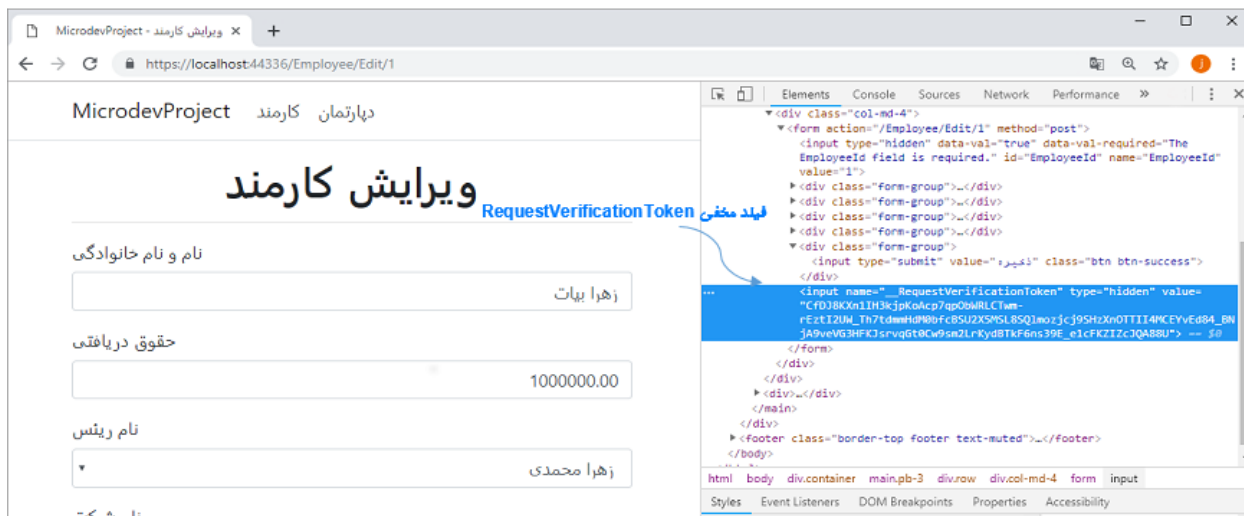
اگر Id دریافتی با EmployeeId شی Employee ما برابر نباشد، صفحه‌ی NotFound نمایش داده می‌شود.

اگر مدل ازسالی ما معتبر باشد، متد UpdateAsync سرویس صدا زده می‌شود و Employee ما در دیتابیس Update می‌شود و View به employee به پاس داده خواهد شد.

در غیر این صورت به صفحه‌ی اصلی Redirect می‌شود.

چيست ValidateAntiForgeryToken

درون فرم‌های ما فیلد مخفی با نام **RequestVerificationToken** وجود دارد که مقدار این فیلد تضمین خواهد کرد که اطلاعاتی که به ما رسیده، از طرف فرم کاربر است. درحقیقت داشتن این فیلد به ما کمک می‌کند تا از وقوع حمله‌ی **Cross Site Request Forgery (CSRF)** جلوگیری کنیم.



نکته!!

برای استفاده از این قابلیت حتما باید [ValidateAntiForgeryToken] را بالای فرم‌های Post خود قرار دهید. اگر از این Attribute استفاده نکنید، به راحتی توسط هر ابزاری (مثلا Postman) می‌توان یک فرم را به سرور ارسال نمود.

متد حذف کارمند

برای حذف کارمند هم، باید دو فعل **Get** و **Post** صورت پذیرد. پس به **EmployeeController** برمی‌گردیم و ابتدا فعل **Get** را برای متد حذف کارمند اضافه می‌نماییم.

```

public async Task<IActionResult> Delete(int? id)
{
    if (id == null) ← | اگر Id = null بود، صفحه NotFound نمایش داده خواهد شد.
        return NotFound();

    Employee employee = await _service.GetAllEmployeesAsync(id); ← | وگرنه Id دریافتی به متد GetAllEmployeesAsync کلاس سرویس پاس داده می‌شود و یک Employee بازگشت داده خواهد شد.

    if (employee == null) ← | اگر Employee = null بود، صفحه NotFound نمایش داده شود.
        return NotFound();

    return View(employee); ← | وگرنه Employee به View پاس داده می‌شود.
}
    
```

MicrodevProject دپارتمان کارمند

حذف کارمند

آیا از حذف این کارمند مطمئن هستید؟

| | |
|-----------|--------------------|
| 1 | کد کارمندی |
| زهرا بیات | نام و نام خانوادگی |

صفحه اصلی
حذف کارمند

© 2019 - MicrodevProject - Zahra Bayat

فعل **Post** برای عملیات حذف کارمند:

حالا باید کدی بنویسیم که با کلیک بر روی حذف کارمند، عملیات حذف انجام شود.

به **EmployeeController** برمی گردیم و متد زیر را اضافه می نماییم.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    await _service.DeleteConfirmedAsync(id);
    return RedirectToAction(nameof(Index));
}
```

پاس دادن id ورودی به متد DeleteConfirmedAsync کلاس سرویس و حذف کارمند از دیتابیس. بازگشت به صفحه اصلی.

حالا اگر بر روی حذف کارمند کلیک کنید، این کارمند حذف خواهد شد.

درج کارمند جدید

آخرین مطلب پروژه **CRUD** درج کارمند جدید می باشد.

به **Employee Controller** برمی گردیم و اینبار متد **Create** را ایجاد می کنیم.

ما باید دو متد **Create** داشته باشیم. یکی برای فعل **Get** و دیگری برای فعل **Post**

فعل **Get** برای متد **Create**:

```
public IActionResult Create()
{
    ViewBag.Bosses = _service.GetDropDownEmployees();
    ViewBag.Departments = _service.GetDropDownDepartments();
}
```



```
    return View();
}
```

این متد لیستی از کارمندان و شرکتها را از متدهای `GetDropDownDepartments` `GetDropDownEmployees` موجود در کلاس سرویس دریافت می کند و مقادیر را در متغیرهای `Department` , `Bosses` از نوع `ViewBag` قرار می دهد.

این مقادیر باید در `DropDown` های `View` نمایش داده شود.

یک `View` به نام `Create` در مسیر `View→Employee` ایجاد نمایید و کدهای زیر را جهت نمایش فرم درج کارمند به آن اضافه نمایید.

```
@model MicrodevProject.Models.Employee
```

```
@{
    ViewData["Title"] = "کارمند افزودن";
}
```

```
<h1 class="d-flex justify-content-center">کارمند افزودن</h1>
```

```
<hr />
```

```
<div class="row" dir="rtl">
```

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

```
        <form asp-action="Create">
```

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

```
            <input type="hidden" asp-for="EmployeeId" />
```

```
            <div class="form-group">
```

```
                <label asp-for="Name" class="control-label" dir="rtl"></label>
```

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

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

```
            </div>
```

```
            <div class="form-group">
```

```
                <label asp-for="Salary" class="control-label"></label>
```

```
                <input asp-for="Salary" class="form-control" />
```

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

```
            </div>
```

```
            <div class="form-group">
```

```
                <label class="control-label">نام رئیس</label>
```

```
                <select asp-for="BossId" asp-items=@ViewBag.Bosses class="form-control"></select>
```

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

```
            </div>
```

```
            <div class="form-group">
```

```
                <label class="control-label">شرکت نام</label>
```

```
                <select asp-for="DepartmentId" asp-items=@ViewBag.Departments class="form-control"></select>
```

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

```
            </div>
```

```
            <div class="form-group">
```

نمایش کارمندان در یک

DropDown

نمایش شرکتها در یک

DropDown

TagHelperها در Formها برای تولید URL Actionها استفاده می شود.

asp-for در input برای تولید Value.

Validation Attributes برای Model

است.

asp-for در Label برای تولید برچسب هایی بر مبنای View

Model می باشد.

```

        <input type="submit" value="ذخیره" class="btn btn-success" />
    </div>
</form>
</div>
</div>
<div>
    <a class="btn btn-info" asp-action="Index">اصلی صفحه</a>
</div>

```

بعد از اجرای برنامه بر روی درج کارمند کلیک کنید تا همانند تصویر زیر وارد فرم افزودن کارمند شوید.

نوبت به نوشتن فعل **POST** برای متد **Create** می‌رسد.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(Employee employee)
{
    if (ModelState.IsValid)
    {
        await _service.AddAsync(employee);
        return RedirectToAction(nameof(Index));
    }
    return View(employee);
}

```

در صورتی که مدل ارسالی **Validate** شده باشد، این **employee** دریافتی به دیتابیس اضافه می‌شود.

حالا اگر در این فرم، بر روی **Button** ذخیره کلیک کنید اطلاعات در دیتابیس ذخیره خواهد شد.

تمام کدهای **EmployeeController**

```
using System.Collections.Generic;
using System.Threading.Tasks;
using MicrodevCoreProject.Services;
using MicrodevProject.Models;
using Microsoft.AspNetCore.Mvc;

namespace MicrodevProject.Controllers
{
    public class EmployeeController : Controller
    {
        private readonly IEmployeeService _service;

        public EmployeeController(IEmployeeService service)
        {
            _service = service;
        }

        // GET: Employees
        public async Task<IActionResult> Index()
        {
            IEnumerable<Employee> employees = await
                _service.GetAllEmployeesAsync();
            return View(employees);
        }

        // GET: Employees/Details/5
        public async Task<IActionResult> Details(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Employee employee = await _service.GetEmployeeAsync(id);
            if (employee == null)
            {
                return NotFound();
            }

            return View(employee);
        }

        // GET: Employees/Create
        [HttpGet]
    }
}
```

```
public IActionResult Create()
{
    ViewBag.Bosses = _service.GetDropDownEmployees();
    ViewBag.Departments = _service.GetDropDownDepartments();

    return View();
}

// POST: Employees/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(Employee employee)
{
    if (ModelState.IsValid)
    {
        await _service.AddAsync(employee);
        return RedirectToAction(nameof(Index));
    }
    return View(employee);
}

// GET: Employees/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Employee employee = await _service.GetEmployeeAsync(id);
    if (employee == null)
    {
        return NotFound();
    }

    ViewBag.Bosses = _service.GetDropDownEmployees();
    ViewBag.Departments = _service.GetDropDownDepartments();

    return View(employee);
}

// POST: Employees/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, Employee employee)
{
    if (id != employee.EmployeeId)
    {
        return NotFound();
    }
}
```

```
    }

    if (ModelState.IsValid)
    {
        await _service.UpdateAsync(employee);
        return View(employee);
    }
    else
    {
        return RedirectToAction(nameof(Index));
    }
}

// GET: Employees/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Employee employee = await _service.GetAllEmployeeAsync(id);
    if (employee == null)
    {
        return NotFound();
    }

    return View(employee);
}

// POST: Employees/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    await _service.DeleteConfirmedAsync(id);
    return RedirectToAction(nameof(Index));
}
}
}
```

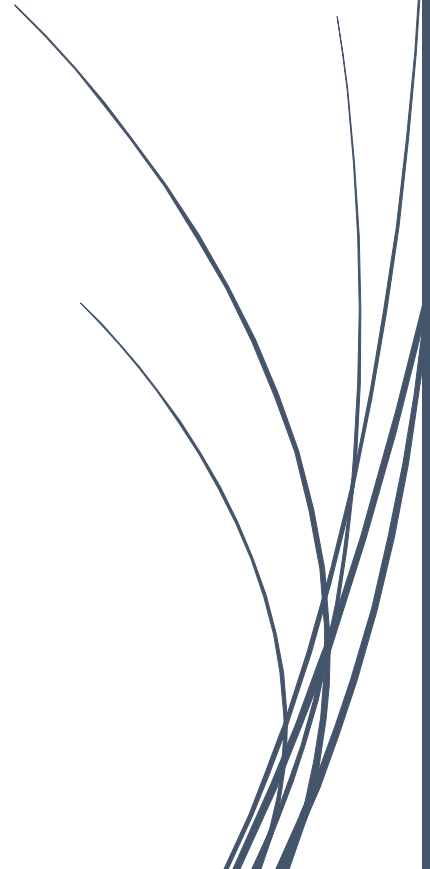
➤ **Authentication و Authorization چیست؟**

➤ **Principal چیست؟**

➤ **مدیریت User**

➤ **Login-Logout-Register**

➤ **Claim چیست؟**



ASP.NET Identity چیست؟

این روزها امنیت برنامه‌های تحت وب، یکی از داغ‌ترین موضوعات دنیای وب است. هر هفته خبرهایی از هک شدن و حمله‌های سایبری به سایت‌های مختلف به گوش می‌رسد. شاید شنیدن این جملات کمی ناامید کننده به نظر برسد، اما نگران نباشید، با دانستن برخی موضوعات می‌توانیم از بسیاری از این حملات جلوگیری کنیم. در این فصل می‌خواهیم به نحوه‌ی حفاظت از اپلیکیشن نگاهی بیندازیم.

یکی از مهمترین ویژگی‌های **ASP.NET Core**، قابلیت ایجاد برنامه‌های دینامیک است. این قابلیت، تنها امکان دیدن بخش‌هایی را به کاربر می‌دهد که اجازه دسترسی داشته باشد. با این حساب اپلیکیشن ما می‌تواند برای کاربران مختلف سفارشی شود.

بسیاری از اپلیکیشن‌ها، مفهومی به نام حساب کاربری دارند، که با آن می‌توانید وارد نرم‌افزار شوید و تجربه کاربری، متفاوت داشته باشید. با داشتن قابلیت حساب کاربری در اپلیکیشن، می‌توانید بسته به شخص لاگین شده امکانات متفاوتی ارائه دهید.

ASP.NET Core Identity یک **Membership system** است که قابلیت ورود به **ASP.NET Core** را اضافه می‌کند. **Identity** می‌تواند با استفاده از دیتابیس **SQL Server** پی‌کرندگی شود تا بدین وسیله نام کاربر، کلمه عبور و اطلاعات پروفایل را ذخیره نماید.

بیا یاد کمی بیشتر به این موضوع بپردازیم و این قابلیت فوق‌العاده را به این اپلیکیشن اضافه نماییم.

Authentication و Authorization چیست؟

زمانیکه می‌خواهید یک کاربر در اپلیکیشن ثبت نمایید، باید به دو جنبه مهم توجه کنید:

Authentication : فرآیند ایجاد کاربری که اجازه ورود به برنامه را دارد.

Authorization : سفارشی سازی میزان دسترسی کاربران و کنترل میزان دسترسی آن‌ها به اپلیکیشن را مشخص می‌نماید.

به عبارت دیگر **Authentication** مشخص می‌کند چه کسی وارد سیستم شده و **Authorization** می‌گوید، به چه چیزهایی باید دسترسی داشته باشد.

ساده‌ترین حالت برای **Authorization** این است که کاربر حداقل باید **Authenticate** شده باشد. این کار توسط اضافه کردن اتریبوت **[Authorize]** به بالای **Action** ها یا **Controller** ها صورت می‌پذیرد.

نکته!!

اتریبوت **[Authorize]** را می‌توان در هر جای برنامه به کار برد، اما استفاده از آن در بالای **Controller** ها و **Action** ها بدین دلیل است که، کنترل کنیم چه کاربری به چه اکشنی دسترسی دارد.

بیا با هم **Identity** را به این اپلیکیشن اضافه نماییم:

(۱) با دستور زیر پکیج **Identity**، را به اپلیکیشن اضافه کنید.

Install-Package Microsoft.AspNetCore.Identity.Core

```

Package Manager Console
Package source: All | Default project: MicrodevProject
PM> Install-Package Microsoft.AspNetCore.Identity.Core
Restoring packages for D:\MicrodevProject\MicrodevProject\MicrodevProject.csproj...
GET https://api.nuget.org/v3-flatcontainer/microsoft.aspnet.identity.core/index.json
OK https://api.nuget.org/v3-flatcontainer/microsoft.aspnet.identity.core/index.json 609ms
GET https://api.nuget.org/v3-flatcontainer/microsoft.aspnet.identity.core/2.2.2/microsoft.aspnet.identity.core.2.2.2.nupkg
OK https://api.nuget.org/v3-flatcontainer/microsoft.aspnet.identity.core/2.2.2/microsoft.aspnet.identity.core.2.2.2.nupkg 247ms
Installing Microsoft.AspNetCore.Identity.Core 2.2.2.
Installing NuGet package Microsoft.AspNetCore.Identity.Core 2.2.2.
Committing restore...
Writing lock file to disk. Path: D:\MicrodevProject\MicrodevProject\obj\project.assets.json
Restore completed in 28.1 sec for D:\MicrodevProject\MicrodevProject\MicrodevProject.csproj.
Successfully installed 'Microsoft.AspNetCore.Identity.Core 2.2.2' to MicrodevProject
Executing nuget actions took 5.96 sec
113 %
    
```

(۲) **EF Core data model** را با **Entity** های **Identity** آپدیت نمایید.

ابتدا در پوشه **Model** یک کلاس به نام **User** ایجاد و در این کلاس از **IdentityUser** ارث‌بری نمایید.

```

using Microsoft.AspNetCore.Identity;
namespace MicrodevProject.Models
{
    public class User: IdentityUser
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
    
```

← برای سفارشی سازی User می‌توانید Property های را در اینجا اضافه نمایید.

(۳) **Identity** از **EF Core** برای ذخیره **User Account** استفاده می‌کند، به همین جهت بعد از اضافه نمودن

User به پروژه، باید **DbContext** را هم کمی تغییر دهیم.

```

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



```
using Microsoft.EntityFrameworkCore;
```

```
namespace MicrodevProject.Data
```

```
{
    public class MicrodevDbContext : IdentityDbContext<User>
    {
        public MicrodevDbContext(DbContextOptions<MicrodevDbContext> options)
        : base(options) { }
        public DbSet<Employee> Employees { get; set; }
        public DbSet<Department> Departments { get; set; }
    }
}
```

جهت پشتیبانی از EF Core Identity باید به جای
ارث‌بری از DbCobtext؛ IdentityDbContext
ارث‌بری نمایید.

برای اینکه EF Core بتواند از Identity پشتیبانی کند باید به جای DbContext از کلاس IdentityDbContext<TUser> ارث‌بری کند. TUser کلاسی است که باید از IdentityUser ارث‌بری کرده باشد.

کلاس IdentityDbContext شامل DbSet<T> های ضروری است که User Entity شما را با استفاده از EF Core ذخیره می‌کند.

با توجه به این موضوع، DbContext باید از IdentityContext<User> ارث‌بری کند.

حالا همانطور که می‌دانید برای اعمال تغییرات باید Migration بزیند.

کد زیر را جهت ایجاد Migration در Package Manager Console وارد نمایید.

Add-Migration IdentityMigration

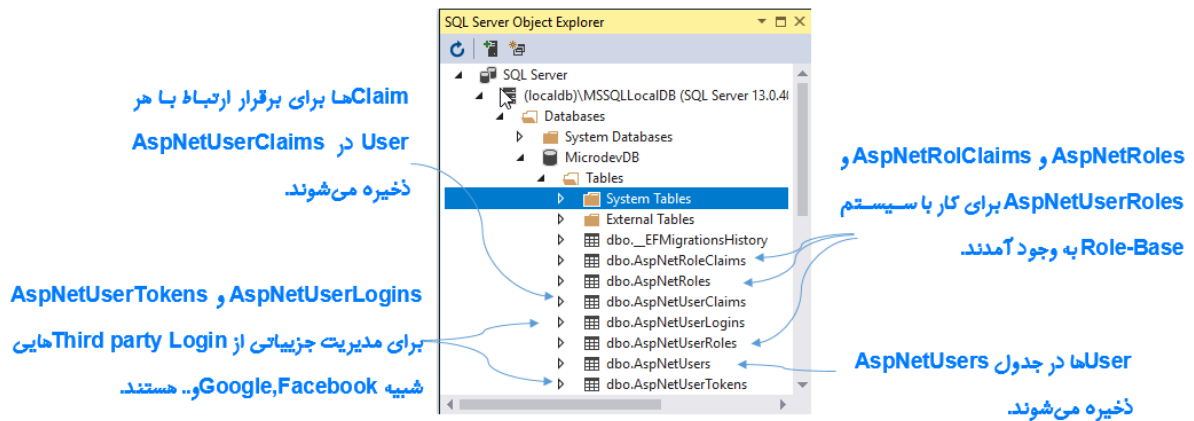
```
Package Manager Console
Package source: All | Default project: MicrodevProject
PM> Add-Migration IdentityMigration
The EF Core tools version '2.1.1-rtm-30846' is older than that of the runtime '2.1.8-servicing-32885'. Update the tools for the latest features and bug fixes.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No type was specified for the decimal column 'Salary' on entity type 'Employee'. This will cause values to be silently truncated if they do not fit in the default
accommodate all the values using 'ForHasColumnType()'.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 2.1.8-servicing-32885 initialized 'MicrodevDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
To undo this action, use Remove-Migration.
PM>
```

حالا دیتابیس را آپدیت کنید.

Update-Database

```
Package Manager Console
Package source: All | Default project: MicrodevProject
PM> Update-Database
The EF Core tools version '2.1.1-rtm-30846' is older than that of the runtime '2.1.8-servicing-32885'. Update the tools for the latest features and bug fixes.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No type was specified for the decimal column 'Salary' on entity type 'Employee'. This will cause values to be silently truncated if they do not fit in the
accommodate all the values using 'ForHasColumnType()'.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 2.1.8-servicing-32885 initialized 'MicrodevDbContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (132ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[_EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[_EFMigrationsHistory]');
```

بیا یاد بگیریم با هم نمای دیتابیس و جداول اضافه شده را ببینیم:



مدیریت User

قصد دارم جهت مدیریت کاربران یک **user** را با استفاده از کلاس **Seeder** در دیتابیس ذخیره کنم.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using MicrodevProject.Models;
using Microsoft.AspNetCore.Identity;
```

```
namespace MicrodevProject.Data
```

```
{
```

```
    public class Seeder
```

```
    {
```

```
        private readonly MicrodevDbContext _context;
```

```
        private readonly UserManager<User> _userManager;
```

```
    public Seeder(MicrodevDbContext context, UserManager<User> userManager)
```

```
    {
```

```
        _context = context;
```

```
        _userManager = userManager;
```

```
    }
```

```
    public async Task Seed()
```

```
    {
```

```
        _context.Database.EnsureCreated();
```

```
        var user = await
```

```
        _userManager.FindByEmailAsync("BytZahra@Gmail.com");
```

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

```
        {
```

```
            user = new User()
```

```
            {
```

یافتن User ای که ایمیل آن با

BytZahra@Gmail.com برابر باشد.

اگر User یافت نشد یک نمونه از

کلاس User ایجاد می کنیم.

```

        FirstName = "Zahra",
        LastName = "Bayat",
        UserName = "BytZahra@Gmail.com",
        Email = "BytZahra@Gmail.com"
    };

    var result = await _userManager.CreateAsync(user,
        "P@ssw0rd!");
    if (result != IdentityResult.Success)
    {
        throw new InvalidOperationException("Failed to create
        default user");
    }
}

if (!_context.Departments.Any())
{
    List<Department> departments = new List<Department>
    {
        new Department{Name="روشنند مدیریت"},
        new Department{Name="باز دکان"},
        new Department{Name="پرداز ایده"},
    };
    _context.Departments.AddRange(departments);
}

if (!_context.Employees.Any())
{
    List<Employee> employees = new List<Employee>
    {
        new Employee {Name="بیات زهرا", Salary=1000 ,BossId=3, DepartmentId=1
        },
        new Employee {Name="بیات علی", Salary=3000 ,BossId=5, DepartmentId=2
        },
        new Employee {Name="مزگانی امین", Salary=1000, BossId=4, DepartmentId=1
        },
        new Employee {Name="سهیلی محمد", Salary = 4000,BossId = 1,
        DepartmentId = 1 },
        new Employee {Name="احمدی سارا", Salary=1000, BossId=4, DepartmentId=2
        },
        new Employee {Name="سعیدی محمد", Salary=1000, BossId=4, DepartmentId=2
        },
        new Employee {Name="افتخاری سهیلا", Salary=2000, BossId=4,
        DepartmentId=2 },
        new Employee {Name="حسینی سعید", Salary=5000, BossId=4,
        DepartmentId=3},
        new Employee {Name="محمدی زهرا", Salary=1000, BossId=8, DepartmentId=3
        }
    };
    _context.Employees.AddRange(employees);
}

```

سپس این User را توسط کلاس UserManager ایجاد می نمایم.

اگر User با موفقیت ایجاد نشود یک اکسپشن پرتاب می شود.

```

        _context.Employees.ToList();
    }
    await _context.SaveChangesAsync();
}
}
}

```

حالا نوبت به ریجستر **Identity** در پروژه می‌رسد:

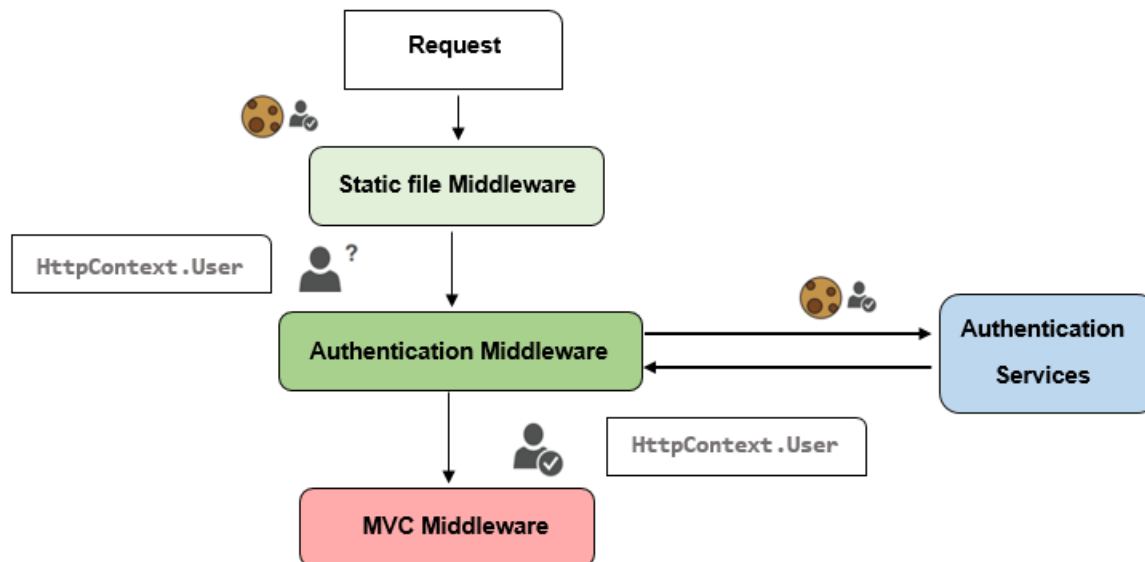
۱) پیکربندی **AuthenticationMiddleware** بسیار ساده است کافیسیت **Middleware** ← **UseAuthentication** را (قبل از **UseMvc**) در متد **Configure** اضافه نمایید.

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    using (var scope = app.ApplicationServices.CreateScope())
    {
        scope.ServiceProvider.GetService<Seeder>().Seed().Wait();
    }
    app.UseAuthentication(); ← پیکربندی AuthenticationMiddleware
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Employee}/{action=Index}/{id?}");
    });
}

```

افزودن **UseAuthentication** امکان **Authenticate** درخواست‌های ورودی را مهیا می‌کند. همانطور که می‌بینید این **Middleware** قبل از **MvcMiddleware** قرار گرفته و قبل از اجرای **MVC** ، **HttpContext.User Principal** را تنظیم و اجرا می‌نماید.



در ادامه مطالبی در مورد HttpContext.User Principal بیان شده است.

(۲) افزودن سرویس Identity و Authentication :

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MicrodevDbContext>(c =>
        c.UseSqlServer(configuration.GetConnectionString("MicrodevConnection")));
```

```
services.AddTransient<Seeder>();
services.AddTransient<IEmployeeService, EmployeeService>();
```

افزودن سرویس
DI به Identity

```
services.AddIdentity<User, IdentityRole>(cfg =>
```

Identity System و پیگر بندی
User و انواع Role

داشتن ایمیل Unique برای
User الزامیست.

```
{
    cfg.User.RequireUniqueEmail = true;
```

پیگر بندی Identity برای ذخیره
داده‌هایش در EF Core

```
.AddEntityFrameworkStores< MicrodevDbContext >());
```

افزودن سرویس
DI در Authentication

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
})
```

```
.AddCookie();
```

سرویس AddCookie برای تنظیم
Principal جاری در کوکی می‌باشد.

```
services.AddMvc();
}
```

نکاتی در مورد AddAuthentication !

- Authentication در Asp.Net Core توسط AddAuthentication انجام می‌پذیرد و همانطور که در کد بالا می‌بینید AddAuthentication تعدادی سرویس جهت تنظیم Principal جاری برای ما فراهم کرده است.
- AddAuthentication یک Option به نام DefaultScheme دارد، که نحوه Authenticate کردن کاربران را تعیین می‌کند.
- این Authenticate می‌تواند در کوکی، توکن و... باشد که در اینجا ما از کوکی استفاده کرده‌ایم.
- ASP.NET Core قابلیت استفاده از توکن bearer برای Authenticate کردن APIها را هم دارد.
- CookieAuthenticationDefaults در فضای نام زیر قرار دارد

Microsoft.AspNetCore.Authentication.Cookies;

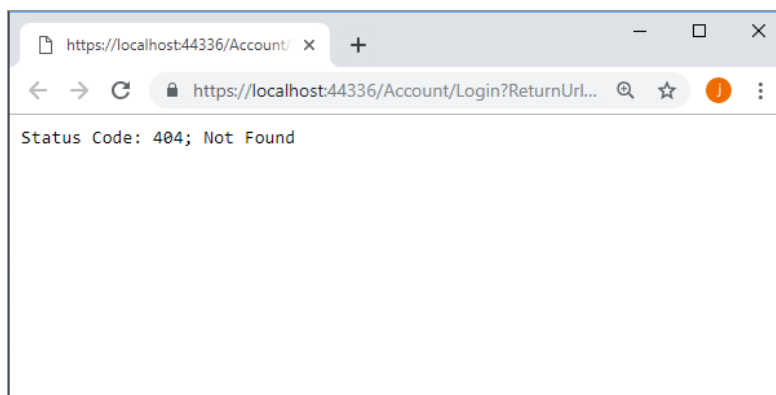
۳) استفاده از Authorize Attribute

در این مرحله بالای Action Method یک [Authorize] قرار دهید.

[Authorize]

```
// GET: Employees
public async Task<IActionResult> Index()
{
    IEnumerable<Employee> employees = await
        _service.GetAllEmployeeAsync();
    return View(employees);
}
```

حالا نوبت به اجرای برنامه می‌رسد. کلید F5 را بزنید تا با هم نتیجه را ببینیم.



همانطور که می‌بینید خطای 404 نمایش داده شد. این خطا بدین دلیل است که ما Controller ی به نام Account و View ی به نام Login در پروژه نداریم.

نکته!!

نمایش Status Code در این صفحه به این خاطر است که ما قبلا در متد Configure، Status Code ها را Handle کرده بودیم.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseExceptionHandler("/Employee/Error");
        app.UseStatusCodePages();
    }
    using (var scope = app.ApplicationServices.CreateScope())
    {
        scope.ServiceProvider.GetService<Seeder>().Seed().Wait();
    }
    app.UseAuthentication();
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Employee}/{action=Index}/{id?}");
    });
}
```

ایجاد صفحه Login

برای رفع این خطا کنترلی به نام AccountController با اکشن متد Login ایجاد می‌کنیم.

```
using Microsoft.AspNetCore.Mvc;

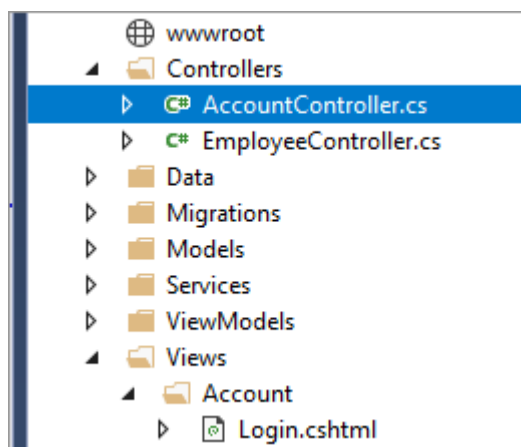
namespace MicrodevProject.Controllers
{
    public class AccountController: Controller
    {
        public IActionResult Login()
        {
            return View();
        }
    }
}
```

AccountController مسئول ریجستر شدن کاربران و لاگین شدن آنهاست.

نکته!!

اگر کاربر لاگین نشده، بخواهد اکشنی را اجرا کند به صفحه لاگین منتقل خواهد شد، اما اگر لاگین شده باشد و دسترسی به Action مورد نظر نداشته باشد به صفحه Access Denied انتقال داده خواهد شد.

حالا در **Folder→View** یک پوشه ای با نام **Account** و **View**ی با نام **Login** ایجاد می‌کنیم.



قبل از نوشتن کدهای صفحه‌ی **Login**، بیاید تغییراتی را به **Layout** پروژه دهیم.

جهت اضافه کردن قابلیت لاگین، به کدهای **navbar** درون فایل **Layout.cshtml** تغییرات زیر را اضافه نمایید.

```
<header>
<nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
  <div class="container">
    <a class="navbar-brand" asp-area="" asp-controller="Home"
      asp-action="Index">MicrodevProject</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-collapse"
      aria-controls="navbarSupportedContent"
      aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
      <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
          <a class="nav-link text-dark" asp-area="" asp-controller="Employee"
            asp-action="Index">کارمند</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```



```

        </li>
        <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
        controller="Department" asp-action="Privacy">دپارتمان</a>
        </li>
    </ul>
</div>
@if (User.Identity.IsAuthenticated)
{
    <div>سلام, @User.Identity.Name</div>
    <a class="nav-link text-dark" asp-area="" asp-
    controller="Account" asp-action="Logout">سایت از خروج</a>
}
else
{
    <a class="btn btn-success" asp-area="" asp-
    controller="Account" asp-action="Login">سایت به ورود</a>
    <a class="btn btn-info" asp-area="" asp-controller="Account"
    asp-action="Register">حساب ایجاد</a>
}
</div>
</nav>
</header>

```

در کدهای بالا می‌گوییم: در صورتی که کاربر **Authenticate** شده باشد **Button** خروج از سایت در **navbar** نمایش داده شود، وگرنه **Button**های ورود به سایت و ایجاد حساب را خواهید دید.

حالا نیاز به یک **ViewModel** برای نگهداری اطلاعات **Login** داریم. در **Folder** **ViewModels** یک کلاس به نام **LoginViewModel** ایجاد نمایید.

```

using System.ComponentModel.DataAnnotations;

namespace MicrodevProject.ViewModels
{
    public class LoginViewModel
    {
        [Required]
        [Display(Name = "کاربری نام")]

        public string Username { get; set; }
        [Required,DataType(DataType.Password)]
        [Display(Name = "عبور کلمه")]

        public string Password { get; set; }
        [Display(Name = "بسیار خاطر مرابه")]
        public bool RememberMe { get; set; }
        public string returnUrl { get; set; }
    }
}

```

```
}
}
```

قبل از اینکه کارتان را شروع کنید، **Namespace** های موردنیازتان را در فایل `ViewImports.cshtml` درج نمایید.

وارد **ViewImport** شوید و فضای نام **ViewModels** را اضافه کنید:

```
@using MicrodevProject
@using MicrodevProject.Models
@using MicrodevProject.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

حالا وارد **View** ← **Login** شوید و کدهای زیر را به آن بیفزایید.

```
@model LoginViewModel
@{
    ViewData["Title"] = "ورود صفحه";
}
```

```
<h1 class="d-flex justify-content-center">ورود صفحه</h1>
<hr />
<div class="row" dir="rtl">
    <div class="col-md-4">
        <form method="post" asp-antiforgery="true">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group" dir="rtl">
                <label asp-for="Username" class="control-label" dir="rtl"
                    style="float: right"></label>
                <input asp-for="Username" class="form-control" />
                <span asp-validation-for="Username" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label" style="float:
                    right"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <div class="checkbox">
                    <label asp-for="RememberMe" style="float: right">
                        <input asp-for="RememberMe" />
                        @Html.DisplayNameFor(m => m.RememberMe)
                    </label>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="ورود" class="btn btn-success" />
            </div>
        </form>
    </div>
</div>
```

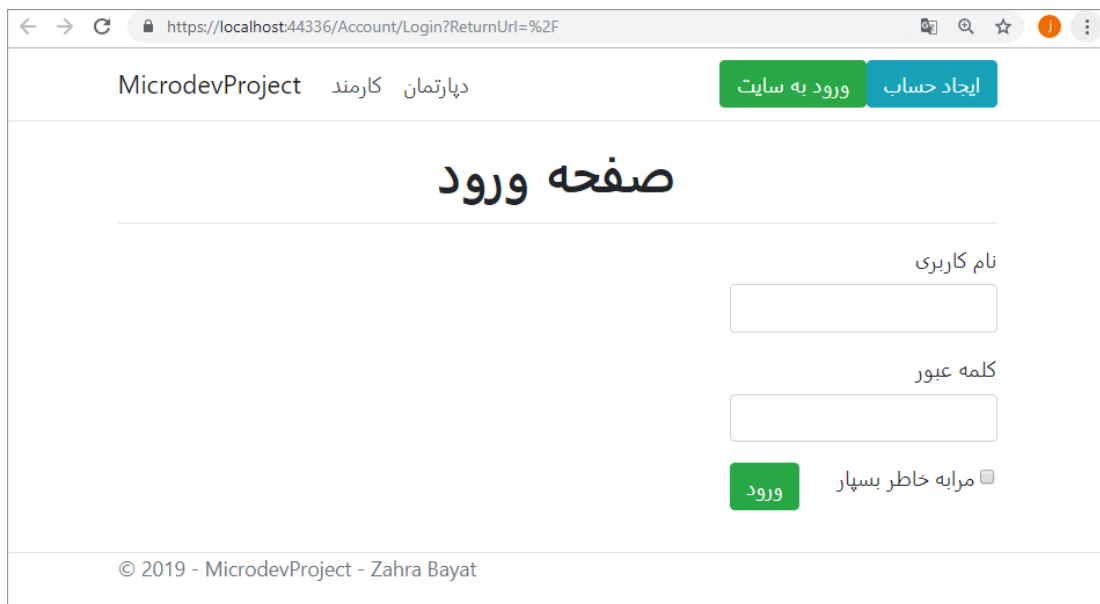
نحوه نمایش خطاها

```

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

```

حالا با اجرای برنامه وارد صفحه لاگین می شوید.



فرم لاگین ، درخواست کاربر برای ورود به سیستم را می گیرد و سپس اجازه دسترسی به کاربر را می دهد.

اگر می خواهید با کلیک بر روی دکمه ورود، لاگین شوید باید متد **Post** را برای اکشن **Login** بنویسید.

```

private readonly SignInManager<User> _signInManager;
private readonly UserManager<User> _userManager;

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

[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await
            _signInManager.PasswordSignInAsync(model.Username,
            model.Password,
            model.RememberMe,

```

برای ایجاد یک کاربر جدید می توانید از کلاس **UserManager<T>** استفاده کنید.

کلاس **SignInManager<T>** برای **login** و **logout** و **پر کردن Cookie** ها و همچنین **لود کردن** کاربر و **چک کردن پسورد** مورد استفاده قرار می گیرد.

با **Sign in** Username, password

```

        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();
}

```

اگر Sign in موفق باشد، HttpContext.user و Authentication Cookie تنظیم می‌شود.

متد IsLocalUrl جهت جلوگیری از آسیب‌پذیری open redirect استفاده می‌شود.

خوشحالم از اینکه توانستید قابلیت **Identity** را به پروژه اضافه و صفحه لاگین پروژه را راه‌اندازی کنید. همراه ما باشید، می‌خواهم کمی در مورد قابلیت **Claim Based** در **ASP.Net Core** صحبت کنم.

Claim چیست؟

Claim یک جفت **Type-Value** (هر دو از نوع **string** هستند) است که مشخص کننده **Principal** می‌باشد.

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.

Propertyهای اساسی مربوط به **Identity** در **ASP.NET Core** هم همانند **Asp.Net** است و تغییر چندانی نکرده. به طور مثال:

در **ASP.NET 4.x** یک **Property** به نام **User** از نوع **IPrincipal** در **HttpContext** وجود داشت که نشان دهنده کاربر فعلی برای یک **Request** بود. در **ASP.NET Core** هم یک **Property** به نام **User** وجود دارد، با این تفاوت که **User** از نوع **ClaimsPrincipal** است، که **IPrincipal** را پیاده‌سازی کرده است.

استفاده از **ClaimsPrincipal** یک تغییر اساسی در **Identity** است که در **ASP.NET Core** آورده شده است.

نکته!!

برای دسترسی به **Principal** درخواست جاری، می توانید از **HttpContext.User** استفاده کنید.

دوست دارم مفهوم **Claim** را با یک مثال بیان کنم.

مثالی در مورد **Claim**:

در گواهینامه رانندگی شما کد ملی درج شده است. در اینجا می توان **Claim**ی به نام **NationalCode** داشت که مشخص کننده کد ملی شما باشد.

پس گواهینامه مجوزی مبتنی بر ادعاهای شماست. براساس این **Claim**ها، ارزش یک ادعا بررسی می شود و براساس این ارزش به شما اجازه داده می شود که دسترسی هایی داشته باشید.

افسر، صحت کارت شما را بررسی می کند و براساس اطلاعات درون کارت، اجازه رانندگی به شما می دهد.

یک **Identity** می تواند شامل **Claim**هایی با چندین مقدار باشد و حتی می تواند چندین **Claim** مشابه داشته باشد.

در ورژن های قبلی **AspNet** به جای **Claim** از **Role** استفاده می شد. هنوز هم می توانید از **Role**ها استفاده کنید، اما بهتر است تا جایی که امکان دارد از **Claim** استفاده نمایید.

Identity در **AspNet Core** یک کلاس **ClaimIdentity** دارد که اینترفیس **IIdentity** را پیاده سازی کرده است. در مثال زیر سعی کردیم این کلاس را شبیه سازی کنیم. احتمالاً این کلاس کمی بزرگتر از مثال ما باشد.

```
public class ClaimsIdentity : IIdentity
{
    public string AuthenticationType { get; }
    public bool IsAuthenticated { get; }
    public IEnumerable<Claim> Claims { get; }

    public Claim FindFirst(string type) { /*...*/ }
    public Claim HasClaim(string type, string value) { /*...*/ }
}
```

در مثال بالا **Property**های اصلی بیان شده است.

➤ **Claims**: این **Property** شامل تمام **Claim**های **Identity** است.

- **AuthenticationType** : این **Property** برای مشخص کردن نوع **Authentication** است. مثلا گواهینامه رانندگی که بالاتر توضیح دادیم **AuthenticationType** است و در **ASP.NET** از طریق **Cookie, Bearer, Google** و... می توان صحت این **Property** را مشخص نمود.
- **IsAuthenticated** : این **Property** مشخص می کند که کاربر **Authenticated** شده یا خیر؟
- تعدادی متدهای کاربردی هم برای کار با **Claim** ها وجود دارد که در اینجا دو مورد آورده شده است. این متدها زمانی مفید است که کاربر **Authorize** شده باشد.

ایجاد یک **Principal**

تا اینجا با نحوه کارکرد **Principal** در **ASP.NET Core** آشنا شدید، حالا چطور می توان یک **Principal** ایجاد نمود؟

به مثال زیر توجه کنید:

```
public async Task<IActionResult> Login(string returnUrl = null)
{
    const string Issuer = "http://www.Microdev.ir";

    var claims = new List<Claim> {
        new Claim(ClaimTypes.Name, "Zahra", ClaimValueTypes.String, Issuer),
        new Claim(ClaimTypes.Surname, "Bayat", ClaimValueTypes.String,
Issuer),
        new Claim(ClaimTypes.Country, "Iran", ClaimValueTypes.String,
Issuer),
    };

    var userIdentity = new ClaimsIdentity(claims, "Passport");

    var userPrincipal = new ClaimsPrincipal(userIdentity);

    await HttpContext.Authentication.SignInAsync("Cookie", userPrincipal,
new AuthenticationProperties
    {
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20),
        IsPersistent = false,
        AllowRefresh = false
    });

    return RedirectToLocal(returnUrl);
}
```

➤ اولین کاری که انجام دادیم لیستی از **Claim**ها را ایجاد نمودیم و برای هر **Claim** یک **Name** و یک **Value** و به صورت اختیاری فیلدهایی برای **Issuer** و **ClaimValueType** ایجاد نمودیم.

نکته!!

در این متد **Claim**ها به صورت هاردکد نوشته شده است اما شما می توانید این مقادیر را از دیتابیس بخوانید.

➤ هنگامی که **Claim**های خود را ساختید می توانید یک **ClaimsIdentity** ایجاد کنید و لیست **Claim**های خود را به آن پاس دهید.

➤ سپس **AuthenticationType** را مشخص کنید. برای اطمینان از اینکه هویت شما **Authenticated** شده باشد **IsAuthenticated = true** قرار دهید.

➤ در نهایت می توانید یک **ClaimsPrincipal** جدید را با استفاده از **identity** خود ایجاد و کاربر را **Sing in** کنید.

➤ در مثال بالا ما به **AuthenticationManager** می گوئیم که از **Cookie** برای **Authentication handler** استفاده کند.

توجه داشته باشید که ما بالاتر **Middleware Pipeline** را برای این بخش پیکربندی کردیم.

Claim برای کاربر

هر کاربر تعدادی **Claim** دارد، این **Claim**ها اطلاعاتی در مورد کاربر هستند (مثل نام، ایمیل و...) که شما می توانید هر جایی از آن ها استفاده کنید:

```
@if (User.Identity.IsAuthenticated)
{
    foreach (var identity in User.Identities)
    {
        <h3>@identity.Name</h3>
        <ul>
            @foreach (var claim in identity.Claims)
            {
                <li>@claim.Type - @claim.Value</li>
            }
        </ul>
    }
}
else
{
```

```
<div>You are anonymouse</div>
}
```

نکته!!

شما می‌توانید Claim‌های دلخواه خود را از طریق متد زیر اضافه نمایید.

```
userManager.AddClaimAsync(user, Claim)
```

افزودن Claim Check

Claims-Based Authorization از Claim‌های کاربر جاری، جهت تعیین اینکه این کاربر اجازه اجرای چه اکشنی را دارد استفاده می‌کند. برای اینکه مشخص کنید چه Claim‌هایی برای اجرای یک Action نیاز است، می‌توانید از Policy‌ها استفاده کنید.

نیازمندی Claim‌ها مبتنی بر Policy است، که برنامه‌نویس باید Policy موردنیاز Claim را ایجاد و و ریجستر نماید.

افزودن Policy‌ها از طریق متد AddPolicy درون متد AddAuthorization() (در بدنه متد ConfigureServices موجود در فایل Startup.cs قرار دارد) تعریف می‌شود. هر Policy شامل یک نام و یک عبارت Lambda می‌باشد، که برای مشخص کردن Claim‌هایی است، که برای اجرا به آن نیاز دارد.

```
public void ConfigureServices(IServiceCollection services).
```

```
{
    services.AddMvc();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy =>
            policy.RequireClaim("EmployeeNumber"));
    });
}
```

در این حالت EmployeeOnly policy برای بررسی حضور EmployeeNumber Claim در Identity جاری است.

بعد از انجام مراحل بالا، باید Policy را با استفاده از Policy Property روی AuthorizeAttribute قرار دهید.

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult Index()
{
```



```
return View();
}
```

نکته!!

`AuthorizeAttribute` می‌تواند به کل `Controller` اعمال گردد.

```
[Authorize(Policy = "EmployeeOnly")]
public class HomeController : Controller
{
    public ActionResult Index()
    {
    }
}
```

شما اگر کنترلی دارید که دارای `AuthorizeAttribute` می‌باشد، و می‌خواهید اجازه `Anonymous access` در اکشن خاص را هم داشته باشید، می‌توانید از `AllowAnonymousAttribute` استفاده کنید.

```
[Authorize(Policy = "EmployeeOnly")]
public class HomeController : Controller
{
    public ActionResult Index()
    {
    }

    [AllowAnonymous]
    public ActionResult Index ()
    {
    }
}
```

ایجاد Logout

کاربر بعد از ورود به سیستم و استفاده از اپلیکیشن باید بتواند از سیستم خارج شود. پس بیایید برای `Logout` هم متدی بنویسیم.

```
[HttpGet]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync(); ←
    return RedirectToAction("Index", "Home");
}
```

anonymous با یک `HttpContext.user`
`principal` جایگزین می‌شود و
`Authentication cookie` حذف خواهد شد.

همانطور که در کد بالا می بینید بعد از Logout شدن به صفحه Index برمی گردید.

ثبت نام کاربر

صفحه Login/Logout شما با موفقیت ایجاد شد. اما متاسفانه هنوز صفحه‌ای برای ثبت نام کاربر وجود ندارد. در کنترلر Account یک متد Register ایجاد نمایید.

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

این متد جهت نمایش صفحه‌ی ثبت نام کاربر می باشد، اما چون هنوز View ی Register ایجاد نشده، اگر بر روی ایجاد حساب در navbar کلیک کنید، به خطا برخورد خواهید خورد.

پس در مسیر Views/Account یک View به نام Register ایجاد نمایید و کدهای زیر را درون آن قرار دهید.

```
@model RegisterUserViewModel
@{
    ViewData["Title"] = "کاربر ثبت";
}

<h1 class="d-flex justify-content-center">کاربر ثبت </h1>
<hr />
<div class="row" dir="rtl">
    <div class="col-md-4">
        <form method="post" asp-antiforgery="true">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group" dir="rtl">
                <label asp-for="Username" class="control-label" dir="rtl" style="FLOAT: right"></label>
                <input asp-for="Username" class="form-control" />
                <span asp-validation-for="Username" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="control-label" style="FLOAT: right"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
        </div>
    </div>
</div>
```

```

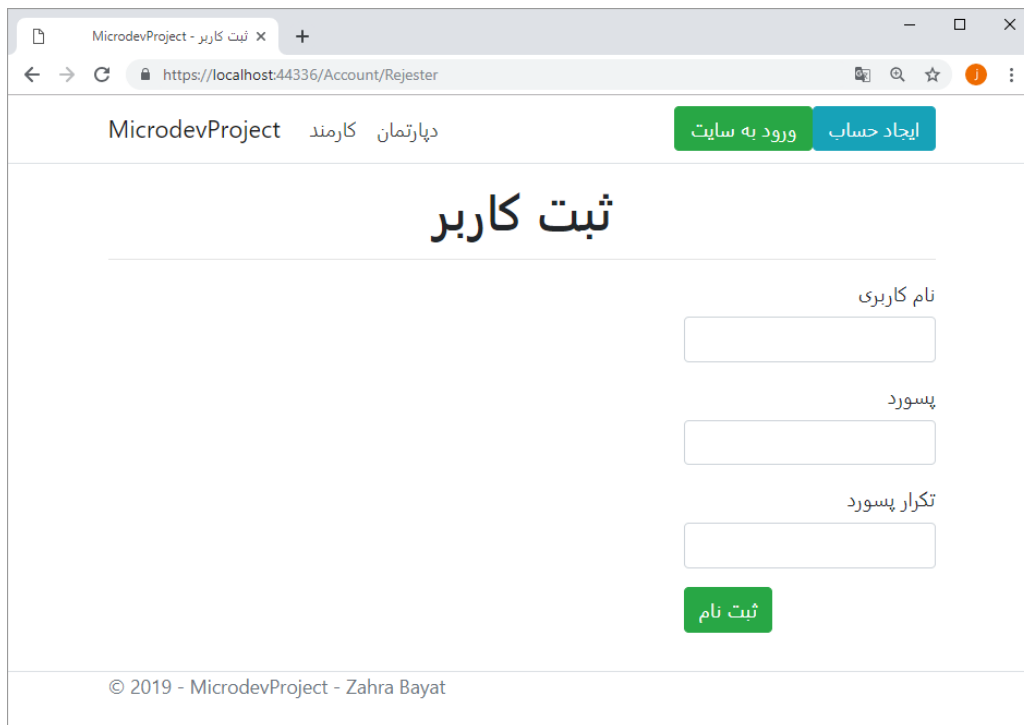
<label asp-for="ConfirmPassword" class="control-label" style="
FLOAT: right"></label>
    <input asp-for="ConfirmPassword" class="form-control" />
    <span asp-validation-for="ConfirmPassword" class="text-
danger"></span>
</div>
<div class="form-group">
    <input type="submit" value="نام ثبت" class="btn btn-success" />
</div>
</form>
</div>
</div>

```

نکته!!

همانطور که مشاهده می‌کنید در View فوق از عبارت using برای Import کردن namespace حاوی ViewModel استفاده نکرده‌ایم؛ زیرا این کار را یکبار درون فایل _ViewImports.cshtml برای تمامی Viewها انجام داده‌ایم:

در navbar بر روی ایجاد حساب کلیک کنید تا نتیجه کدهای بالا را ببینیم.



برای ثبت کاربر نیاز به یک ViewModel جهت نگهداری اطلاعات داریم. پس یک ViewModel به نام RegisterUserViewModel در Folder ViewModels ایجاد کنید.

```
using System.ComponentModel.DataAnnotations;
```

```
namespace MicrodevProject.ViewModels
{
    public class RegisterUserViewModel
    {
        [Required ,MaxLength(256),Display(Name = "نام کاربری")]
        public string Username { get; set; }
        [Required ,DataType(DataType.Password), Display(Name = "پسورد")]
        public string Password { get; set; }
        [DataType(DataType.Password),Compare(nameof(Password)), Display(Name = "تکرار پسورد")]
        public string ConfirmPassword { get; set; }
    }
}
```

حالا نوبت به نوشتن متد **Post** برای **Button** ثبت کاربر می‌رسد.

```
[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterUserViewModel model)
{
    if (ModelState.IsValid)
    {
        User user = new User { UserName = model.Username };
        IdentityResult result = await _userManager.CreateAsync(user,
            model.Password);
        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
            return RedirectToAction("Index", "Home");
        }
    }
    return View();
}
```

فرم تنها محدود به RegisterUserViewModel است.

اگر Binding معتبر باشد یک Instance از User ایجاد می‌شود.

اعتبار پسورد را چک می‌کند و سپس یک User را در دیتابیس ایجاد می‌کند.

اگر ایجاد User موفق باشد، کاربر Sign in می‌شود.

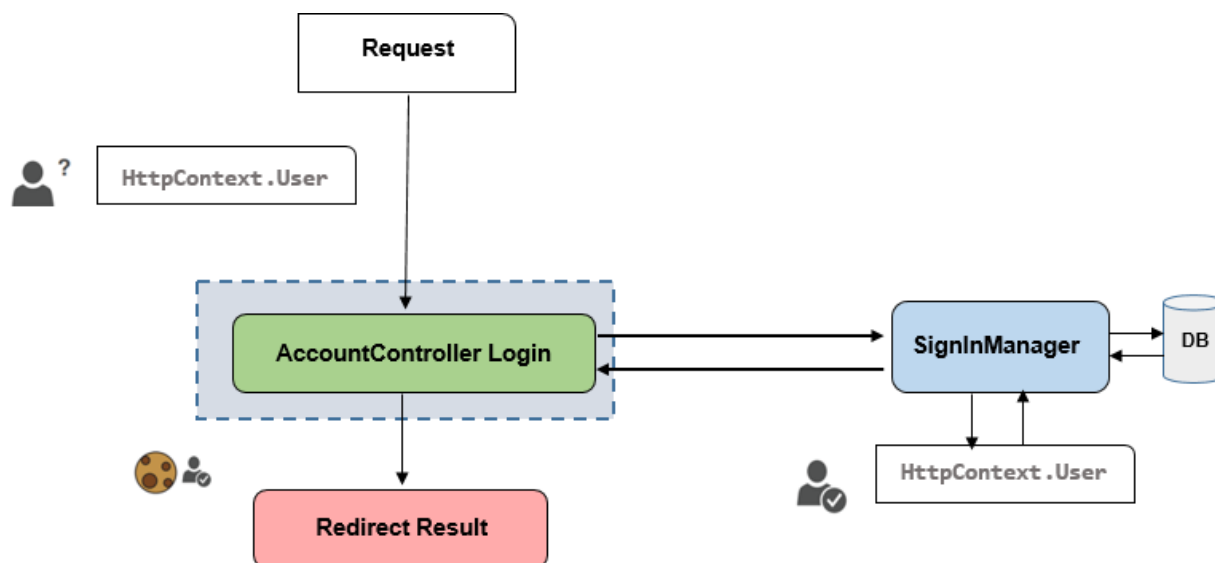
HttpContext.user principal آپدیت و کوکی شامل Serialized principal تنظیم می‌شود.

اگر هر گونه خطایی رخ دهد به خطاهای ModelState اضافه می‌شود.

هنگامی که بر روی ثبت‌نام کلیک کنید این متد فرم ثبت‌نام را ارسال خواهد کرد.

نکاتی در مورد متد Register !

- اگر UserManager، کاربر را با موفقیت ایجاد کند، پس از آن می‌توانید از SignInManager برای Sign in کاربر ایجاد شده استفاده کنید.
- SignInManager مسئول تنظیم خصیصه HttpContext.User برای User Principal و Serializing Principal در استفاده از یک کوکی در درخواست بعدی است.
- وقتی کاربر Sing in می‌شود، شما می‌توانید آن را به هر صفحه‌ای که قبلاً در حال انجام آن بوده Redirect نمایید.
- اگر نتیجه‌ی یک کاربر false باشد؛ یعنی Succeed برابر با false بوده، پس تمامی خطاهای درون پراپرتی Errors را، به ModelState اضافه می‌کنیم.



سخن پایانی

دنیای کامپیوتر را کارهای رایگان زنده نگه داشته است. شما به عنوان یک برنامه‌نویس می‌توانید هر هفته چند ساعت را به حل مشکلات دیگران بپردازید. این حل مشکلات می‌تواند ساخت یک برنامه‌ی متن باز یا پاسخ دادن به یک پرسش در **stackoverflow** باشد.

از اینکه وقت با ارزشتان را صرف خواندن این کتاب کردید سپاسگزارم. اگر با خواندن این کتاب چیزی آموختید، لطفاً آن را با دوستانتان به اشتراک بگذارید. و اگر جایی از کتاب مبهم بود یا من نتوانسته بودم موضوع را خوب بیان کنم، لطفاً در قسمت نظرات کتاب به من اطلاع دهید. تا در ویرایش‌های بعدی اصلاح کنم. این تمام چیزی است که از شما انتظار دارم.

امیدوارم با تلاش‌های تک‌تک ما، دنیا جای بهتری برای زندگی شود.

<https://www.linkedin.com/in/alibayatgh/>

<https://www.linkedin.com/in/zahrabayat/>

<https://stackoverflow.com/users/3427324/ali-bayat>

<https://stackoverflow.com/users/9423189/zahra-bayat-gholilaleh>

کتاب های نوشته شده:

