



ANY
Server

ASP.NET
Core

ANY
Client



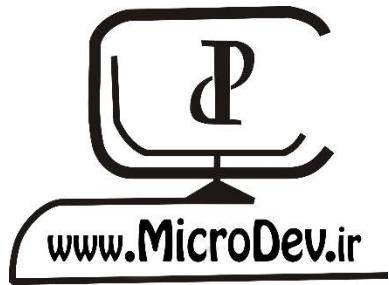
ویدايش دوم

ASP.NET Core

به زبان ساده

مولفین:

زهرا بیات قلی لاله
علی بیات قلی لاله



ASP.NET Core

به زبان ساده

زهرا بیات قلی‌لاله

علی بیات قلی‌لاله

ASP.NET Core

به زبان ساده

مؤلفین : زهرا بیات قلی‌لاله – علی بیات قلی‌لاله

طراح جلد : زهرا بیات قلی‌لاله

ناشر : سایت کتاب راه

مشخصات ظاهری : ۱۳۹ ص

سال انتشار: دی ۹۸

قیمت : رایگان

فهرست

تقدیم به:

با تشکر:

مقدمه

فصل اول: مقدمات **ASP.NET Core**

ASP.NET Core چیست؟

ASP.NET Core مزایای

ایجاد اولین اپلیکیشن **ASP.NET Core**

ساختار پروژه

کلاس **Program**

کلاس **Startup**

ASP.NET Core در **Middleware** چیست؟

قابلیت‌های **Middleware**

موارد استفاده از **Middleware**

نحوه‌ی عملکرد **Middleware**

کاربرد متدهای **ConfigureServices** در کلاس **Startup**

چطور **HandleRequest** را با **Middleware** می‌شوند؟

کاربرد متدهای **Configure** در کلاس **Startup**

wwwroot

فصل دوم : مدیریت **Exception** ها

مدیریت **Exception** ها

مدیریت **Exception** ها و محیط‌های اجرایی

در محیط اجرایی **Exception**

تعریف **Environment**

مدیریت Status code ها

فصل سوم : مقدمات EF Core

MVC چیست؟

Routing چیست؟

مقدمه ای در مورد EF Core

اضافه کردن دیتابیس به پروژه

DataAnnotation چیست؟

DbContext تعریف

ConnectionString چیست؟

مقدمه ای در مورد Dependency Injection

DI مزایای

طول عمر یک سرویس ایجاد شده توسط DI

انواع طول عمر

فصل چهارم : ایجاد دیتابیس

رجیستر DI از طریق DbContext

نکاتی در مورد IConfiguration service و appsettings.json

Entity Framework Migration چیست؟

ایجاد Migration

: EF Core نکاتی در مورد

فصل پنجم : عملیات CRUD

Mutation Of Concerns مفهوم

CRUD شروع عملیات

Controller چیست؟

Layout چیست؟

نمایش لیست کارمندان

چیست؟ Razor

چیست؟ Tag Helper

چیست؟ _ViewStart و _ViewImports

جزئیات کارمند

ایجاد URL برای اکشن

چیست؟ Model Binding

بر روی Server Validation

چیست؟ ValidateAntiForgeryToken

متدهای حذف کارمند

درج کارمند جدید

تمام کدهای EmployeeController

فصل ششم : ASP.NET Identity

چیست؟ ASP.NET Identity

چیست؟ Authorization و Authentication

مدیریت User

ایجاد صفحه Login

چیست؟ Claim

چیست؟ Principal

ایجاد یک Claim

برای کاربر Claim

افزودن Claim Check

ایجاد Logout

ثبت نام کاربر

سخن پایانی

کتاب های نوشته شده:

تقدیم به:

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

با تشکر:

از شرکت مدیریت روشمند و همکار 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 باورنگردنی 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 موردنبیاز 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 بباید 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.

بباید 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

➤ ایجاد اولین Middleware در ASP.NET Core

ASP.NET Core چیست؟

این روزها، وب اپلیکیشن‌ها، در همه جا دیده می‌شوند و انتظار می‌رود که این اپلیکیشن‌ها، بی‌نهایت 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.

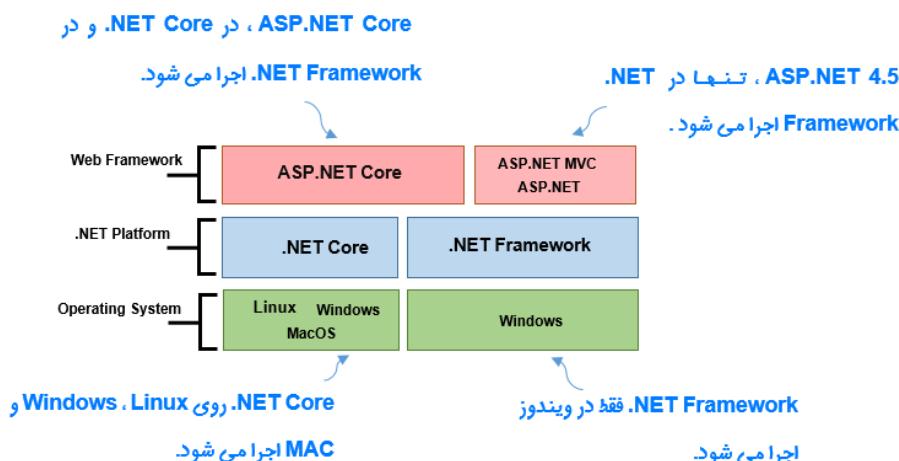
نکته بسیار مهم و جذاب ماجرای .NET Core. این است که، این تکنولوژی می‌تواند هم در ویندوز، هم لینوکس و هم سیستم‌عامل اپل اجرا شود.

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

آخرین دستاوردهای ASP.NET می‌باشد و با هدف Little Coupling و Highly Modular طراحی شده است. با این روند برنامه شما با مازوچیتی 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.

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 **Host** که باشد، می‌تواند به درخواست **HTTP** پاسخ دهد و اپلیکیشن شما در هر سروری که بتواند به **Request**های **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 می‌شوند:

۱. درخواست از یک **Browser**، که **HTML** برمی‌گرداند. این برای **ASP.NET** به عنوان **Web UI** است.
۲. یک درخواست از نرمافزار که داده‌ها را برمی‌گرداند. این برای **ASP.NET Core** به عنوان **Web API** یا **WebService** است.

یک موضوع جالب:

یک بار دیگر:

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

مزایای ASP.NET Core

برای دانستن اینکه، چرا **مايكروسافت** تصمیم گرفت تا یک فریم‌ورک جدید را ایجاد نماید، باید مزایای آن را دنبال نمایید.

امکانات زیادی را با خود به ارمغان آورده است:

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

شاید نسبت به بعضی از این واژگان بیگانه باشد.

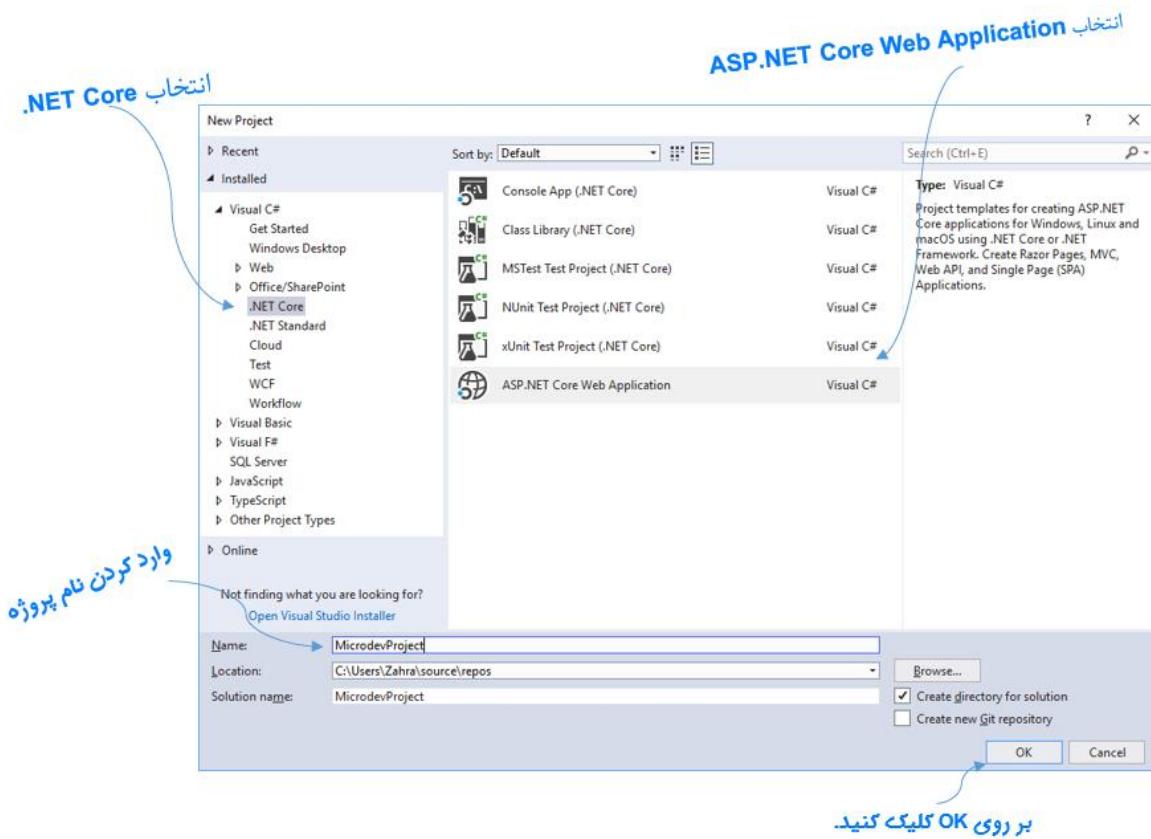
نگران نباشید!! در طول آموزش این مفاهیم توضیح داده شده و مدام تکرار می‌شود.

ایجاد اولین اپلیکیشن ASP.NET Core

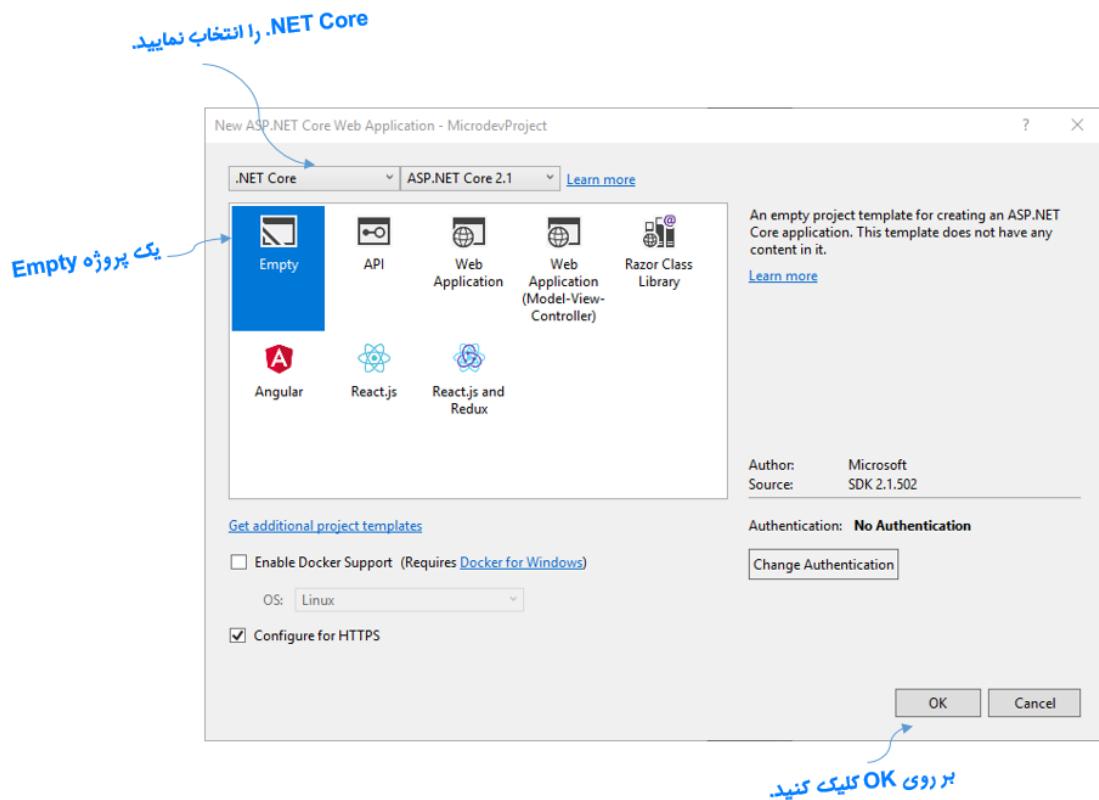
باعث می‌شود برنامه‌های کاربردی وب، سریع‌تر، آسان‌تر، راحت‌تر و امن‌تر ایجاد شوند.

پس بباید 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** را انتخاب نمایید.
- نام پروژه، مکان ذخیره‌سازی و نام **Solution** (اختیاری) را وارد و بر روی **OK** کلیک نمایید.



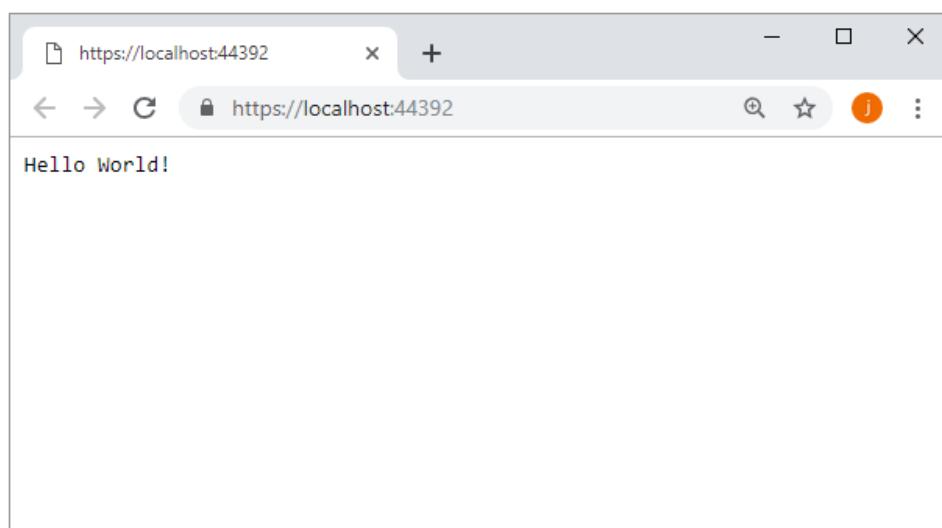
چون می‌خواهم مباحث **ASP.Net** را از پایه شروع کنم، پس در کادر بعد، گزینه‌ی **Empty** را انتخاب نمایید و بر روی **OK** کلیک کنید.



منتظر بمانید تا پروژه‌ی شما ایجاد شود....

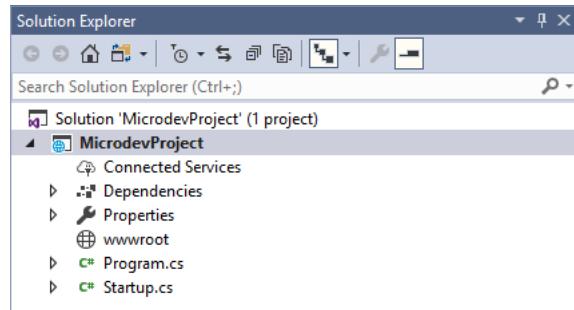
تبریک!! پروژه‌ی شما با موفقیت ایجاد شد.

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



ساختار پروژه

در تصویر زیر ساختار پروژه 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 کلاس **Program.cs** شروع می‌شوند. این کلاس یک متدهای **Main** دارد و زمانی که اپلیکیشن شما **Start** شود، این متدها 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>();
}
```

هدف این متدها ایجاد یک **IWebHost** و صدای زدن متدهای **Run** آن می‌باشد.

متدهای **CreateDefaultBuilder** یک نمونه از **IWebHostBuilder** ایجاد و تنظیمات زیر را انجام می‌دهد.

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

بیشتر تنظیمات اپلیکیشن توسط **WebHostBuilder** انجام می‌شود، اما بعضی از وظایف به کلاس **Startup** واگذار خواهند شد. به همین منظور برای معرفی کلاس **Startup** باید از متدهای **UseStartup** استفاده نمایید و در نهایت با صدا زدن متدهای **IWebHost** \leftarrow **Build** را ایجاد کنید.

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

چرا **Kestrel**؟

Kestrel وب‌서وری است که به عنوان بخشی از **ASP.NET Core** عرضه شده. این وب‌سرور به صورت **Cross Platform** است، یعنی بر روی تمامی سیستم‌عامل‌ها پشتیبانی شده و توسط **.NET Core** قابل اجرا می‌باشد.

کلاس **Startup**

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

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

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

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

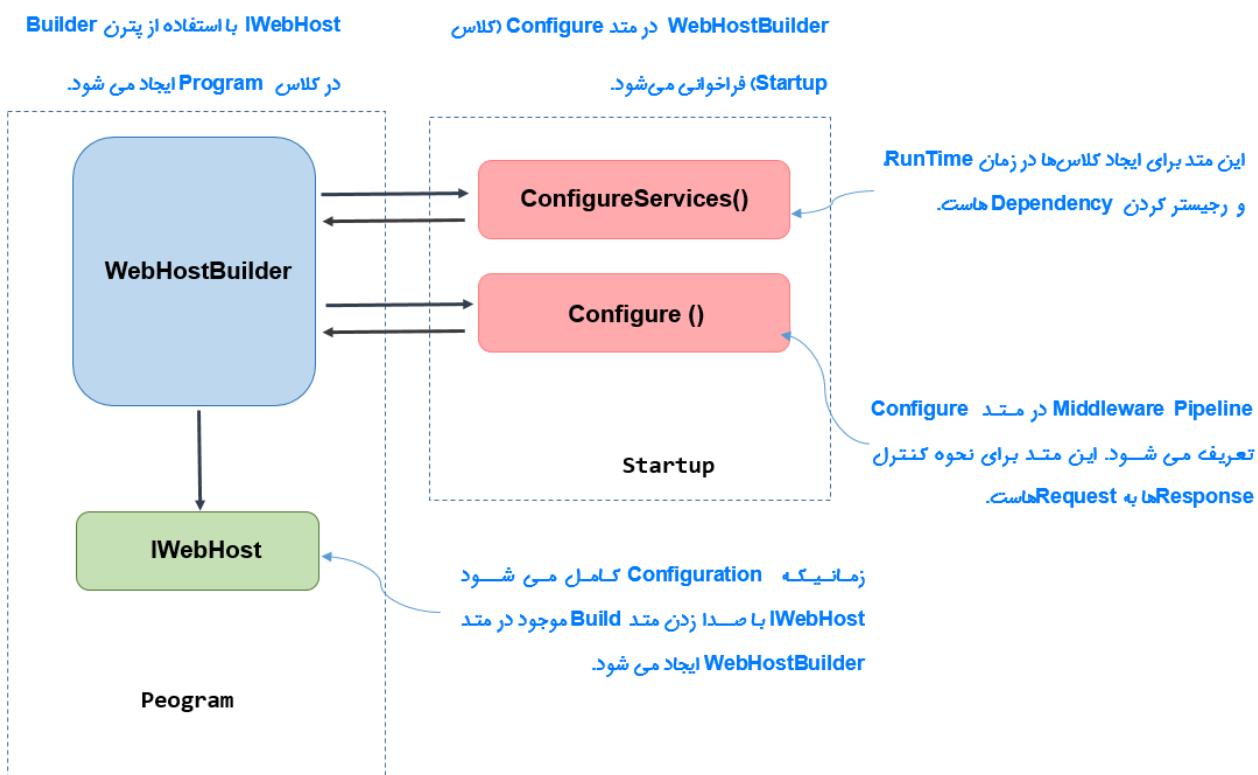
(۲) **Middleware‌ها**: نحوه **Handle** کردن اپلیکیشن، برای **Request**‌ها به **Response**‌ها از طریق **Middleware**‌هایی است، که باید اینجا مشخص می‌شود.

```
public void ConfigureServices(IServiceCollection services)
{
    // رجیستر گردن سرویس‌ها با استفاده از IServiceCollection
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // پیکربندی Middleware Pipeline برای Handle Request‌های HTTP
}
```

هر فراخوانی موجود در کلاس `WebHostBuilder`، متدهای `Configure` و `ConfigureServices` را فراخوانی می‌کند. هر فراخوانی یک بخش متفاوتی از درخواست شما را تنظیم می‌نماید، که آن تنظیم برای فراخوانی بعدی در دسترس است.

هر سرویس ثبت شده در متدهای `Configure` و `ConfigureServices` در دسترس است. هنگامی که پیکربندی کامل شود، یک `IWebHost` توسط `WebHostBuilder` موجود در `Build()` ایجاد می‌شود.



ASP.NET Core در Middleware چیست؟

Middleware در ASP.NET Core ابزاری است که `Request` و `Response` ها را `Handle` می‌کند. مهم‌ترین `Middleware` در اپلیکیشن‌های `ASP.NET Core`، `MvcMiddleware`، `API` و `HTML page` است که می‌تواند `Request` و `Response` را تولید کند.

Middleware قابلیت‌های

مهم‌ترین قابلیت‌های `Middleware` را می‌توان به سه دسته تقسیم نمود:

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

با وجود این سه قابلیت نتیجه می‌گیریم: با استفاده از **Middleware**، می‌توان نحوه‌ی پاسخ‌گویی درخواست‌های **HTTP** را کنترل نمود.

موارد استفاده از **Middleware**

ها برای **Handle** کردن دو چیز بسیار مناسب هستند:

- **Error handling**
- **Logging**

اما مهم‌ترین موارد استفاده **Middleware**، رفع نگرانی‌های مدیریتی است.

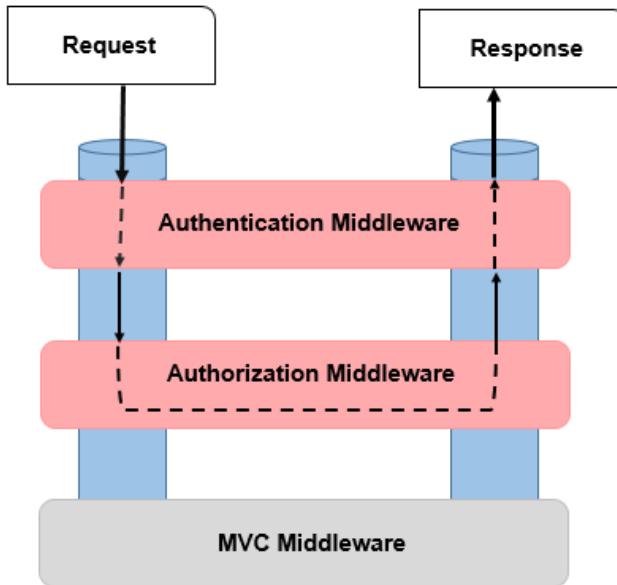
به‌طور مثال:

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

نحوه‌ی عملکرد **Middleware**

نحوه عملکرد بدین صورت است که: یک درخواست را دریافت کرده، گاهی آن را تغییر داده و درنهایت 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**، دو طرفه است. تا زمانیکه **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 متدهای `ConfigureServices` در کلاس `Startup`

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

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

اکستنشن متدهای `AddMvc` بر روی **IServiceCollection** است که سرویس **MVC** را به اپلیکیشن شما اضافه می‌کند.

طمئیناً با دیدن این کد شگفت زده شدید!!

تنهایا با نوشتن یک متدهای `AddMVC()` یک برنامه کامل **MVC** در اپلیکیشن تان راه اندازی شد. با نوشتن این خط کد، پشت‌صفحه تمام سناریوهای مختلف، از جمله سرویس‌های **Razor**، **HTML**، **Render** کردن، **Routing** و... انجام می‌شود.

چطور **Handle Request** ها با **Middleware** می‌شوند؟

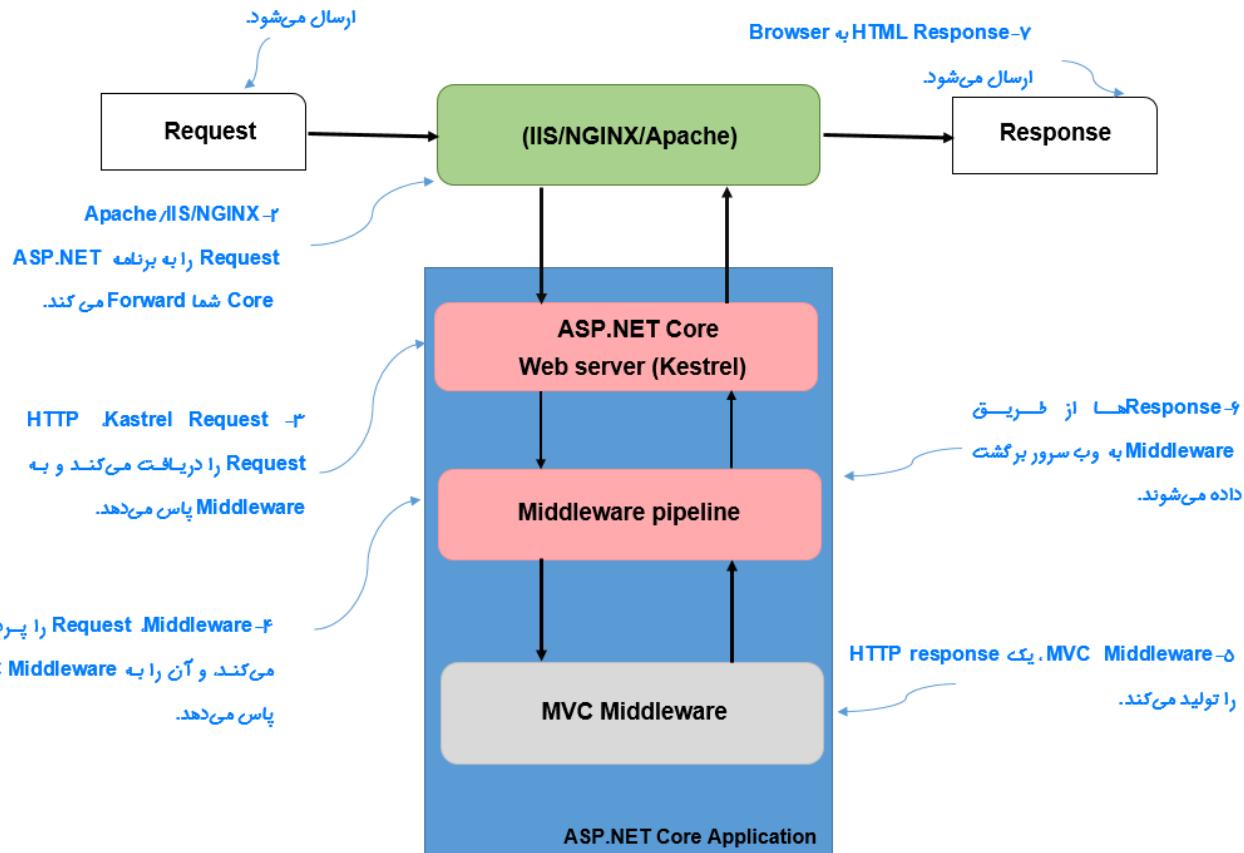
اپلیکیشن **ASP.NET Core** شامل بلوک‌هایی برای پردازش یک **Request** ورودی از **Browser** است.

هر **Request** به طور پیش‌فرض به **HTTP Server** ما می‌رسد (**Kestrel**)،

سپس HTTP server این درخواست را به Middleware Pipeline پاس می‌دهد و در اینجا، تغییراتی به آن Request داده خواهد شد.

سپس این تغییرات به MVC Middleware پاس داده شده، تا Response تولید شود. سپس این تغییرات به Browser از طریق Response تولید شده، از طریق Server به سرور برگشت دیده خواهد شد.

- یک HTTP Request به سرور



نکته!!

تنها HTTP Server موجود در ASP.NET Core نیست، اما می‌توان به قطعیت گفت که این Kestrel دارای Performance Server باشد و Cross-Platform Reverse Proxy است. و ها از IIS یک URL به عنوان می‌توانند انجام دهد (مثلاً : امنیت، Load Balance و...) را به سرانجام رسانیم.

کاربرد متدهای Startup در کلاس Configure

کامپوننت‌های کوچکی هستند که زمان دریافت یک **HTTP Request**، به صورت پشت سرهم اجرا می‌شوند. وظیفه‌ی متدهای **Configure**، تعریف رفتارهای اپلیکیشن شماست و باید هنگام دریافت این **HTTP Request**، موارد نیاز را معرفی نماید.

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

در متدهای **Configure**، یک اینترفیس **IApplicationBuilder** تزریق شده است که از طریق آن می‌توانید ترتیب اجرای **Middleware**‌ها را مشخص کنید.

نکته!!

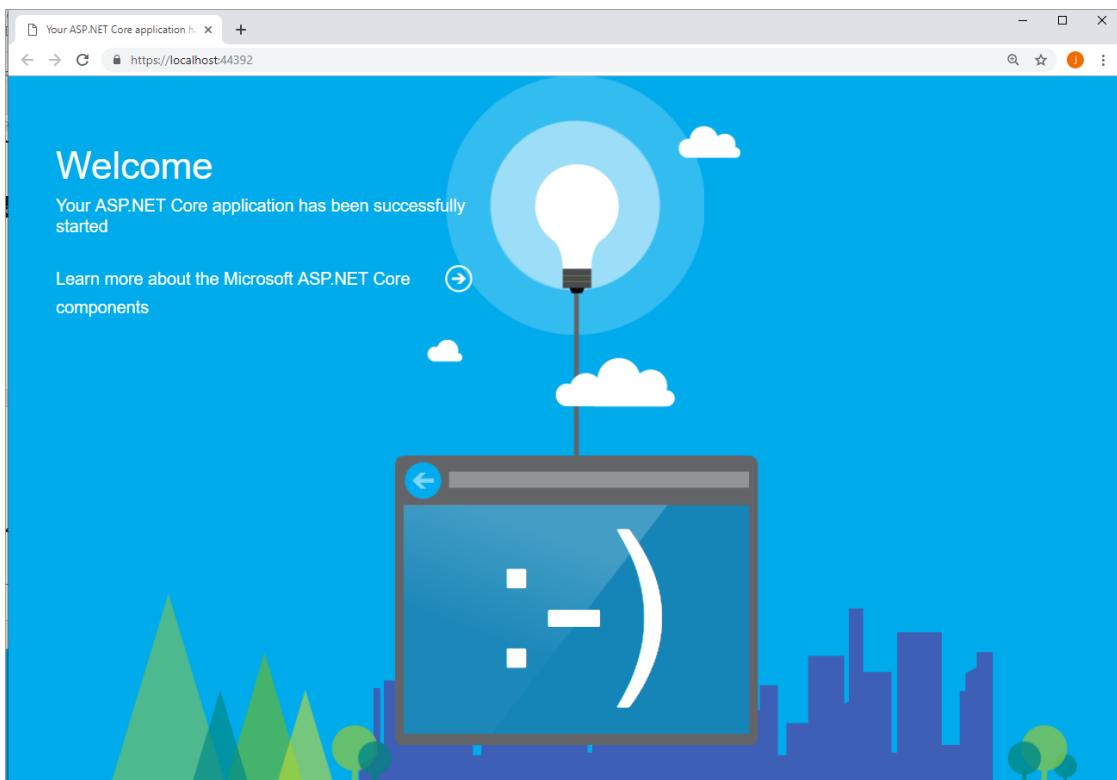
بیشتر **Middleware**‌ها به صورت **Extension method** برای اینترفیس **IApplicationBuilder** تعریف می‌شوند و معمولاً با عبارت **Use** شروع می‌شوند.

به مثال زیر دقت کنید:

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

کد فوق یک **Middleware** ساده برای نمایش صفحه خوش‌آمدگویی است؛ که در حالت پیش‌فرض به هر درخواستی پاسخ خواهد داد:

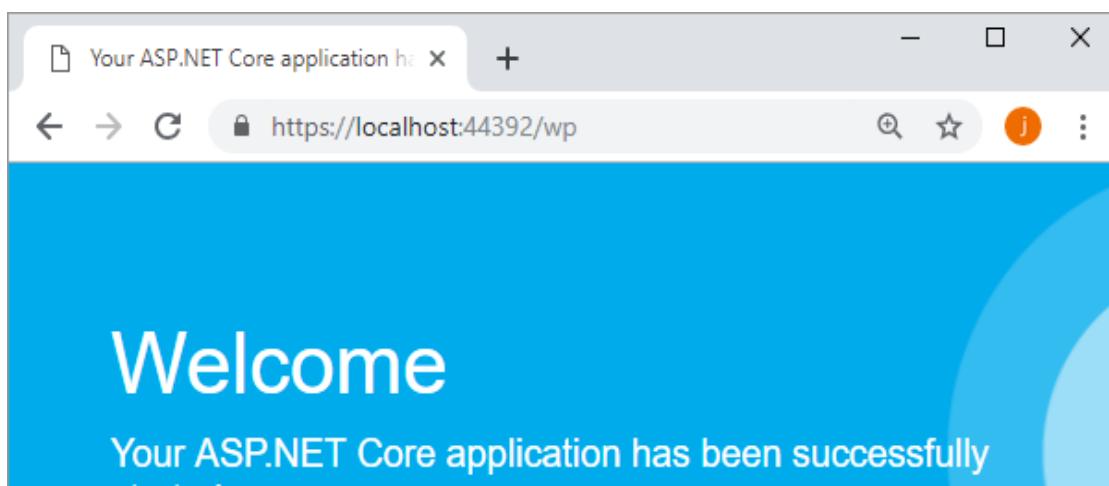
حالا با اجرای برنامه (F5) :



شما می توانید **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
});           ← خواهد گویی.
```

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



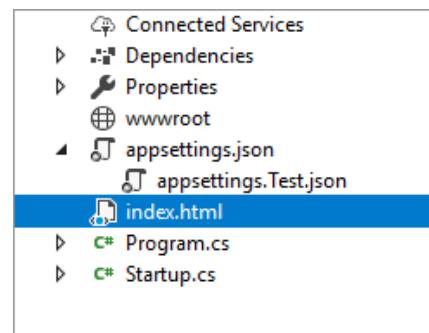
نکته!!

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

wwwroot

یکی از ویژگی‌های مهم هر وب‌اپلیکیشن، داشتن قابلیت استفاده از فایل‌های استاتیک است.

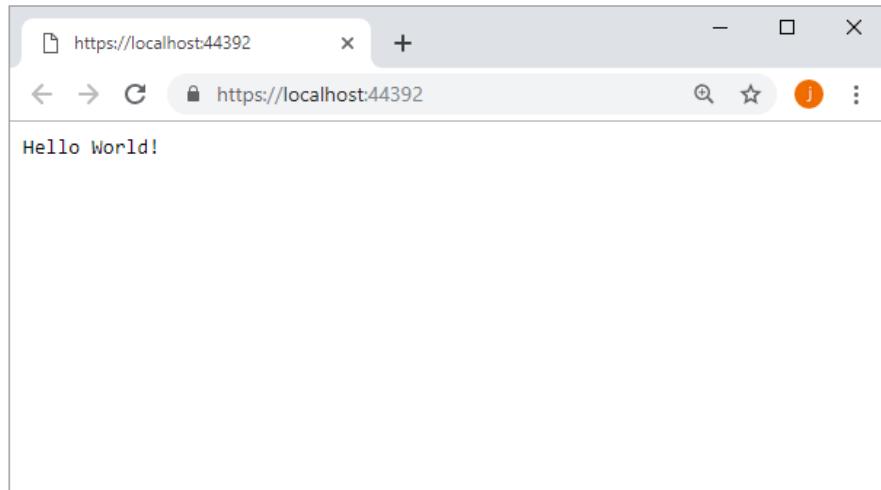
سعی کنید به این پروژه یک فایل `HTML` اضافه نمایید و سپس برنامه را اجرا کنید.



:index.html فایل

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

اجرای برنامه:

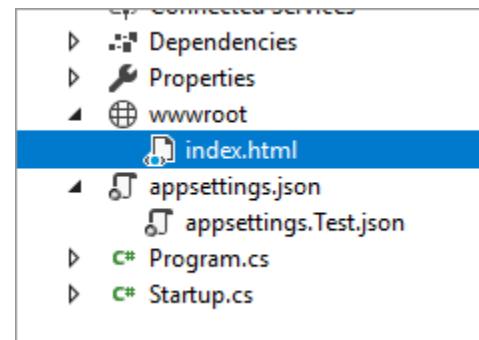


همانطور که می‌بینید محتوای فایل **Index.html** نمایش داده نشد.

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

۱. فایل‌های استاتیک باید درون **wwwroot ← Folder** قرار گیرند.
۲. درون **Pipeline** ما، هنوز **Middleware**‌ی جهت رسیدگی به درخواست نمایش فایل‌های استاتیک ثبت نشده است.

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



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

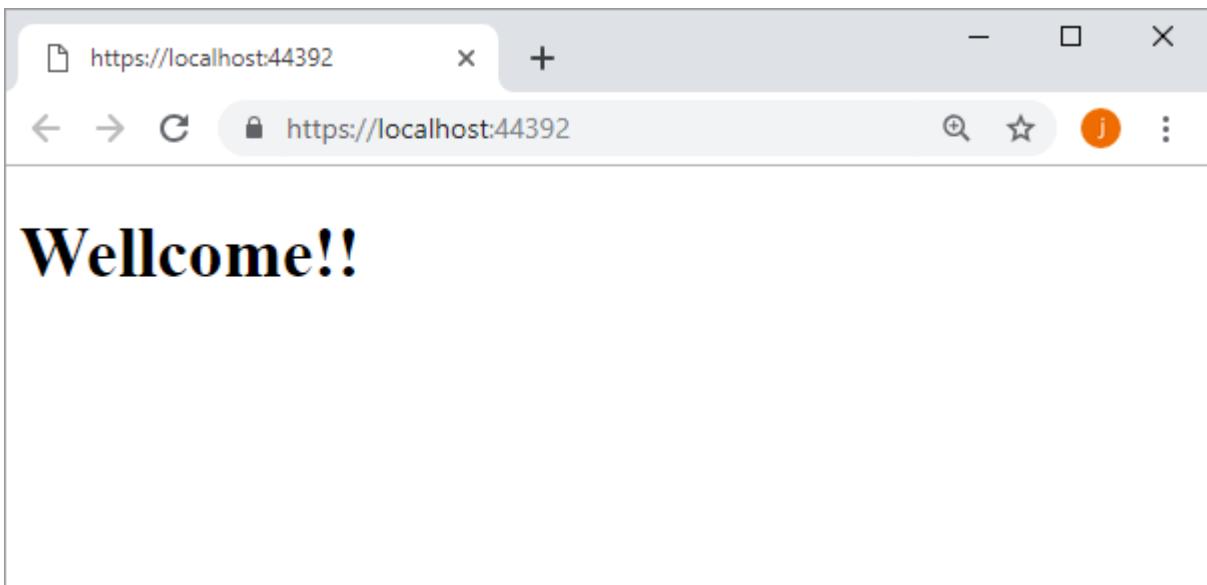
در حالت پیش‌فرض، **UseStaticFiles** به دنبال فایل‌های استاتیک درون **wwwroot ← Folder** می‌گردد. اگر **Middleware** در هنگام اجرای پروژه، فایل **index.html** به عنوان پیش‌فرض درنظر گرفته شود، باید بخواهیم دیگری با نام **UseDefaultFiles** را قبل از **UseStaticFiles** قرار دهیم:

```
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 wwwroot است.

یکبار دیگر برنامه را اجرا کنید و خروجی را ببینید.



همانطور که قبل گفتیم، ترتیب اجرای Middleware ها 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 فایل های استاتیک را انجام نمی دهد.

یک خبر خوب !!

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

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

ما حتی می توانیم به این Directory Browsing آپشن هایی مانند فعال سازی **Middleware** هم ارسال نماییم.

فصل دوم : مدیریت Exception ها

➤ مدیریت Status Code ها و Exception 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 Deploy و Release نمایید، کاربران چه عمدًا و چه سهوا، بالاخره راهی برای شکستن آن می‌یابند.

بنابراین در تولید اپلیکیشن، باید برای این‌گونه خطاهای 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 Option، یک Feature است. بنابراین مدیریت خطاهای، یک Feature است که شما باید به صراحت آن را در اپلیکیشن خود فعال نمایید.

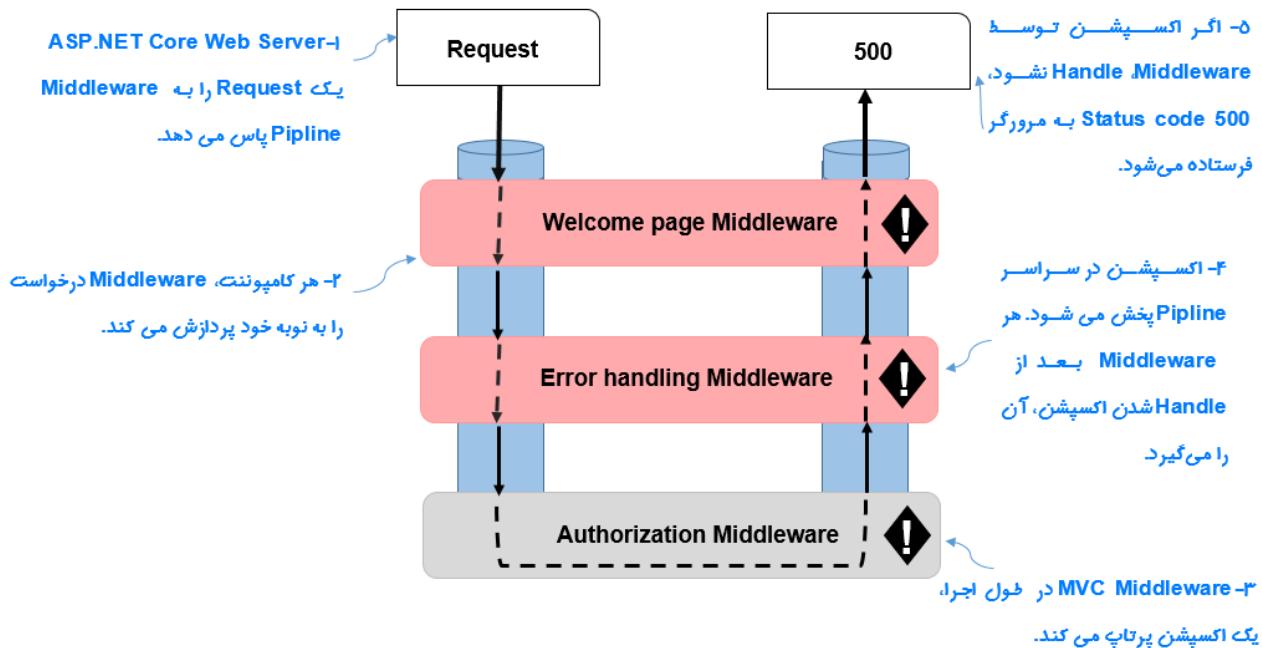
خطاهای متفاوتی می‌توانند در اپلیکیشن شما رخ دهند و راههای بسیاری برای بررسی آن وجود دارند، اما در اینجا می‌خواهم دو مورد از مرسوم‌ترین خطاهای را به شما نشان دهم:

Exception‌ها

و خطاهای Status code

Exception‌ها زمانی اتفاق می‌افتد که یک شرایط غیرمنتظره یافت شود. نمونه‌ای این است که بدون شک همه‌ی شما آن را تجربه کرده‌اید و زمانی رخ می‌دهد که تلاش برای دسترسی به Object‌ی دارید، که هنوز Initialized نشده است.

اگر یک Exception در یک کامپوننت Middleware اتفاق بیفت، (همانطور که در شکل زیر می‌بینید) در سراسر Pipeline منتشر می‌شود. اگر 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 باشد، زمانیست که مسیر **Handle Request** نمی‌شود. در این وضعیت خطای **404** را برمی‌گرداند. درنتیجه صفحه خطای **404** به کاربر نمایش داده می‌شود.

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

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

جزئیات خطای رخ داده را در یک صفحه کاربرپسند نمایش می‌دهد.

با توجه به اینکه هرگونه خطای رخ داده را با شکست روبرو کند، پس عجله کنید و هرچه سریع‌تر با توجه به اینکه هرگونه خطای رخ داده را با شکست روبرو کند، پس عجله کنید و هرچه سریع‌تر **Middleware Pipeline** را به **Error handling middleware** 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 محیط‌های اجرایی

همانطور که بالاتر گفتیم، یکی از نگرانی‌های برنامه‌نویسیان، مدیریت **Exception**‌هاست. اما برای مدیریت **Exception**‌ها، یکی از اصلی‌ترین مسائل، موضوع محیط‌های اجرایی است.

اینترفیس دیگری که در متدهای **Configure** کلاس **Startup** تزریق شده، اینترفیس **IHostingEnvironment** می‌باشد. هدف این اینترفیس، ارائه رفتارهای متفاوت، در انواع محیط‌های توسعه می‌باشد.

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

خوب‌بختانه با اینترفیس **IHostingEnvironment**، به این نگرانی پاسخ درست داده شده است. پس می‌توان گفت: شی **IHostingEnvironment** حاوی اطلاعات کاملی در مورد محیط فعلی ما می‌باشد.

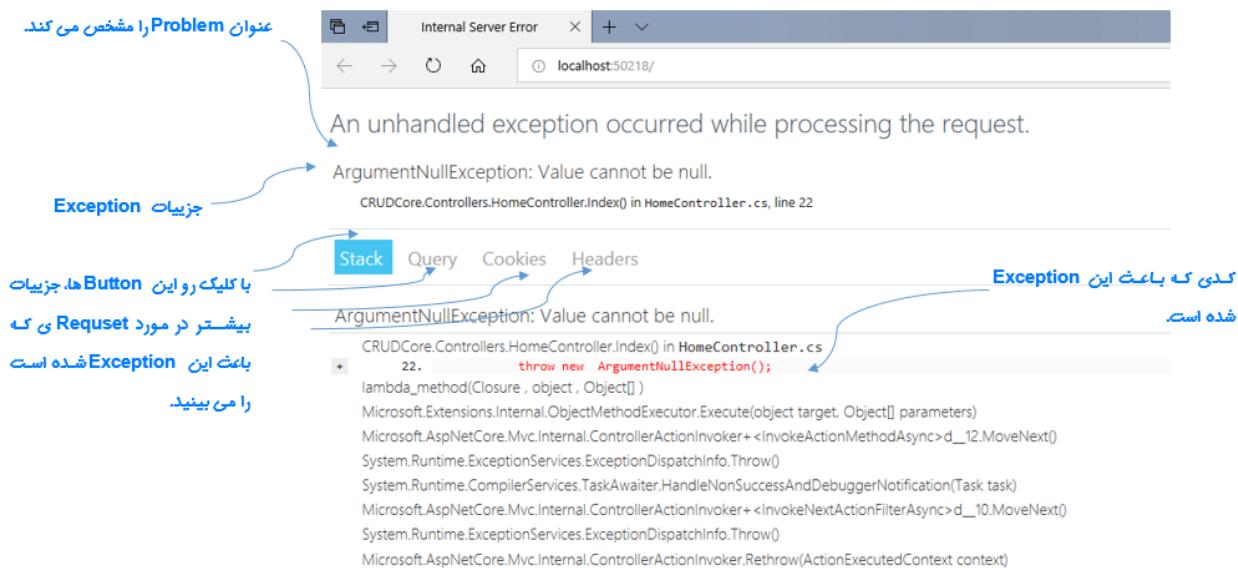
ما از طریق **IHostingEnvironment** به محیط‌های متفاوتی دسترسی خواهیم داشت و **Middleware Pipeline** بسته به اینکه در چه محیطی قرار داشته باشید، می‌توانند واکنش‌های متفاوتی را نشان دهند.

▪ با این **Middleware** **Configure** تضمین می‌کنیم، 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** برنامه‌نویس 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 می‌دهد درخواست به **Middlewares** 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 پایین)



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

در محیط اجرایی Exception

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 به همه نمایش داده می شود و هکرها راحت‌تر به اهدافشان می‌رسند.

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

خوبیختانه برای این موضوع، **Middleware** دیگری تعییه شده که شما می‌توانید در صورت بروز **Exception** پیغام را در صفحه‌ی سفارشی، به کاربر نمایش دهید.

با استفاده از **Middleware**‌یی به نام **UseExceptionHandler** این امکان به برنامه‌نویس داده می‌شود تا در صورت بروز یک **Exception**، کاربر با صفحه‌ای اختصاصی روبرو شود. این باعث می‌شود برنامه ما برای زمان‌های بحرانی کاربرپسندتر باشد.

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

```
{
```

در صورتیکه در محیط Development باشیم و یک Exception رخ دهد، صفحه‌ای برای نمایش جزئیات Exception نمایش داده می‌شود.

```
if (env.IsDevelopment())
```

```
{  
    app.UseDeveloperExceptionPage();  
}  
else if (env.IsProduction()) {  
    app.UseExceptionHandler("/home/error");  
}  
}
```

در صورتیکه در محیط Production باشیم و یک Exception رخ دهد، صفحه‌ای سفارشی شما نمایش داده می‌شود.



تعريف Environment

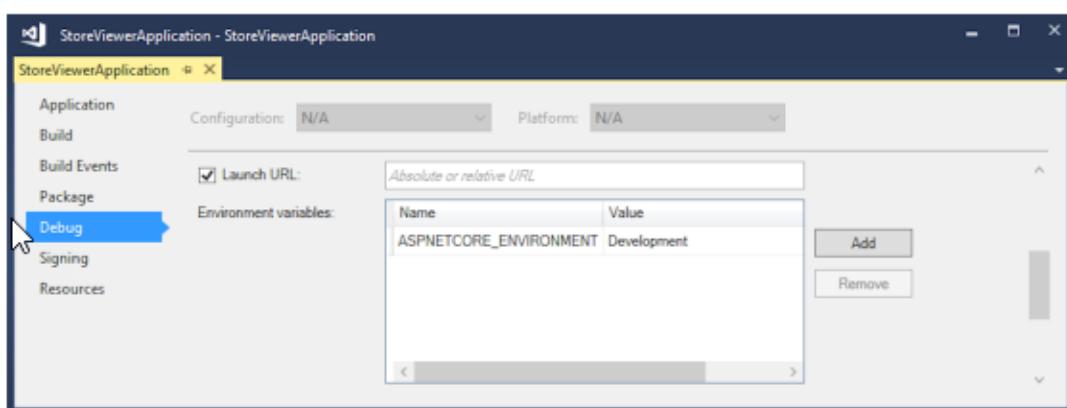
داشتن محیط‌های توسعه، باعث می‌شود شما برای هر هدفی، محیط اجرایی خودش را Handle کنید؛ مثلا برنامه در محیط تست با محیط تولید و.. باید متفاوت عمل نماید.

اما یک سوال؟

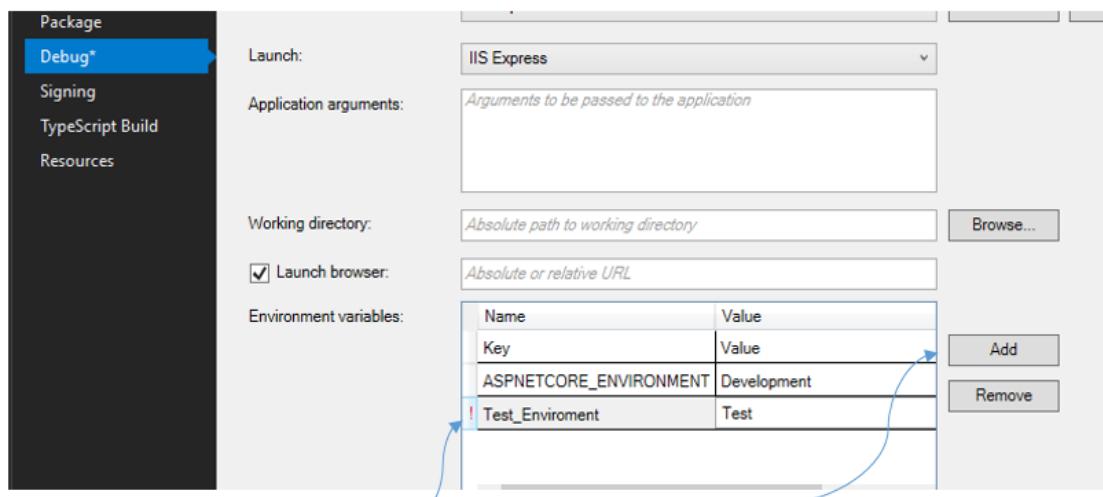
کجا می‌توان این محیط‌های توسعه را تعریف یا تغییر داد؟

چند روش برای تنظیم EnvironmentName و تعریف Hosting Environment 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!!

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

روش دوم: زمانیکه یک پروژه از نوع ASP.NET Core ایجاد می‌کنیم، Visual Studio یک فایل launchSettings.json می‌سازد که پیکربندی اپلیکیشن اضافه خواهد کرد. شما می‌توانید 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 }

```

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

```

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

  app.UseMvc(configureRoutes);
}

```

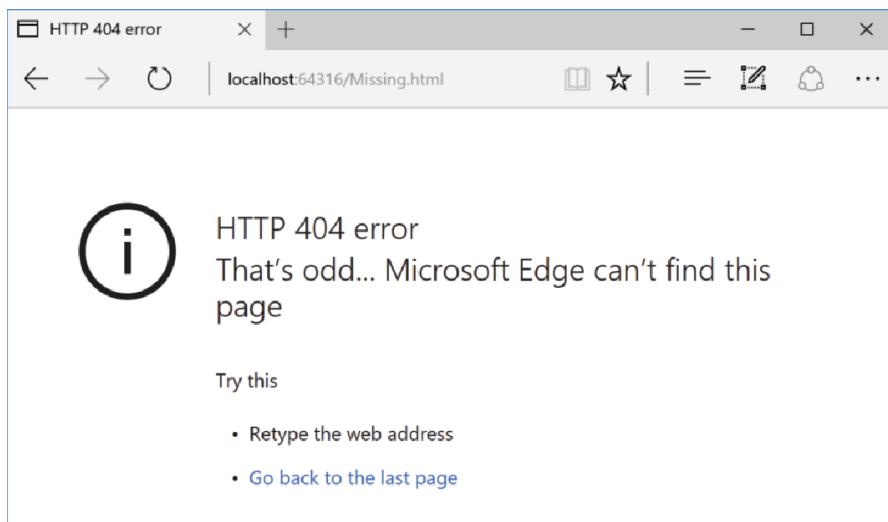
اگر در محیط **Test** باشید، صفحه‌ای سفارشی (در مسیر مشخص شده) برای نمایش خطا ظاهر خواهد شد.

مدیریت Status code ها

گزینه‌هایی مناسب، برای مدیریت `DeveloperExceptionPageMiddleware` و `ExceptionHandlerMiddleware` هاستند، اما نکته‌ای که باید بدانید، این است که، `Exception` ها همیشه در قلب خطا ظاهر نمی‌شوند؛ گاهی `Response` را در `HTTP error status code` یک `Middleware Pipeline` برمی‌گرداند که در این حالت، باید برای یک ارائه خوب به کاربر، `Status code` ها را هم مدیریت نمایید.

برنامه‌ی شما برای نمایش انواع خاصی از وضعیت‌های خطأ، می‌تواند طیف گسترده‌ای از `HTTP Status code` داشته باشد. به عنوان مثال: قبله دیدید که عدد 500 یعنی `Server Error` و زمانی اتفاق می‌افتد، که یک `Exception` رخ داده و هنوز `Handle` نشده است. یا خطای 404 یعنی "file not found" و زمانی رخ می‌دهد که کاربر یک `URL` نامعتبر وارد می‌کند و این `Handle` نمی‌شود.

بدون `Handle` کردن این `Status code` ها، کاربران صفحه‌ی خطای عمومی (مانند تصویر پایین) می‌بینند، که این می‌تواند کاملاً گیج‌کننده باشد.

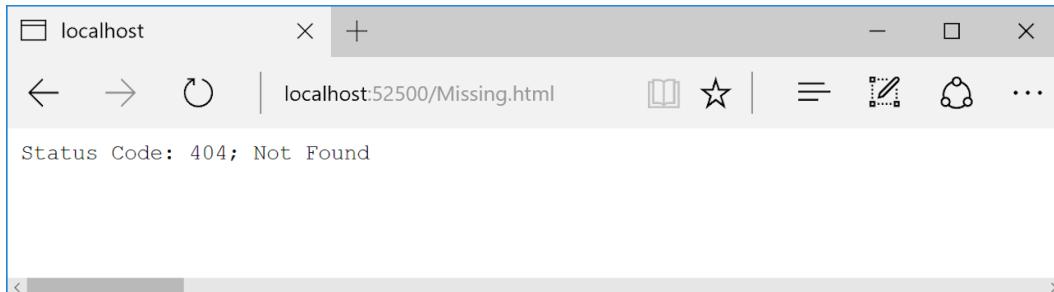


بهترین رویکرد برای حل این مسئله، مدیریت `Status code` ها و برگردن‌دن یک صفحه خطأ، با توجه به درخواست دریافتی می‌باشد. مایکروسافت هم با همین رویکرد `StatusCodePagesMiddleware` را تعریف کرده است.

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

در این متده، `HTTP StatusCode` هر آن با 4xx یا 5xx شروع شده باشد و هیچ `Intercept` را باشد `Response body` می‌کند.

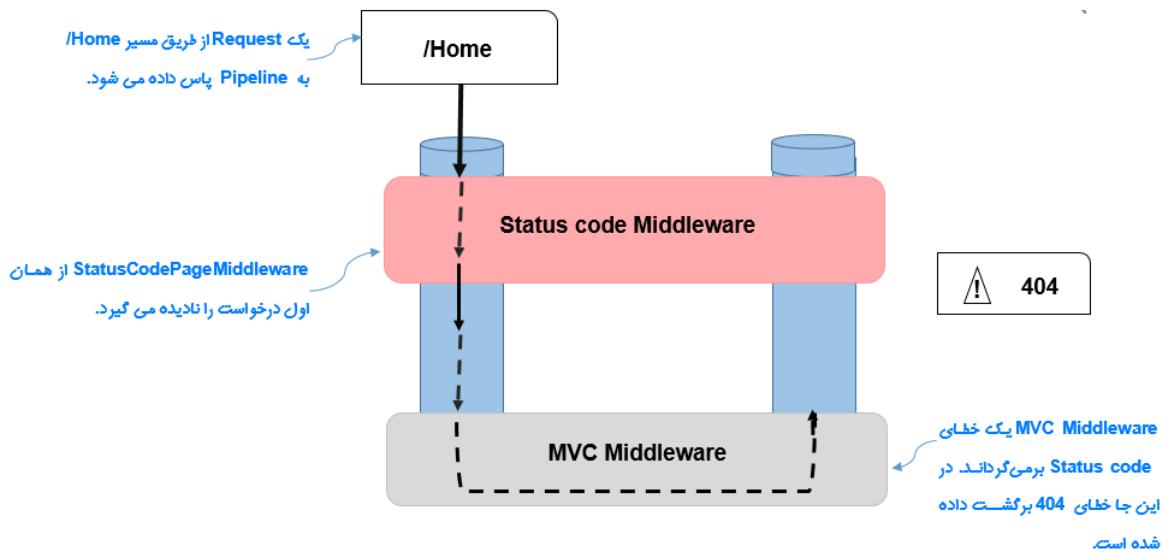
یک مثال ساده: جایی که شما هیچ گونه پیکربندی اضافه نمی‌کنید، **Middleware** (مانند تصویر پایین) یک متن شامل نوع و نام **Response** است را برمی‌گرداند.

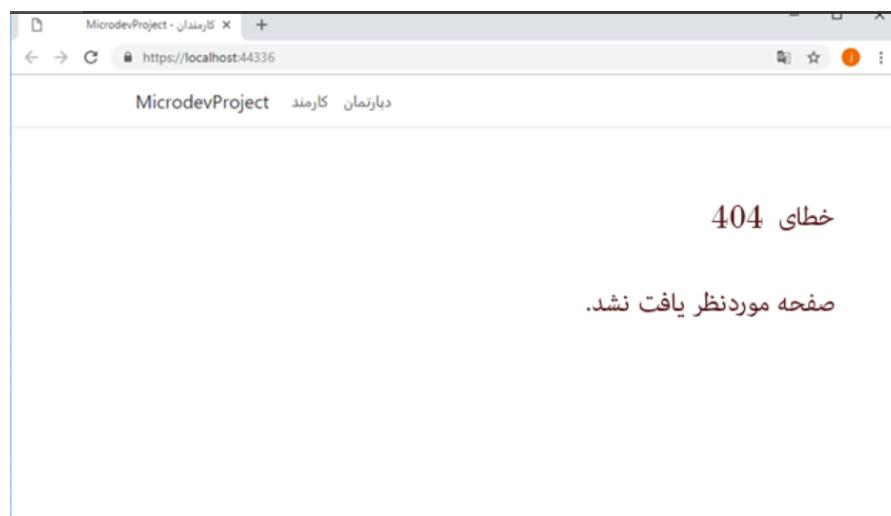


شاید با دیدن تصویر بالا، بگویید: دیدن این صفحه، از خطای عمومی که بالاتر دیدیم هم، بدتر است. اما نظر من این است که این روش می‌تواند شروع خوبی برای یک ارائه‌ی بهتر باشد.

هم، شبیه **ExceptionHandlerMiddleware** استفاده می‌شود. پس در این محیط به جای استفاده از متدهای **UseStatusCodePages()** از **app.UseStatusCodePagesWithReExecute("/error/{0}");**

این متدهای **Response code** می‌باشد، که هر بار **UseStatusCodePages** بر روی **Extension method** استفاده می‌کند. بین **4xx** یا **5xx** باشد، از **Error HandlingPath** استفاده می‌کند.





فصل سوم : مقدمات EF Core

Routing و MVC چیست؟

EF Core در مورد مقدمه‌ای

Dependency Injection در مورد مقدمه‌ای

DI طول عمر یک سرویس ایجاد شده توسط

MVC چیست؟

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

این الگو یکی از الگوهای محبوب در طراحی UI است.

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

۱- تمرکز بر روی **Single Responsibility** بیشتر می‌شود.

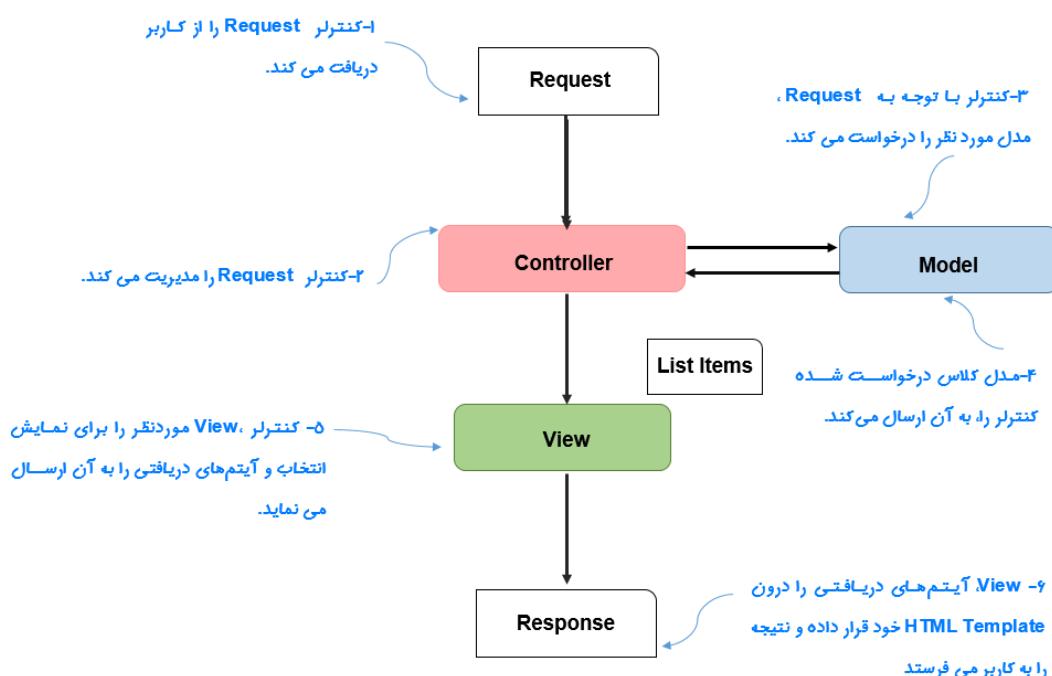
۲- وابستگی متقابل در سیستم کاهش می‌یابد.

به طورکلی سه مفهوم، **MVC Design Pattern** را می‌سازند:

▪ **Model**: داده‌هایی که باید نمایش داده شود.

▪ **View**: صفحه‌ای که داده‌های ارائه شده توسط مدل را نشان می‌دهد.

▪ **Controller**: مدل را **Update** و **View** موردنظر را نمایش می‌دهد.



مراحل انجام کار:

- ۱) دریافت می‌کند.
- ۲) بسته به Controller، Request اطلاعات درخواست شده را از Model دریافت، یا اینکه Update Model را نماید.
- ۳) View، Controller را جهت نمایش و انتقال Model انتخاب می‌کند.
- ۴) View از اطلاعات موجود در Model برای UI استفاده می‌نماید.
- ۵) سپس این HTML به عنوان یک HTTP Response به کلاینت ارسال خواهد شد.

Routing چیست؟

برای مدیریت منطق‌های پیچیده‌ی اپلیکیشن شماست و با Invoke کردن یک متده، یک مناسب را Handle می‌نماید. حالا یکی از سوال برانگیزترین جنبه‌های این Middleware، این است که چطور یک متده برای اجرای یک Request انتخاب می‌شود؟

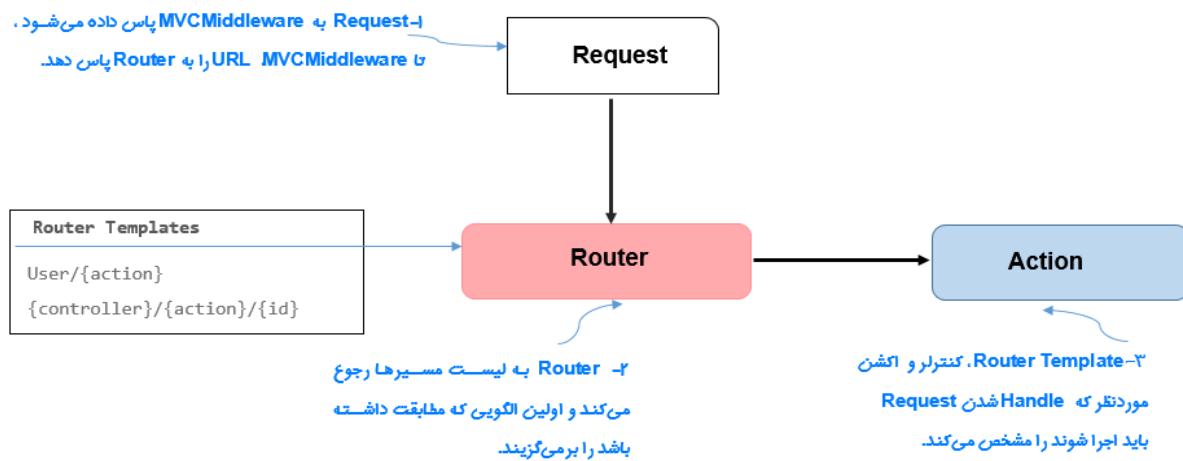
این جوابی است که Routing به ما می‌دهد:

Routing (یا مسیریابی) قسمت کلیدی MVC Design Pattern در ASP.NET Core می‌باشد که ورودی را به یک Controller-Action خاص متصل می‌کند.

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

به عنوان مثال:

آن Create View و EmployeeController Router که به URL →/Employee/Create می‌گردد.



حالا که با این مفهوم آشنا شدید، بیایید قابلیت **Routing** را در کلاس **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?}");
            });
        }
    }
}
```

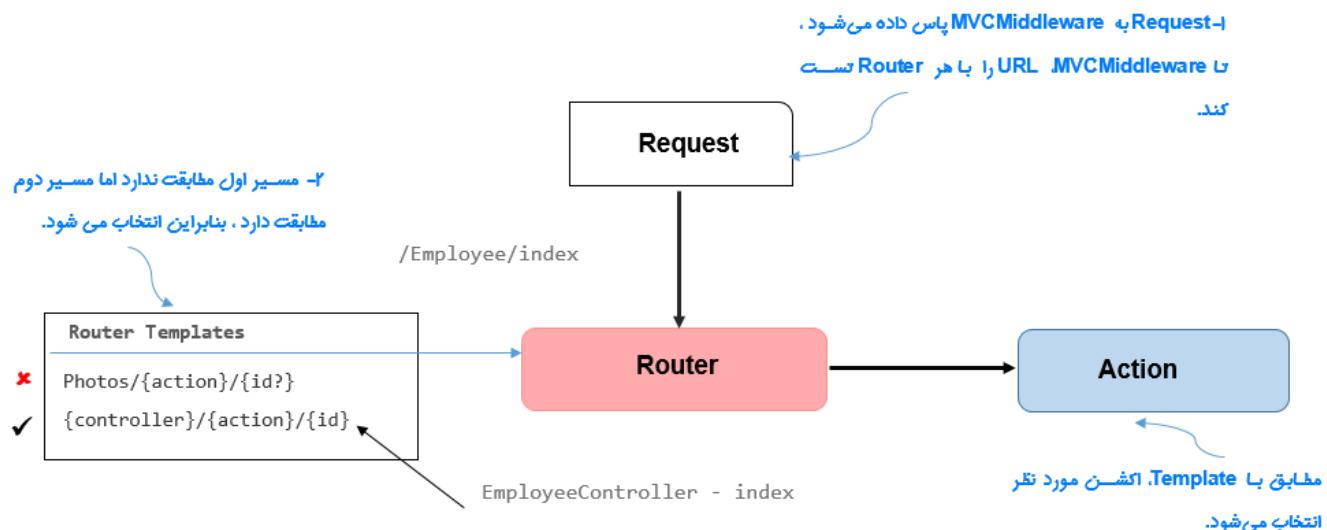
الگوی Route ساختاری برای انتساب با URL تعریف کنند.

}

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

شما می‌توانید با قالب گفته شده بالا، **Routing Pattern**‌های زیادی را اضافه نمایید که هر کدام می‌تواند یک عدد را برای `Map` کردن URL‌های مختلف به اکشن‌متد موردنظر تنظیم کند.

قالب‌های مسیریابی، ساختار URL مشخص شده را، در اپلیکیشن شما تعریف می‌کنند. به طور مثال: URL‌های `Employee /{action=index}` و `Employee /Employee/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 اکشن‌متد را در **Controller** مربوطه می‌یابد و سپس **Middleware** از این اکشن‌متد 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**.

به‌طور پیش‌فرض مسیریابی ذکر شده بالا، به عنوان یک **مسیریابی عمومی** در اپلیکیشن‌ها استفاده می‌شود. در این روش، باید **Controller**‌ها و اکشن‌متد‌های خود را تعریف نمایید تا **Request** دریافتی، مطابق با قرارداد حرکت و بالاخره **مسیریابی** موردنظر خود را بیابد.

اما در **AttributeRouting** باید عبارت **[Route]** را بر روی اکشن‌متد‌های خود قرار دهید. این روش **انعطاف‌پذیری** بیشتری دارد و برای زمانیکه از **Web Api** استفاده می‌کنید، بسیار کارآمد است.

```
[Route("[Employee]")]
public class EmployeeController: Controller
{
    // Route Template برای Route تعیین
    [Route("")]
    // Default بودن آن است.
    public IActionResult Index()
    {
        var model = new Employee();
        //..
        return View(model);
    }
    // AttributeRouting برای
    [Route("[Create]")]
    // مسیریابی
    [HttpGet]
    public IActionResult Create()
    {
        return View();
    }
}
```

توجه داشته باشید: در هر کدام از روش‌های ذکر شد، باید **URL**‌های موردنظر خود را با استفاده از قالب **مسیریابی** تعریف نمایید. اگر در این قالب چیزی برای **Route** تعیین نشود، به معنای **Default** درنظر گرفته

می شود. (عنی: با اجرای شدن این **Controller** به صورت پیش فرض متدى که ["] را داشته باشد اجرا خواهد شد).

حالت دیگری از **AttributeRouting**, استفاده از توکن است. در این حالت دیگر نیازی به نوشتند نام **Controller** یا اکشن متده نیست.

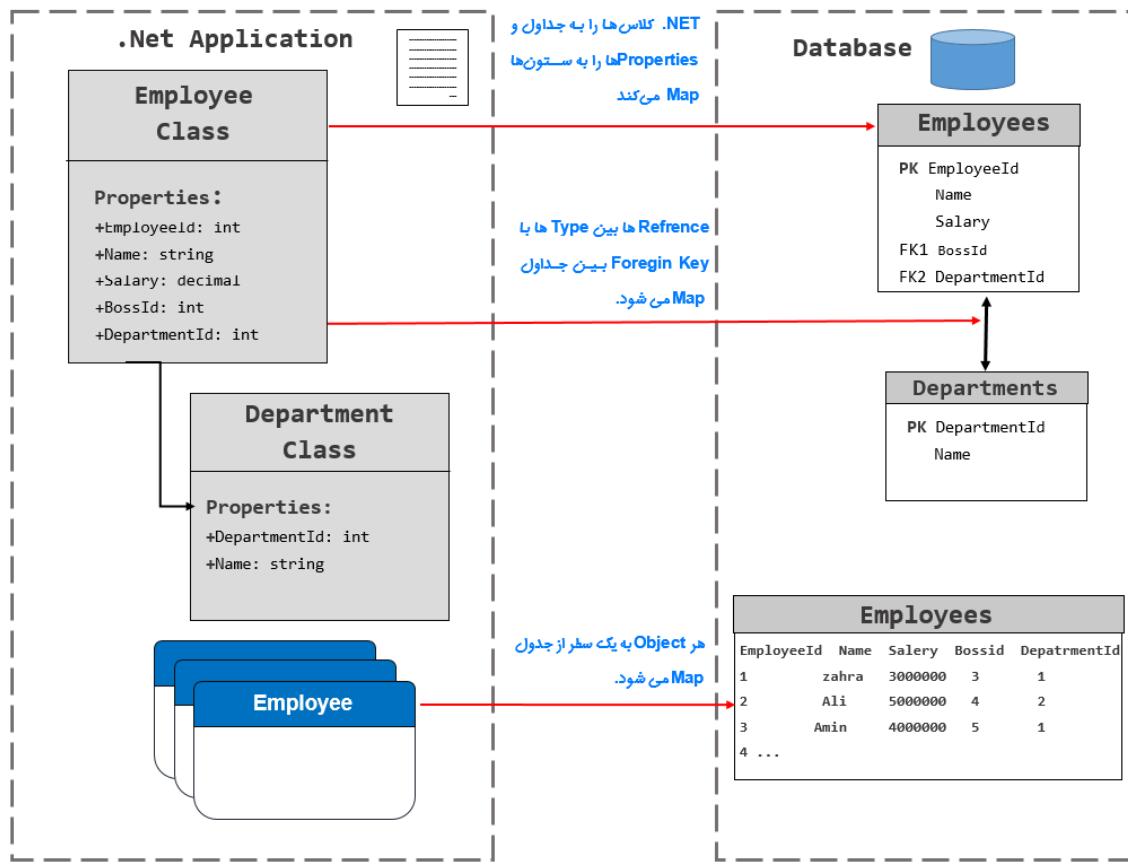
```
[Route("[controller]/[action]")]
public class EmployeeController: Controller
{
    public IActionResult Index()
    {
        var model = new Employee();
        //...
        return View(model);
    }

    [HttpGet]
    public IActionResult Create()
    {
        return View();
    }
}
```

مزیت نوشتند توکن این است که، اگر بعداً نام **Controller** را تغییر دهیم، فریمورک آن را تشخیص خواهد داد، به همین خاطر به صورت صریح نام کنترلر را ننوشته ایم.

EF Core مقدمه ای در مورد

کتابخانه ای برای دسترسی به دیتابیس است که به عنوان یک **ORM** عمل می نماید و با **Map** کردن کلاس ها و اشیاء دات نت به اجزای دیتابیس (مثل جداول و ردیفها) بین آنها ارتباط برقرار می کند.



مهمترین ویژگی EF core برای Developerها، 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، EF Core بخشی از نام EF Core می‌باشد. البته در همینجا بگوییم که یکی از مهمترین ویژگی‌های Open Source .NET Core بودن آن است.

نسخه‌ی اولیه‌ی 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.

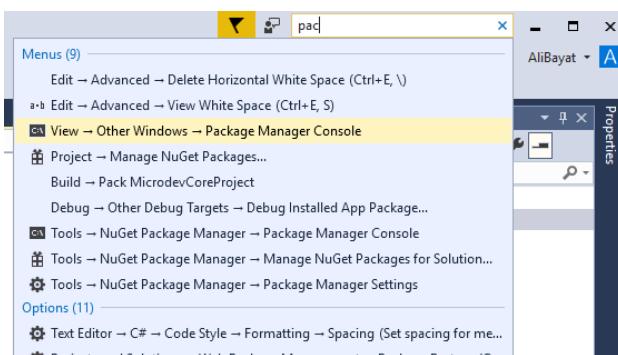
اضافه کردن دیتابیس به پروژه

برای سادگی این پروژه، می‌خواهیم 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 نماییم.

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

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

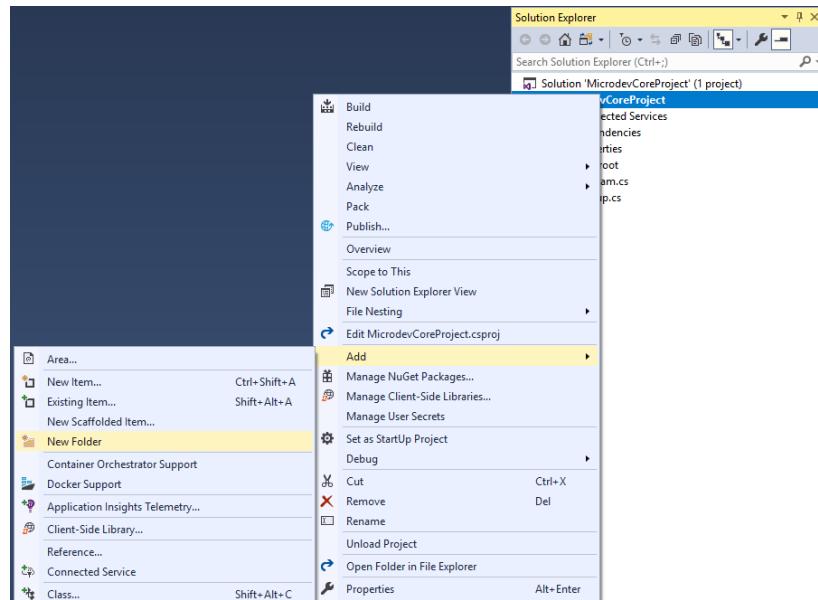
- قبل از هر کاری باید **EF Core** را به پروژه اضافه کنید. برای انجام این کار، مانند تصویر زیر **Package Manager Console** را باز نمایید.



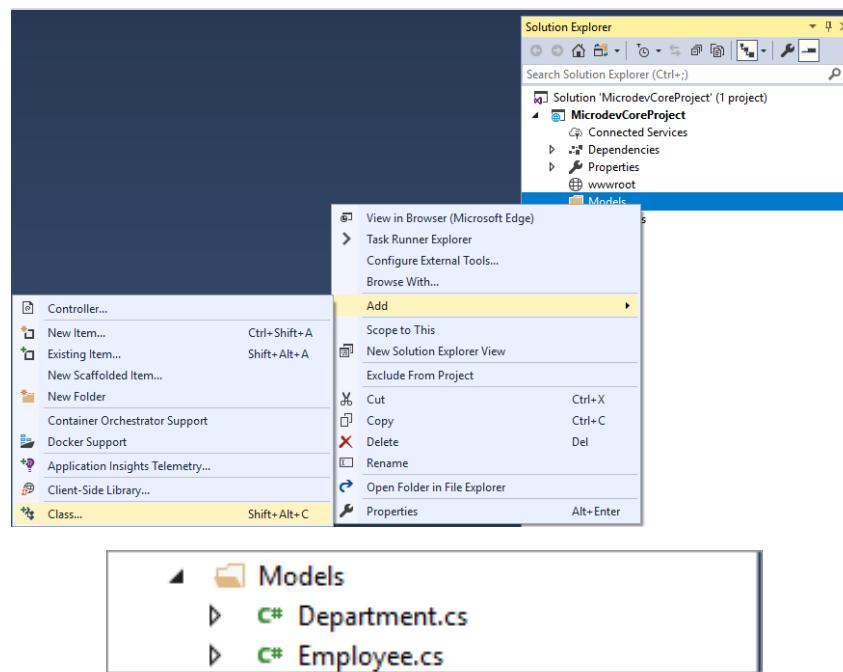
- حالا دستور زیر را در آن تایپ و درنهایت **Enter** را بزنید.

Install-Package Microsoft.EntityFrameworkCore.Tools

در سمت راست بر روی **Solution** راست کلیک نمایید و در منوی مربوطه با انتخاب گزینه **.NET Models** فلدری با نام **Add→Folder** ایجاد نمایید.



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



چیست؟ DataAnnotation

داده‌ها قبل از ارسال باید اعتبارسنجی شوند و این، موضوع خیلی مهمی در ذخیره‌سازی 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 **Property**‌ها در **Model** مطابق با این قوانین عمل نمایند.

شیوه‌ی کار بدین صورت است که، **Metadata**‌ها **DataAnnotation**‌هایی، برای توصیف داده آماده می‌کنند و داده‌ها، باید از قوانین این **Metadata**‌ها پیروی نمایند.

به عنوان مثال:

به جای اینکه **null** بودن داده‌ها به صورت دستی چک شود، **[Required]** را بالای **Property**‌های خود بگذارید و دیگر خیالتان راحت باشد که نمی‌توان داده‌ی **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** 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 بیفرایید.

```

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) باشد و یک رئیس داشته باشد. پس:

▪ **Department** به عنوان کلید خارجی برای ارتباط **Employee** با **DepartmentId** است.

▪ هم، یک کلید خارجی از جدول **Employee** به **BossId** خودش است.

DbContext

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

در **Employee** کلاس‌های شما به صورت **DbSet<T>** تعریف می‌شود. T درواقع **Entity** ما می‌باشد.
▪ **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.
▪ **Aين کلاس باید از کلاس DbContext ارث بری نماید و جهت تعریف جدول‌های شما در دیتابیس، باید شامل مجموعه‌هایی از نوع **DbSet**، باشد.**

یک `DbContextOptions<T>` دارد که پارامتر `Constructor` در `ASP.NET Core` `DbContext` های دیتابیس را تعریف می کند.

```
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; }
    }
}
```

ارث بری از `DbContext` در `Constructor` شامل جزییاتی از جمله: `ConnectionString` می باشد.
تعریف جداول شما در `DbSet`'ها داده شده است.
دیتابیس

ConnectionString چیست؟

زمانیکه می خواهید اپلیکیشن تان را توسعه دهید و در ماشین های مختلف مستقر نمایید، موضوع مشخص کردن مکان دیتابیس مطرح می شود.

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

برای مثال:

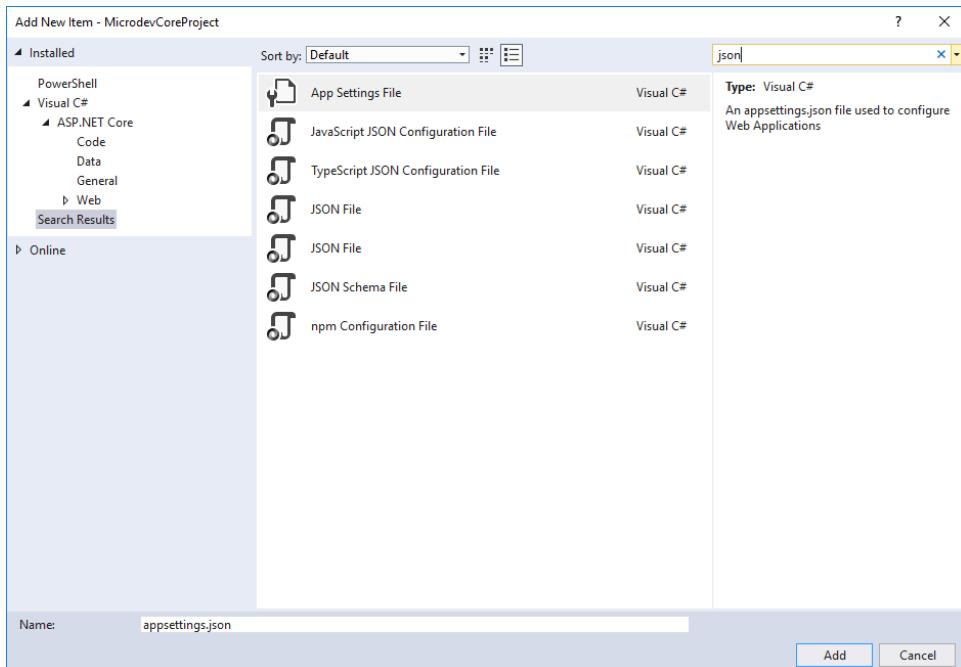
در اپلیکیشن های وب، به طور معمول محل دیتابیس، بر روی یک `Host` قرار دارد (جایی که کاربران واقعی به آن دسترسی داشته باشند) و درون سخت افزار شما نیست. بنابراین **مکان** و **تنظیمات** مختلف دیتابیس معمولاً در یک `ConnectionString` ذخیره می شود.

به فرمورک می گوید، 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 کامپیوتر های مختلف مشخص کنیم.

بیایید به پروژه خود، فایلی به نام `appsettings.json` اضافه و در آن `ConnectionString` را مشخص کنیم.

➤ روی پروژه راست کلیک نمایید و گزینه `Add→NewItem` را انتخاب کنید.

حالا در کادر باز شده و در قسمت **Search File** کلمه **json** را تایپ و سپس **Search** را انتخاب و در پایان **OK** نمایید.



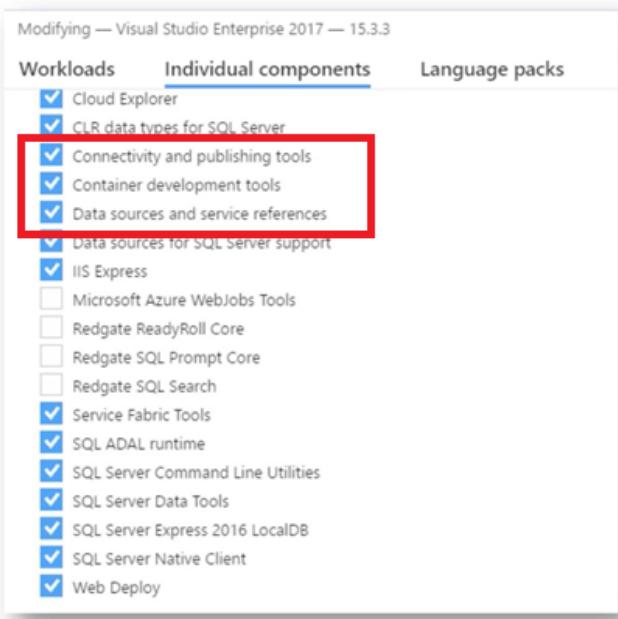
شما می‌توانید **ConnectionString** را به سلیقه خود سفارشی نمایید. البته باید توجه داشته باشید، نام این **ConnectionString** بعدا در معرفی دیتابیس موردنیاز است.

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

!!نکته!!

یک نسخه از **SQL Server** با **Visual Studio 2017** همراه 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 رجیستر کردن DbContext کلاس Startup (در متدها ConfigureServices) می‌رسد.
اما قبل از انجام این کار، می‌خواهیم کمی در مورد DI صحبت کنم.

مقدمه‌ای در مورد Dependency Injection

ممکن است قبلاً 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 برای دریافت هر سرویسی از DI استفاده می‌کند.

یک 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 ماس است. این تغییرات به مرور زمان باعث خراب شدن پایه‌های اپلیکیشن می‌شود. آقای رابرت سسیل مارتین معروف به عمومی 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 مأذول نرم افزاری می بایست تنها یک دلیل برای تغییر داشته باشد.
- اصل دوم O - OCP – Open Closed Principle : مأذول های نرم افزار باید برای تغییرات بسته و برای توسعه باز باشند.
- اصل سوم SubClass : L – LSP – Liskov Substitution Principle : ها باید بتوانند جایگزین نوع پایه‌ی خود باشند.
- اصل چهارم ISP– Interface Segregation Principle : I-ا: کلاینت‌ها نباید وابسته به متدهایی باشند که آنها را پیاده سازی نمی‌کنند.
- اصل پنجم D – DIP– Dependency Inversion principle : مهم‌ترین اصل که می‌گوید: مأذول های سطح بالا نباید به مأذول های 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 مأذول ها را کمتر می‌کنند. با کم شدن این وابستگی‌ها، تغییر یک مأذول تمام برنامه را تحت تاثیر قرار نمی‌دهد.

فریمورک ASP.NET Core برای مأذول از بھترین شیوه‌ی مهندسی نرم افزار، یعنی اصول SOLID پیروی می‌کند. اصول SOLID یکی از بھترین تجربه‌ها در طول دوران برنامه‌نویسی شی گرا است.

DI مزایای

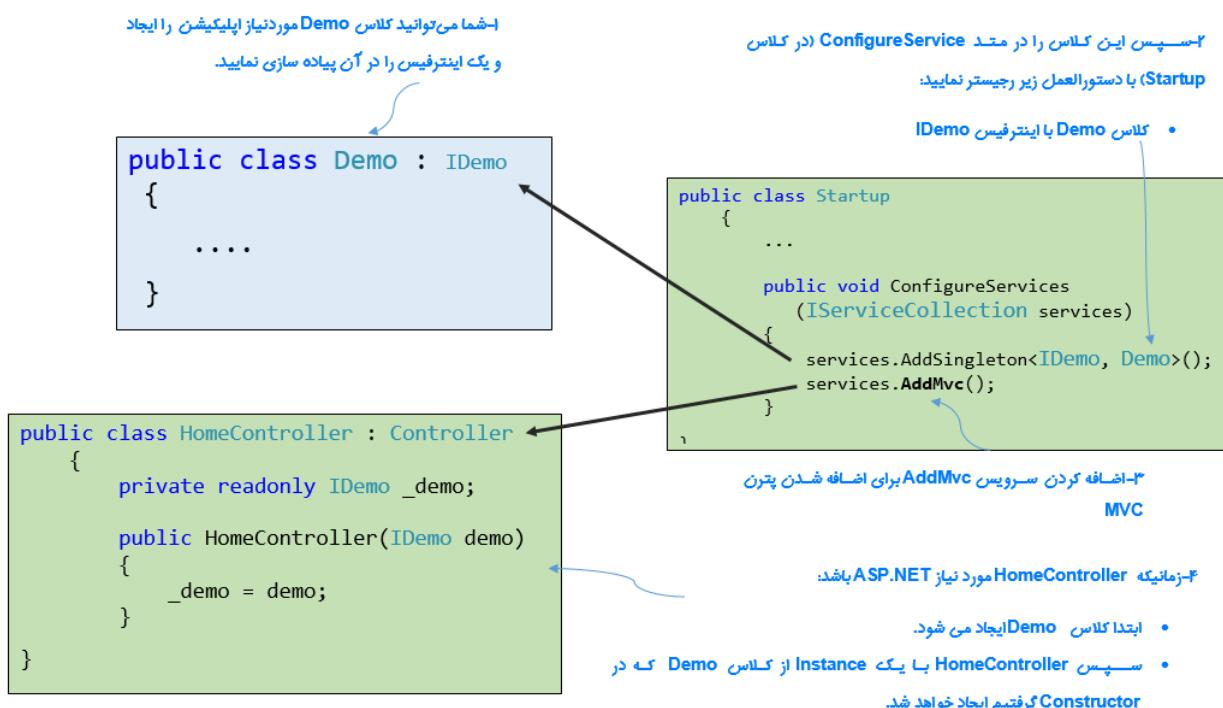
- DI به اپلیکیشن شما این امکان را می‌دهد، تا به صورت داینامیک با خودش پیوند داشته باشد. Provider به این صورت عمل می‌نماید که درخواست شما را دریافت و سپس با توجه به این درخواست، کلاس موردنظر را ایجاد می‌نماید.

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 Tighty Coupled ایجاد نمایید.

- استفاده از DI و اینترفیس‌ها باعث می‌شود اپلیکیشن، **Coupling Loosely Coupled** 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.

مثال: در برنامه تجاری خود بخواهید یک کارت اعتباری ساختگی به جای کارت اعتباری واقعی داشته باشید و این موضوع، 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 Inject یا شود.

در تصویر بالا: متدهای `AddMvc` و `Configure` که در کلاس `Startup` می‌توانند سرویس `IDemo` را در کلاس `HomeController`登録 کنند.

در این مثال شما اینترفیس `IDemo` را در `HomeController` کلاس `Constructor` تزریق کردید و DI یک از کلاس `Demo` از کلاس `Instance`

طول عمر یک سرویس ایجاد شده توسط DI

طول عمر یک سرویس یعنی، قبل از ایجاد شدن `Instance` از سرویس، مدت زمان زندگی ماندن سرویس را در `Container` مشخص کنید.

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

چقدر طول می‌کشد که این `Instance` ایجاد شده `Dispose` شود؟

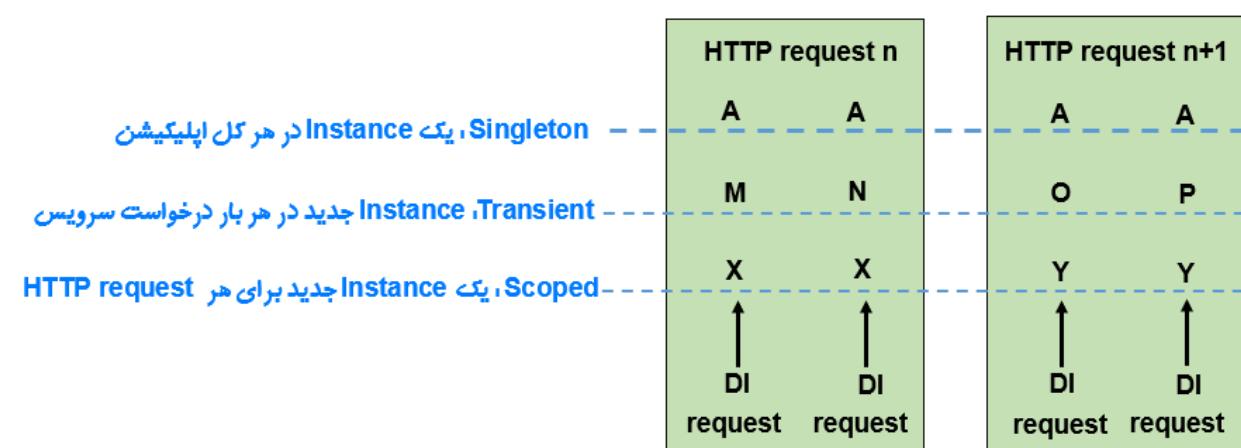
هر گاه از DI، یک سرویس رجیستر شده، درخواست شود یکی از دو مورد زیر اتفاق می‌افتد:

- یک `Instance` جدید از آن سرویس ایجاد و بازگشت داده می‌شود.
- یک `Instance` موجود از سرویس بازگشت داده می‌شود.

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

این سخن تصدیق می‌کند، زمانی که یک سرویس ایجاد می‌شود، DI Container، می‌تواند `Instance` موجود از سرویس را، استفاده مجدد نماید.

یک سرویس می‌تواند در سه طول عمر در DI Container زندگی کند.



انواع طول عمر

- 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.
 - در این طول عمر، به ازای هر درخواست HTTP، یک نمونه از سرویس ایجاد می‌شود.
 - 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**登録しました。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، حتماً 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.

فصل چهارم : ایجاد دیتابیس

- » رجیستر کردن DbContext از طریق DI
- » ایجاد Upadat و Migration
- » تعریف Seed

رجیستر DI از طریق DbContext

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

قبل از رجیستر 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 کرم. پس باید پکیج Microsoft.EntityFrameworkCore.SqlServer در اپلیکیشن اضافه شود.

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

```
Package Manager Console
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، لذا چطور می‌توان یک فایل Json را در بدنه برنامه بخوانیم؟

پاسخ این سوال را IConfigurable 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 `Configuration` توسط `appsettings.json`

در کلاس `Startup` 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?}");
            });
        }
    }
}
```

```
}
```

```
}
```

```
}
```

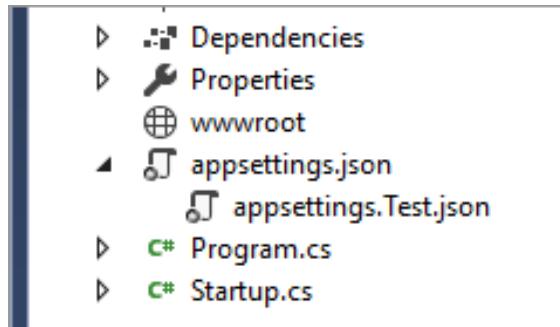
IConfiguration service و appsettings.json

- اولین نکته: این سرویس به طور پیش‌فرض جستجوی خود را در فایل `appsettings.json` آغاز می‌کند، اگر در این فایل به نتیجه‌ای نرسد به دنبال سایر سورس‌های ذکر شده خواهد رفت.
- دومین نکته: ما می‌توانیم درون پروژه، چندین فایل `appsetting` داشته باشیم. در حالت پیش‌فرض اپلیکیشن از فایل `appsettings.json` استفاده می‌کند؛ اما `ASP.NET Core` فایل دیگری با الگوی نام‌گذاری زیر را هم جستجو خواهد کرد:

appsettings.`EnvironmentName`.json

ما با استفاده از الگوی بالا می‌توانیم تنظیمات را برای محیط‌های مختلف سفارشی کنیم. شاید بپرسد چگونه؟ یک فایل با نام `appsettings.Test.json` ایجاد کنید و تنظیمات `ConnectionString` 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 می‌شود.



- دومین نکته: علاوه بر `StringConnection` می‌توانید، 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);
        });
    }
}
```

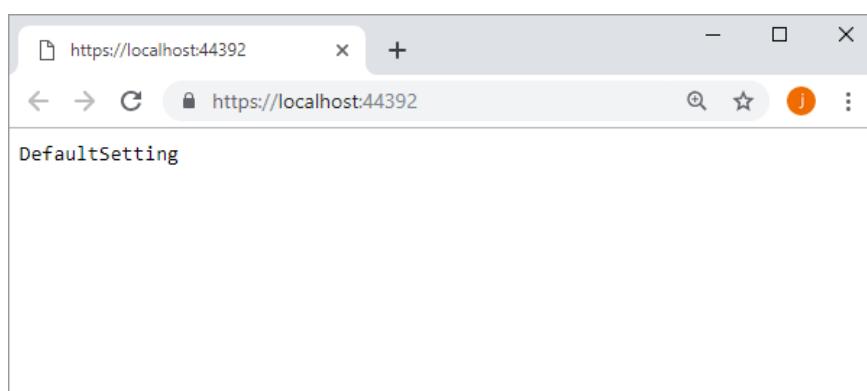
خواندن مقدار کلید

TestSetting

فرستادن متغیر setting به خروجی.

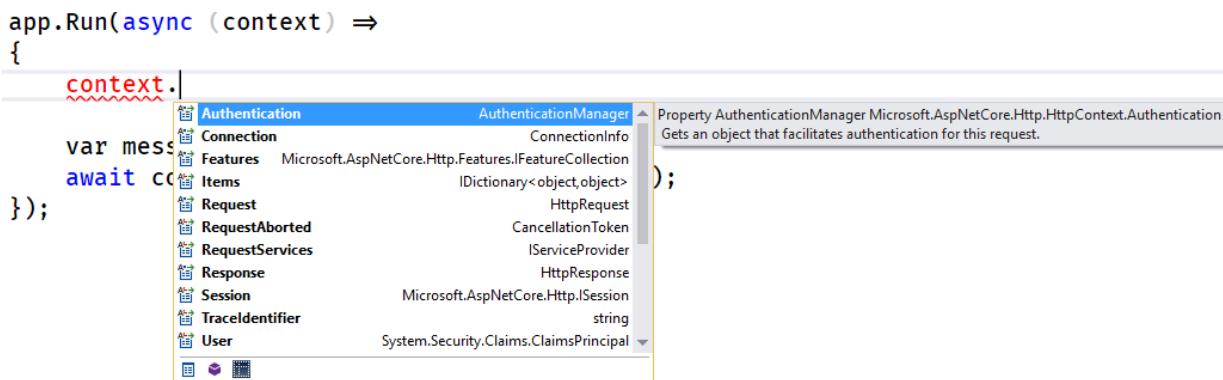
خروجی.

بعد از اجرای برنامه (F5):



نکته!!

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



۲- مرحله بعد، استفاده از **DbContext** برای معرفی دیتابیس به برنامه است.
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)
{
    مشخص کردن یک
    Database Provider
    services.AddDbContext<MicrodevDbContext>(option =>
        option.UseSqlServer(configuration.GetConnectionString("MicrodevCo
nnection")));
    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?}");
    });
}
}

```

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:

- برای هر بار دسترسی به دیتابیس، باید کد دسترسی را تکرار کنیم.
- رشته دسترسی به پایگاه داده ثابت می‌شود و اگر زمانی بخواهید سایت خود را جایی دیگر Host نمایید، به مشکل برخواهید خوردید، چون مکان دیتابیس برای Host دیتابیس، متفاوت از دیتابیسی است که در محیط توسعه استفاده می‌کنید.

از لحاظ فنی چون **DbContext**، اینترفیس **IDisposable** را پیاده‌سازی کرده است، به صورت خودکار **Dispose** می‌شود 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 می‌رسد. اما دیتابیس چطور ایجاد می‌شود؟

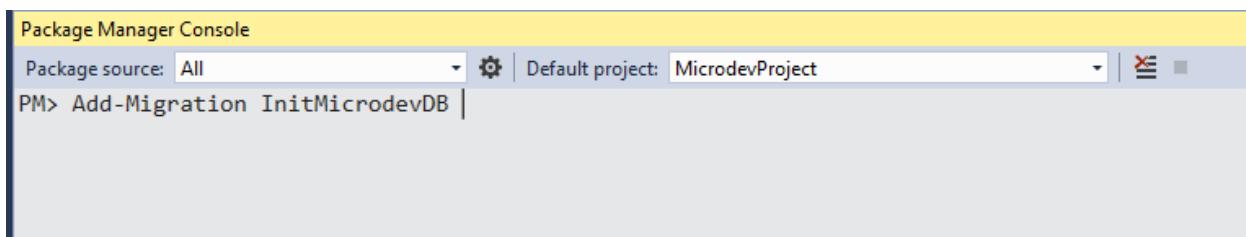
یک روش خوب برای ایجاد دیتابیس، وادار کردن **EF** به ساخت دیتابیس است. ساده‌ترین رویکرد **EF** برای انجام این کار، استفاده از **Migration** است.

راه حلی برای مدیریت جداول در دیتابیس می‌باشد. با **Migration** می‌توانید بدون هیچ‌گونه دردسری، تغییرات را به جداول دیتابیس اعمال نمایید.

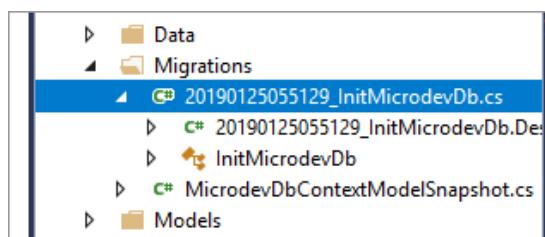
ایجاد Migration

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



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



```

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

```

یک هشدار !!

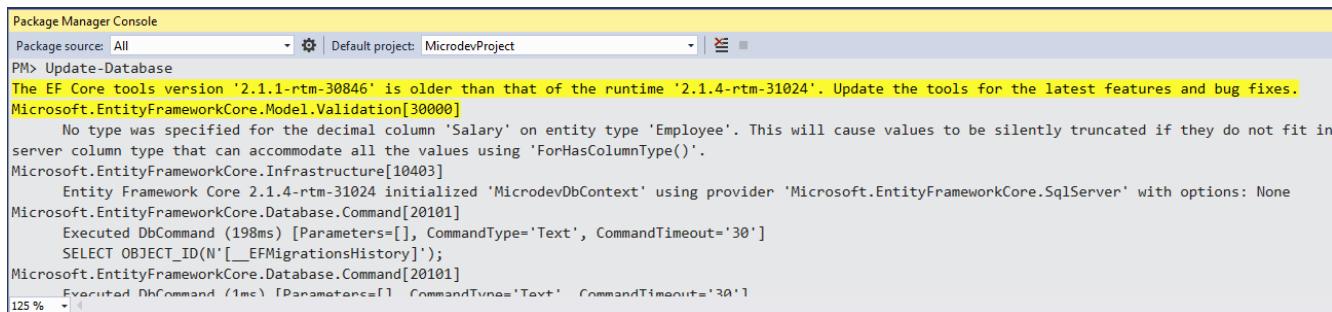
تغییر ساختار دیتابیس در وب سایت، باید به دقت مورد توجه قرار گیرد، بعضی موقعیت‌های از چیزها می‌توانند به اشتباه باشند و تاثیر آن می‌تواند باعث از دست رفتن اطلاعات یا خراب شدن وبسایت شود.

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

- ۱) اپلیکیشن شما می‌تواند در طول اجرا شدن **Startup**، دیتابیس را چک و **Migrate** کند.
- ۲) می‌توانید یک اپلیکیشن مستقل برای **Migrate** دیتابیس داشته باشید.
- ۳) می‌توانید از دستورات **SQL** برای دیتابیس استفاده کنید.

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

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

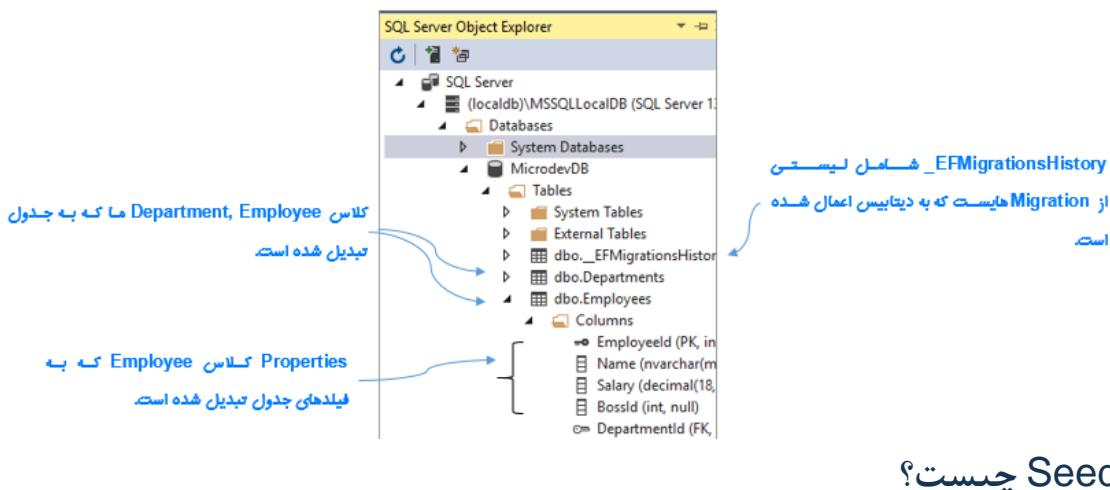


```

Package Manager Console
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 (1mc) [Parameters=[], CommandType='Text', CommandTimeout='30']
125 %

```

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



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

در 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 Data ← Folder یک کلاس به نام Seeder ایجاد نمایید.

- در این کلاس یک **Constructor** ایجاد کرده و **Inject** را به آن **MicrodevDbContext** نمایید و در پایان مقدار آن را درون یک متغیر **Readonly** قرار دهید.
- حالا یک متده **Seed()** تعریف کنید و با دستور **_context.Database.EnsureCreated();**

درون این متده، 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 می‌نمایید.

```
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="روشنده مدیریت"}, 
                    new Department{Name="باز کار"}, 
                    new Department{Name="پرداز ایده"}
                };
                _context.Departments.AddRange(departments);
            }

            if (!_context.Employees.Any())
            {
                List<Employee> employees = new List<Employee>
                {
```

اگر در جدول **Department** هیچ
شرکتی ثبت نشده باشد لیست
شرکت‌های بالا به این جدول اضافه
می‌شود.

مرحله دوم: يابد اين کلاس را در متده ConfigureServices، حسته نماییم.

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

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

در این مرحله ما یک Scope ایجاد می‌کنیم، سپس کلاس Seeder را به متدهای GetService و Seed داده و در پایان متدهای Seed را صدا می‌زنیم. عبارت Wait به این خاطر است که متدهای Seed در کلاس Seeder به صورت تعویض شده اند.

```
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 **View→SQL Server Object Explorer** شوید.

داده‌های جدول :Department

	DepartmentId	Name
▶	1	مدیریت روشمند
	2	دکان باز
	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 مژگانی	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	شهيلا 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

- ✓ 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.
- ✓ کلاس‌های EF Core را به جدول‌های متناظر و Property‌های Entity را به Column‌های EF Core و هر Instance از شی Entity به یک ردیف از جدول Map می‌کند.
- ✓ از Database Provider از EF Core استفاده می‌کند، که از طریق آن می‌توانید، بدون دستکاری Microsoft SQL Server، دارای EF Core Provider‌های SQLite، PostgreSQL، MySQL و ... است.
- ✓ یک ORM چند سکویی یا همان Cross Platform با Performance بالا است.
- ✓ بر اساس EF Core برای این Entity‌ها در DbSet<T>، یک نمای داخلی از Property‌های 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.
- ✓ دارای EF Core 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 رجیستر شدن DbContext است و شما همینجا می‌توانیدConnectionString provider را هم تعریف کنید.
- ✓ از EF Core برای رهگیری تغییرات تعریف Entity‌ها استفاده می‌کند تا مطمئن شود، تعریف Entity‌های شما، مدل‌های داخلی EF Core و اسکیماتی دیتابیس، همگی با هم یکی هستند.
- ✓ بعد از تغییر-یک Entity، شما می‌توانید یک Migration از طریق ابزار .NET CLI یا cmdlet PowerShell ویژوال 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 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 **Entity Instance** از یک **DbContext** و فرستادن آن به `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 سی‌شارپی 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

➤ چیست؟ Layout و Controller

➤ افزودن ViewStart ، TagHelper 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 **Cohesion** و **Loosely coupling** 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:

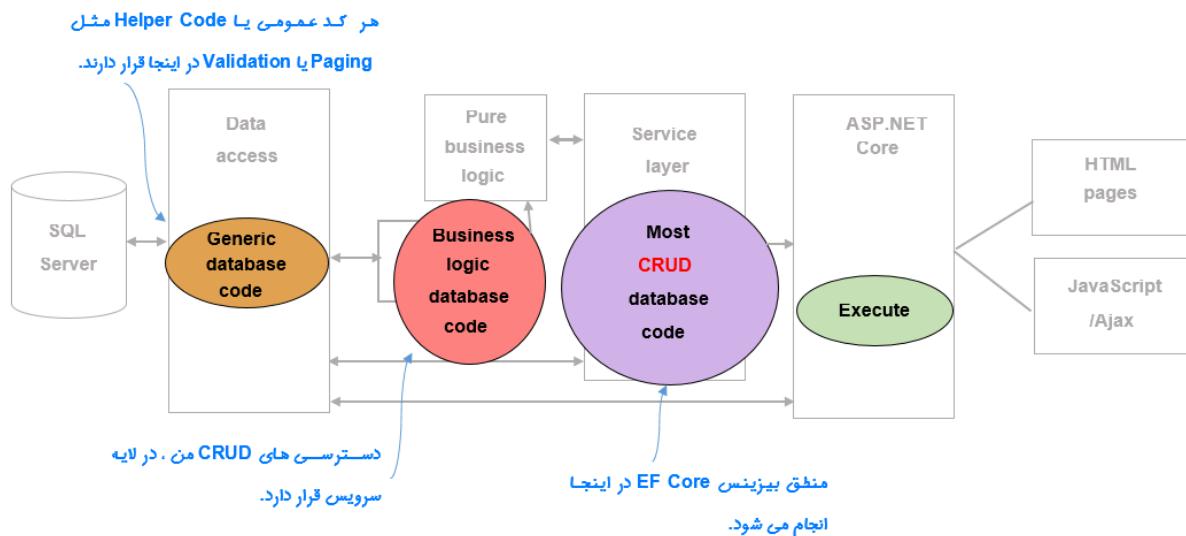
- **Frontend**، تماماً درباره نمایش اطلاعات است و برای انجام درست این وظیفه نیاز به مرکز زیادی دارد. بنابراین شما از لایه **Service** برای **Handle** کردن دستورات **EF Core** و انتقال اطلاعات(به شکلی که به راحتی قابل استفاده در **Frontend** باشد) استفاده می‌کنید (غلب از طریق **DTO**‌ها یا همان **ViewModel**‌ها). با این کار می‌توانید به جای فکر کردن به درستی **Query**‌های دیتابیس، بر روی ایجاد بهترین **User Experience** تمرکز کنید.

- **ASP.NET Controller**‌های معمولاً دارای چندین اکشن متدهستند(مثلاً یکی برای لیست اقلام، یکی برای اضافه کردن آیتم جدید، یکی برای ویرایش و...) که هر اکشن متدهست، نیاز به کدهای دسترسی به دیتابیس دارد. با انتقال کد دیتابیس به لایه **سروریس**، 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، می‌خواهم یک پروژه **CRUD** را شروع استارت بزنم.



نیازمندی‌های 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 سرویس‌های موردنظرمان را می‌نویسیم و سپس آن را به پروژه اضافه می‌کنیم.

روش استاندارد برای انجام عملیات بالا به شرح زیر است:

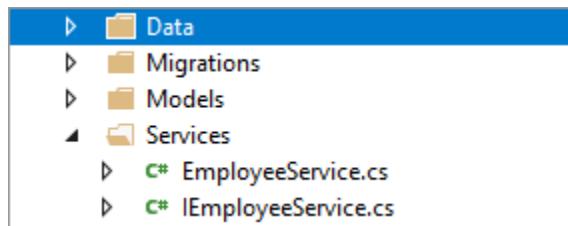
- ابتدا یک Interface تعریف نمایید و هر آنچه به آن نیاز دارید را در آن مشخص کنید.
- سپس این Interface را در یک کلاس پیاده‌سازی نمایید.

از لحاظ فنی این گونه برنامه‌نویسی، یک Best Practice است، زیرا می‌توان با استفاده از این اینترفیس Unit Test را به آسانی در پروژه پیاده‌سازی نمود.

نکته!!

Tenhe با این Interface در ارتباط است پس مطمئن شوید تمام متدها و Property‌های موردنیاز Controller را در آن قرار داده‌اید.

در پروژه‌ی خود، Folder 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:

- ✓ واکسی اطلاعات تمام کارمندان
- ✓ واکسی 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>> GetAllEmployeesAsync();
```

```

        Task<Employee> GetEmployeeAsync(int? id);
```

دریافت لیست تمام Employee

دریافت یک Employee براساس Id

```

        IQueryable<SelectListItem> GetDropDownEmployees();
        IQueryable<SelectListItem> GetDropDownDepartments();
```

دریافت لیست Employee و Department

```

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

```

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

افزودن یک Employee

Update یک Employee

```

    Task DeleteConfirmedAsync(int id); ← عملیات حذف یک Employee
    Task<bool> IsBoos(int id); ← برسی رئیس بودن یک فرد
}
}

```

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 DbContext

        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 => x.Department).ToListAsync(); ← برگرداندن تمام Employee
        }

        public async Task<Employee> GetEmployeeAsync(int? id)
        {
            return await _context.Employees.Include(x=>x.Department).Include(x=>x.Boss).FirstOrDefaultAsync(m => m.EmployeeId == id); ← جوین زدن با جدول Department و Employee
        }

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

```

```

public IQueryable<SelectListItem> GetDropDownDepartments()
{
    return _context.Departments.Select(x => new SelectListItem { Value =
x.DepartmentId.ToString(), Text = x.Name });
} برمگرداندن لیستی از نام شرکت‌ها

public async Task<Employee> AddAsync(Employee employee)
{
    _context.Employees.Add(employee); افزودن Employee دریافتی به
    await _context.SaveChangesAsync(); جدول Employees
    return employee; ذخیره تغییرات در دیتابیس
}

public async Task<Employee> UpdateAsync(Employee employee)
{
    _context.Update(employee); Update
    await _context.SaveChangesAsync(); دریافت Employee
    return employee; ذخیره تغییرات در دیتابیس
}

public async Task DeleteConfirmedAsync(int id) يافتن Employee که با
{
    var employee = await _context.Employees.FindAsync(id); id ورودی مبارابر باشد

    _context.Employees.Remove(employee); حذف Employee از
    await _context.SaveChangesAsync(); جدول Employees
    ذخیره تغییرات در دیتابیس
}

public async Task<bool> IsBoos(int id) بررسی هی کند این شخص
{
    var isBoos = await _context.Employees.AnyAsync(x=>x.BossId==id);
    return isBoos;
}
}

}

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

```

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

```

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

```

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

رجیستر گردن سرویس

EmployeeService

Interface

من در بالا سرویس **EmployeeService** را به صورت **Scoped** رجیستر نمودم، زیرا **DbContext** در واقع به صورت **Thread-safe** نیست؛ دلایل جالت ملائم 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، دیگر در گیر 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 سرویس‌های **TestService** را استفاده می‌کند.

Controller چیست؟

ASP.NET.Core در کلاسی است که می‌تواند، تعدادی اکشن‌متداشته باشد و این اکشن‌متدها می‌توانند به صورت منطقی با هم در ارتباط باشند.

هنگامی که یک درخواست به Controller فرستاده می‌شود، یک **MVC Middleware Instance** جدید از ساخته شده و اکشن‌متد آن صدای می‌شود.

قراردادهای استفاده از Controller‌ها به شرح زیر می‌باشد:

- **Controller** خاتمه **Controller** باشد.
- از کلاس **Controller** یا **ControllerBase** یا **HandleRequest** ارث بری کنند. (یا هر کلاسی که از این دو کلاس ارث بری کرده است).

HandleRequest هر کلاسی را که مطابق این شرایط باشد، در زمان اجرا شناسایی می کند و در زمان **MvcMiddleware** شدن **Request**ها، آن را در دسترس قرار می دهد.

توجه داشته باشید ارث بری از کلاس **Controller** اجباری نیست، ولی بهتر است که تمام **Controller**ها از این کلاس ارث بری کنند، زیرا این کلاس، متدهای کمکی زیادی در اختیار ما قرار می دهد.

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

چیست **Layout**

هر **HTML Document**، دارای المنشاهی تکراری **<HTML>**, **<Head>**, **<Body>** و در برخی صفحات **Footer**، **Header** می باشد. و احتمالا هر صفحه از برنامه شما به فایل های **Bootstrap**, **JQuery** 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 می توانید 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 ساده تر می شود و شاید بخواهید، برای هر بخش، قالب متفاوتی درنظر بگیرید، در این صورت می توانید **Layout** های متفاوتی داشته باشید. به صورت قراردادی **Layout** ها در یک **Folder**ی به نام **Layout** قرار می گیرند و نام **Layout** با کاراکتر **_** شروع می شود.

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.

یک **Folder**ی به نام **Shared** در **View** **Folder** در آن قرار دهید. حالا کدهای زیر را در آن کمی 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" integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO" crossorigin="anonymous">

</head>
<body>
    <script
        src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6ZbWh2IMqE241rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>

    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">مشترک صفحات navbar
            <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() ← می‌گوید که محتوای Template Engine پ
    </main> ← Child View ها باید کجا قرار گیرد.
</div>

<footer class="border-top footer text-muted">
    <div class="container">
        &copy; 2019 - MicrodevProject - Zahra Bayat ← مشترک صفحات Footer
    </div>
</footer>

```

</body>
</html>

بعد از اضافه کردن این **EmployeeController** در **Layout** ایجاد نمایید.

```

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc;

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

```

اکنون نیاز است بعد از رجیستر شدن **IEmployeeService** به عنوان سرویسی که می‌تواند از طریق **Inject** ← DI خود **Controller** را به آن شود، این سرویس را به **Controller** 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; ← سپس مقدار آن درون یک متغیر EmployeeController
                                                ← تزریق و readOnly ریخته شده است.
    }
}

```

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

نمایش لیست کارمندان

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

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

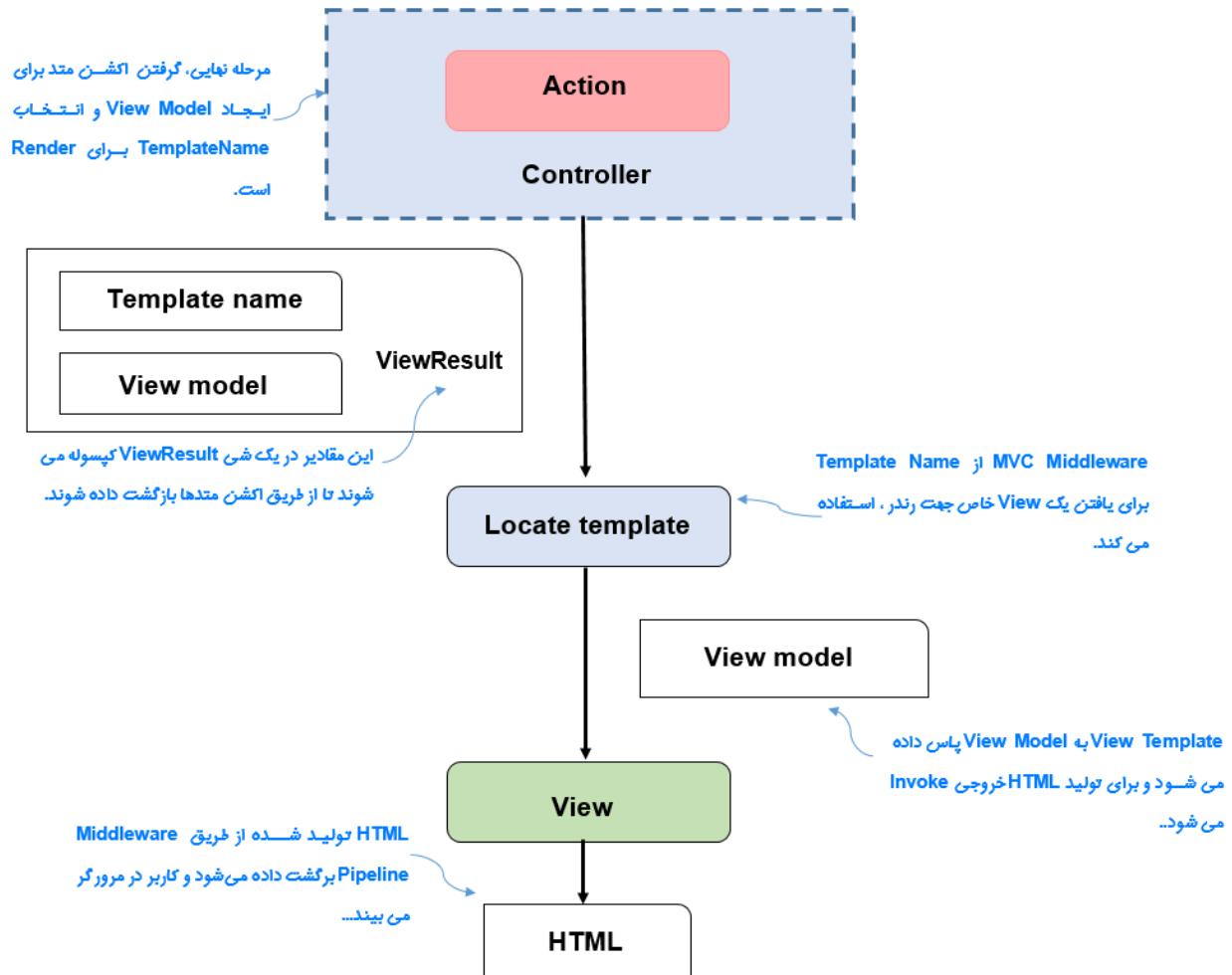
namespace MicrodevProject.Controllers
{
    public class EmployeeController: Controller
    {
        private readonly IEmployeeService service;

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

        public async Task<IActionResult> Index()
        {
            var employees = await _service.GetAllEmployeesAsync();
            return View(employees);
        }
    }
}
```

دریافت لیست کارمندان از سرویس

پاس دادن نتیجه به متده View



نکته!!

شما می‌توانید خروجی متدهای را هر آنچه که نیاز دارید قرار دهید، اما در MVC مرسوم است که خروجی اکشن‌متدها 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 C# 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"}  
}
```

در اینجا لیستی از نام اشخاص را درون

```
<h1>Names: </h1>
```

متغیری قرار دهیم.

```
<ul>
```

```
    @for (int i = 0; i < names.Count; i++)  
    {
```

ترکیب C#, HTML این امکان را می دهد

```
        var name = names[i];  
        <li>@i - @name</li>
```

که به صورت داینامیک در زمان Runtime

```
    }
```

بتوانیم HTML تولید کنیم.

```
</ul>
```

در مثال بالا Razor مستقیما، HTML را همانطور که هست به خروجی کپی می کند. کدهای HTML در مثال بالا بین <> قرار دارند و تعدادی عبارات سی شارپ می بینید که با کدهای HTML، نتیجه موردنظر شما را تولید می کند.

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

(۱) آن هایی که تنها، داده های نمایش داده شده در برنامه شما را می خوانند.

(۲) آن هایی که داده هایی را به برنامه شما ارسال می کنند.

زبان Razor حاوی تعدادی سازنده است که ساخت هر دو نوع اپلیکیشن را ساده تر می کند. با استفاده از Razor، می توانید HTML را با مقادیری از ViewModel ترکیب کنید و داده ها 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.

چیز هایی که با HTML نمی توان به آن رسید، با Razor به راحتی می تواند به واقعیت بپیوندد.

Tag Helper چیست؟

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

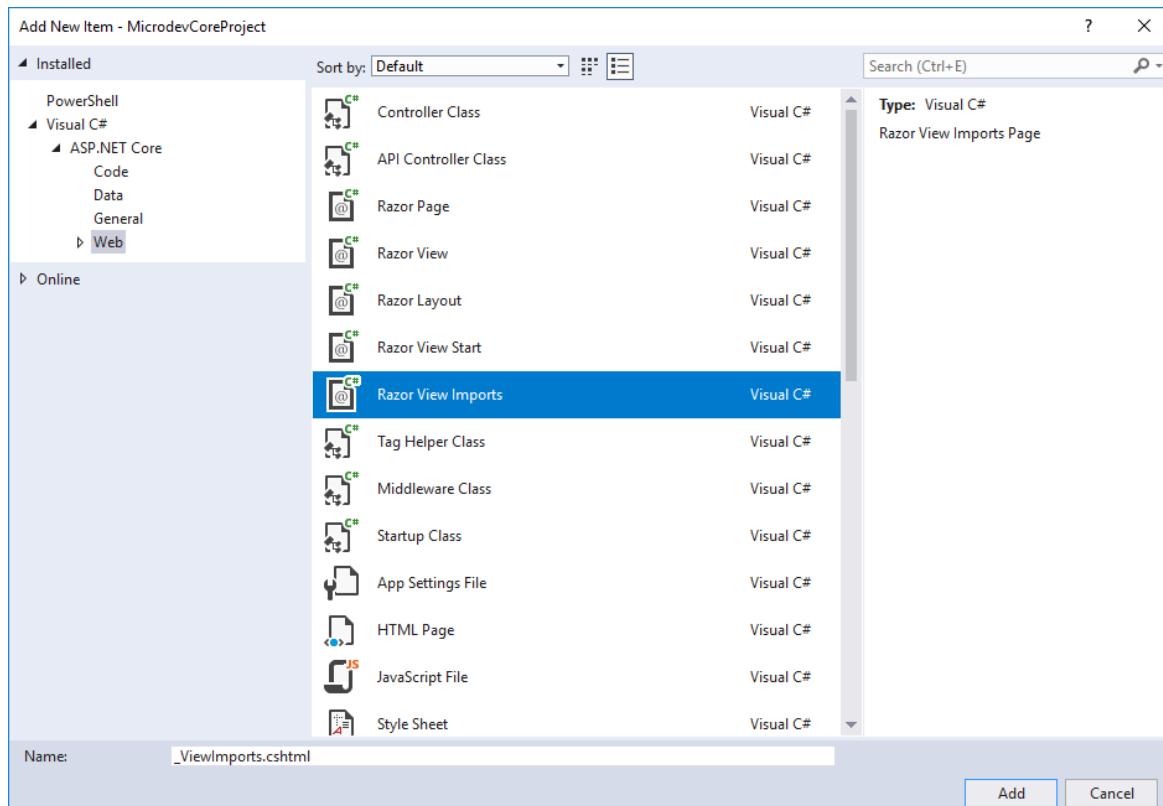
فرم ها یک روش کلیدی، برای برقراری ارتباط کاربران با اپلیکیشن، در مرورگر است. پس مهم است که فرم ها هم برای اپلیکیشن شما، درست و کاربرپسند تعریف شوند.

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

Tag Helper ها می توانند با هدف سفارشی کردن ویژگی های HTML، به یک المنش (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 را انتخاب نموده و Add → New Item را راست کلیک کنید و Views ← Folder روی Tag Helper سپس Razor View Import را برگزینید.

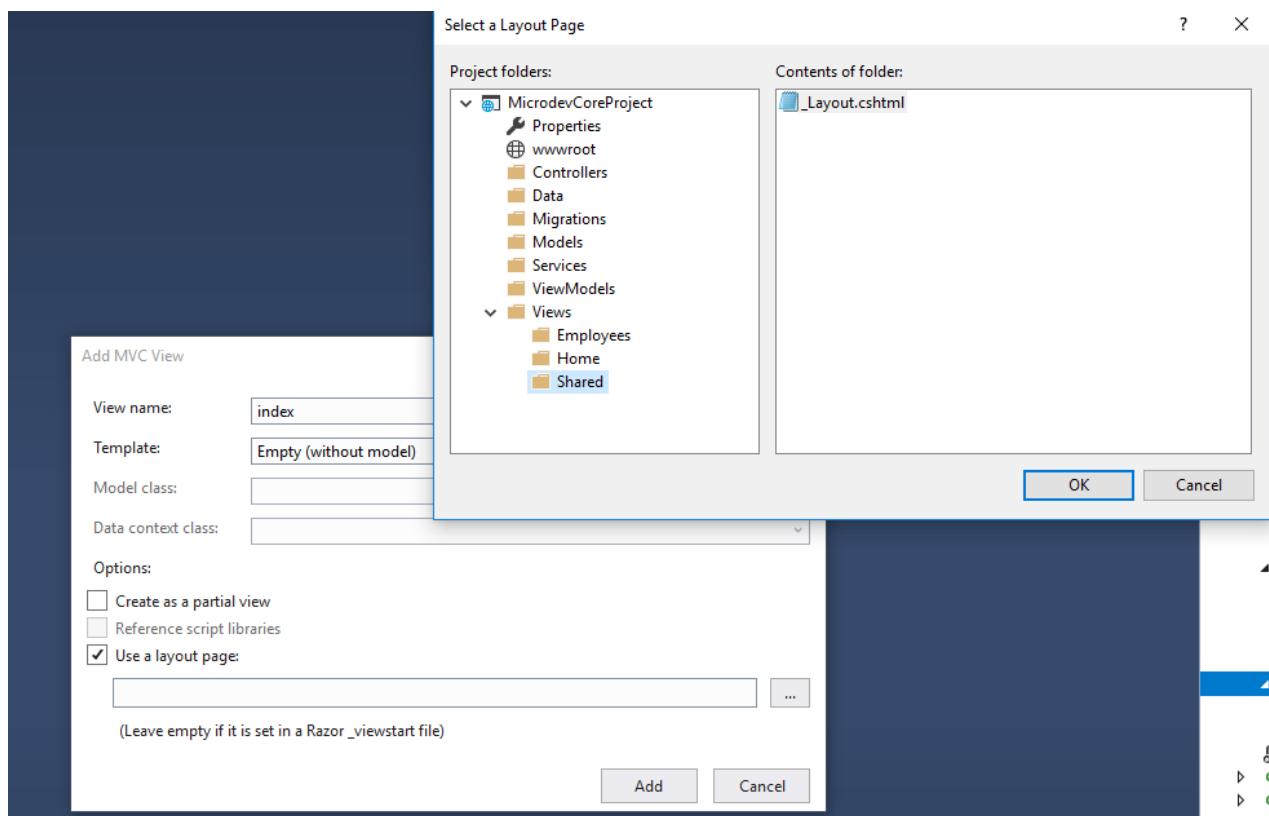
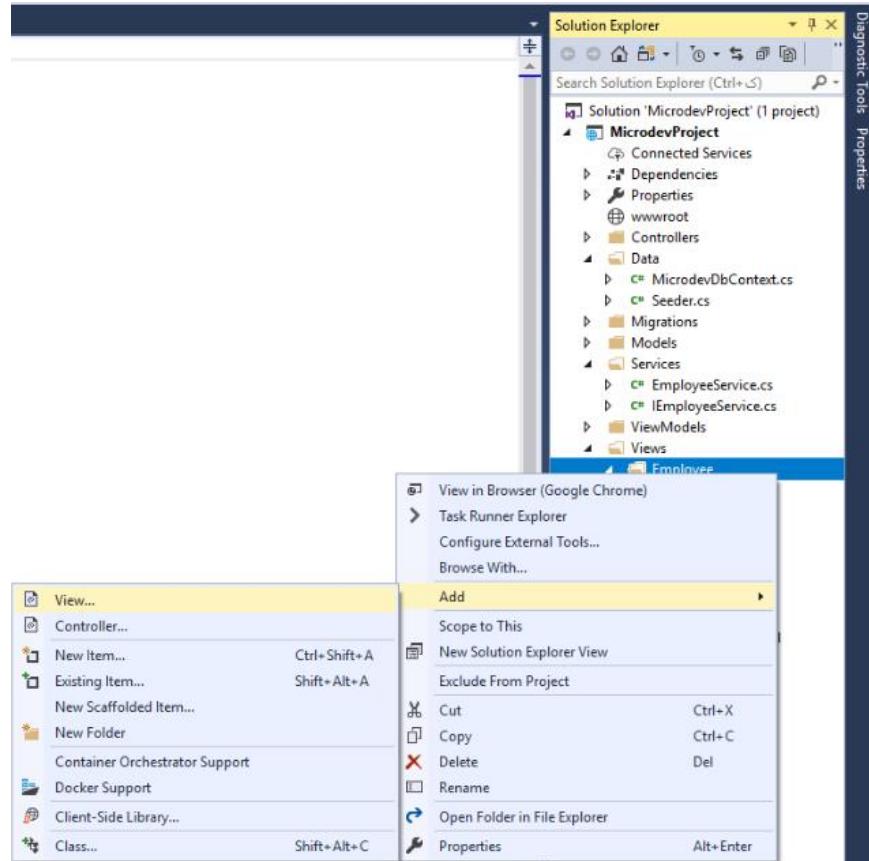


حالا، کد Tag Helper زیر را به این فایل اضافه نمایید.

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

خوشحالم که با هم توانستیم Tag Helper را به این پروژه اضافه نماییم بیایید به پروژه CRUD برگردیم...

ایتم Folder به نام Views ایجاد کنید و درون آن، Folder دیگری به نام Employee و درون این آیتم index.cshtml را اضافه نمایید.



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

```

@model IEnumerable<MicrodevProject.ViewModels.Employee>;           استفاده از @model جهت
@{                                         توصیف برای Model Type
    ViewData["Title"] = "کارمندان";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div dir="rtl">
    <h1 class="d-flex justify-content-center">کارمندان لیست</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>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) ساختار سی شارپ می تواند
        {                                                 با Razor همراه باشد.
            <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>

```

برای این View در نظر گرفته می شود.

نمایش عناوین Property ها در کلاس Employee

می توانید مقادیر را در المنت های سی شارپ نمایش دهید.

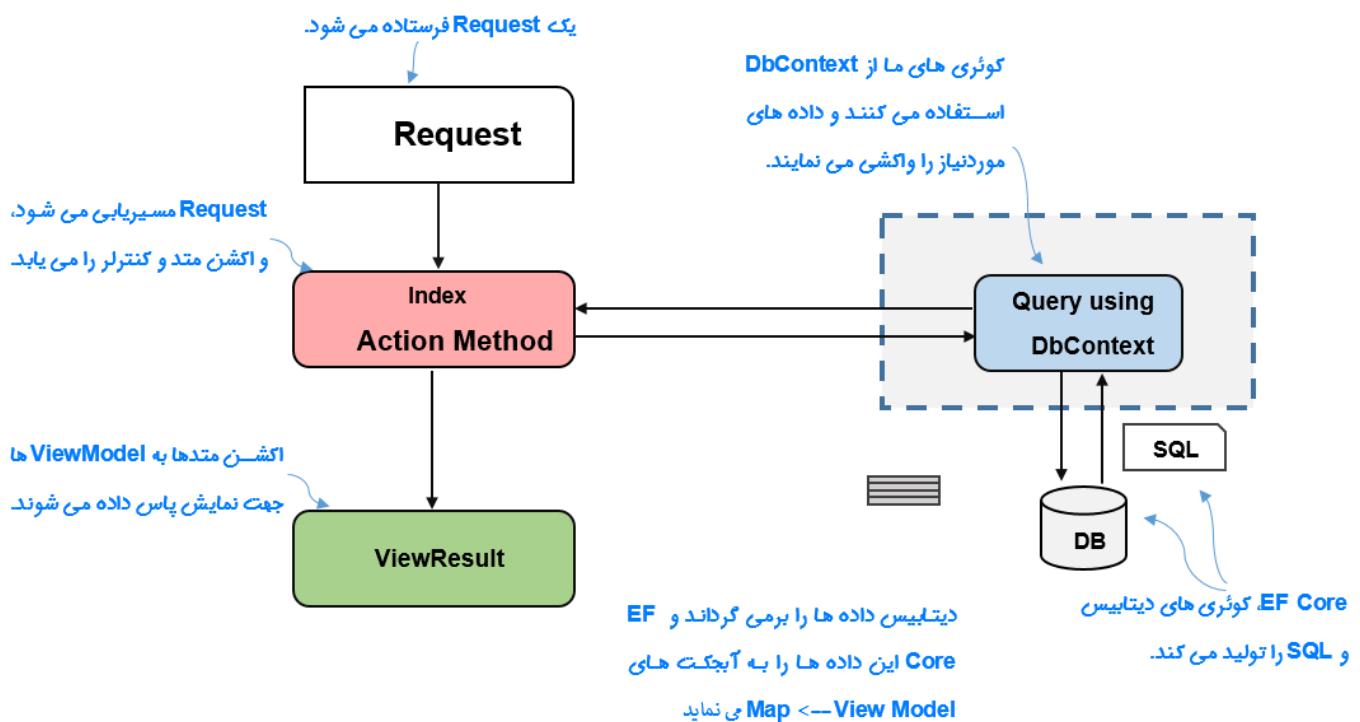
```

<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.



MicrodevProject دیارتمان کارمند

لیست کارمندان

[افزودن کارمند](#)

نام و نام خانوادگی	حقوق دریافتی	شرکت	جزئیات	ویرایش	حذف
زهرا بیات	1000.00	مدیریت روشنمند			
علی بیات	3000.00	دکان باز			
امین مژگانی	1000.00	مدیریت روشنمند			
محمد سهیلی	4000.00	مدیریت روشنمند			
سara احمدی	1000.00	دکان باز			
محمد سعیدی	1000.00	دکان باز			
سهیلا افتخاری	2000.00	دکان باز			
سعید حسینی	5000.00	ایده بردار			
زهرا محمدی	1000.00	ایده بردار			

© 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.

خوبی بختانه **.NET Core** دو مکانیزم برای حل این مسائل بیان کرده است:

▪ **_ViewImports.cshtml**
▪ **_ViewStart.cshtml**

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

این فایل را می‌توان در هر Folderی قرار داد (معمولاً در Viewها 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 نخواهد بود.

_ViewImports.cshtml :

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

پیش‌فرض اپلیکیشن Namespace
نوشتن این در فایل، یعنی در هر View، این Namespace در دسترس است.
برای تمام View‌ها اضافه شدن TagHelper

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 View‌ها، ابتدا محتويات View‌یی با نام _ViewStart.cshtml را اجرا خواهد کرد، پس هر کدی که در این فایل باشد، قبل از اجرای تمام View‌ها اجرا خواهد شد.

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

نکته!!

اگر فلیل ViewStart.cshtml را درون Home Layout قرار دهیم؛ تنها برای Home View‌ها در نظر گرفته خواهد شد. اما با قرار دادن این فلیل درون Views Layout، برای تمامی View‌ها تنظیم خواهد شد.

به پروژه برگردیم....

جزئیات کارمند

برای نمایش جزئیات کارمند، ابتدا یک View به نام Details 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> ریس نام </th>
                <td> @Html.DisplayFor(model => model.Boss.Name) </td>
            </tr>
            <tr>
                <th> @Html.DisplayNameFor(model => model.Salary) </th>
                <td> @Html.DisplayFor(model => model.Salary) </td>
            </tr>
            <tr>
                <th> <a class="btn btn-info" asp-action="Index">اصلی صفحه</a> </th>
                <td><a class="btn btn-success" asp-action="Edit" asp-route-id="@Model.EmployeeId">ویرایش</a> </td>
            </tr>
        </table>
    </div>

```

نمایش جزئیات کارمند

→ |

TagHelper ها برای مسیریابی بین صفحات

→ |

برای ادامه کار باید با نحوه ایجاد URL برای اکشن متده آشنا شوید.

ایجاد URL برای اکشن

گاهی اوقات پیش می آید که بخواهید یک URL را به عنوان بخشی از نتیجه ای اکشن متده برگردانید. این اتفاق، معمولاً زمانی می افتد که بخواهید یک مرورگر را به یک اکشن متده 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 متده یک RedirectActionResult با یک URL باشد که تولید می کند. نام Action Method را در RedirectResult می کند.

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 متده که `RedirectToRoute` با یک URL تولید می‌کند. `RedirectToRouteResult` هم معادل است. `Url.RouteUrl`

دانستن این مطالب به ادامه بحث ما کمک بسیاری می‌کند....

حالا نوبت به نوشتن متده `Details` می‌رسد. در `EmployeeController` متده 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();  
    }  
    return View(employee);  
}
```

گرفتن Id کارمند از ورودی گردید. این Id=null نبود. این آنرا به متده `GetEmployeeAsync` سرویس مانده تا کارمند موردنظر را از دیتابیس واکشید.

اگر id null باشد، صفحه.NotFound را به متده `GetEmployeeAsync` سرویس مانده تا کارمند موردنظر را از دیتابیس واکشید.

اگر کارمندی با این id وجود نداشت، صفحه.NotFound نمایش داده می‌شود.

وگرنه کارمند واکشید شده را به View موردنظر می‌فرستد.

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

The screenshot shows a web page titled "جزئیات کارمند". The page displays the following information:

زمینه	اطلاعات
نام و نام خانوادگی	زهرا بیات
نام شرکت	مدیریت روشمند
نام رئیس	امین مژگانی
حقوق دریافتی	1000.00

At the bottom of the page, there are two buttons: "ویرایش" (Edit) and "صفحه اصلی" (Main Page).

Page footer: © 2019 - MicrodevProject - Zahra Bayat

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

نوبت به نوشتتن متدهای ویرایش می‌رسد.

دوباره به **EmployeeController** بر می‌گردیم و این بار متدهای **Edit** را مانند کد زیر می‌نویسیم.

باید دو متدهای **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 متدهای صدای زده می‌شود).

```
متدهای Get: گرفتن Id کارمند از ورودی
public async Task<IActionResult> Edit(int? id)
{
    اگر Id=null نبود، این Id را به متدهای GetEmployeeAsync بفرستد.
    if (id == null)           GetEmployeeAsync
        اگر این Id=null باشد،
        return NotFound();    سرویس ما می‌دهد تا کارمند موردنظر را از دیتابیس واکنش کند.
    صفحه NotFound نمایش
    Employee employee = await _service.GetEmployeeAsync(id);   دریافت لیست Employee و Department از سرویس و قرار
                                                                در آن نتیجه در دو متغیر
                                                                ViewBag.Bosses, ViewBag.Department
                                                                در آن نتیجه در دو متغیر
                                                                ViewBag.Bosses = _service.GetDropDownEmployees(); ←
                                                                در آن نتیجه در دو متغیر
                                                                ViewBag.Departments = _service.GetDropDownDepartments(); ←
                                                                در آن نتیجه در دو متغیر
                                                                کارمند واکنش شده را به View موردنظر می‌فرستد.
                                                                return View(employee);
}
کارمند واکنش شده را به View موردنظر می‌فرستد.
```

بعد از نوشتتن متدهای بالا در **EmployeeController**، باید در **View** 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">          ها برای Form TagHelper
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>          ها استفاده می‌شود.
            <input type="hidden" asp-for="EmployeeId" />          Action URL
        <div class="form-group">          برای تولید input و Value در asp-for
            <label for="EmployeeName">نام کارمند:</label>          برای Model Validation Attribute
            <input asp-for="EmployeeName" />
        </div>
    </div>
</div>
```

```

<label asp-for="Name" class="control-label d-flex p-2" dir="rtl"></label>
    View Model Label برای تولید برچسب هایی بر مبنای asp-for دار.
    <input asp-for="Name" class="form-control" />
    <span asp-validation-for="Name" class="text-danger"></span>
</div>
    من باشد. Validation Message ها با استفاده از Validation Message
    نوشته می شوند. TagHelper ها در یک Span می شوند.

<div class="form-group">
    <label asp-for="Salary" class="control-label d-flex p-2"></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 d-flex p-2">رئیس نام</label>
    نمایش کارمندان در یک DropDown <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 d-flex p-2">شرکت نام</label>
    نمایش شرکت ها در یک DropDown <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 اطلاعات" class="btn btn-success" />
    Button ذخیره اطلاعات
</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 خواهد شد، که می توانید مشخصات کارمند را تغییر دهید.

توجه داشته باشید: هنوز Button ذخیره، هیچ اطلاعاتی را بروزرسانی نمی کند.

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

MicrodevProject دیارتمان کارمند

ویرایش کارمند

نام و نام خانوادگی

زهرابیات

حقوق دریافتی

1000.00

نام رئیس

زهرا محمدی

نام شرکت

مدیریت روشنمند

ذخیره

صفحه اصلی

© 2019 - MicrodevProject - Zahra Bayat

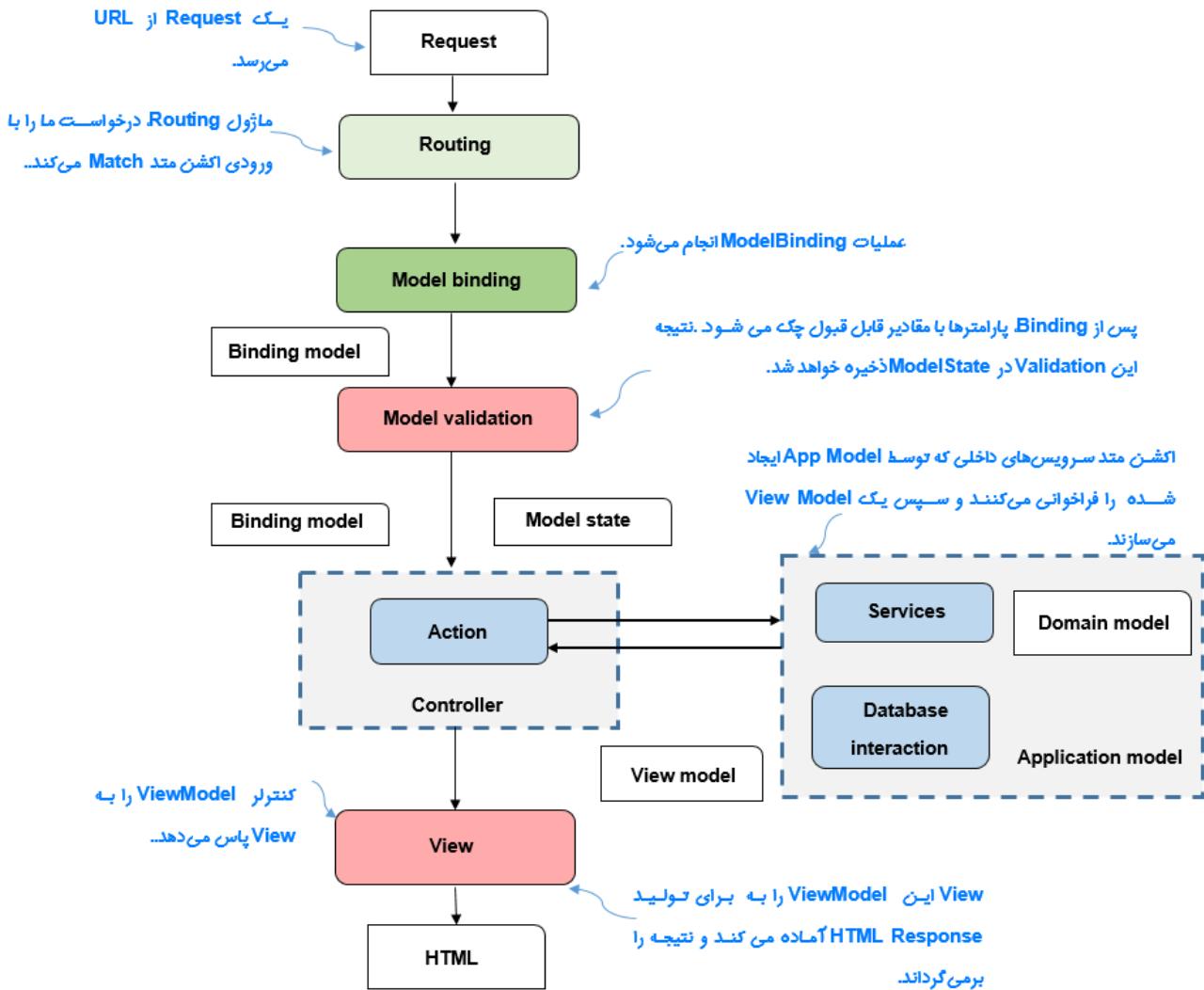
Model Binding چیست؟

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

اطلاعاتی که در هنگام ایجاد یک **Request** توسط کاربر، به صراحت درخواست می‌شود، از طریق **Model Binding** به آبجکت‌های شما در اکشن‌متد بایند می‌شود.

این همان چگونگی پذیرش داده‌های ارسال شده، توسط کاربر و تبدیل آن به یک شی سی‌شارپ است. شما می‌توانید از **Model Binding** در اکشن‌متدها استفاده نمایید.

قبل از اکشن‌متد اجرا می‌شود و بررسی می‌کند: آیا مقادیر اتصالی معتبر است یا خیر؟



Server بر روی Validation

اعتبارسنجی **Model Binding**، قبل از اجرای اکشن مت رخ می‌دهد اما توجه داشته باشید، که اکشن مت‌ها در هر صورت (چه **Validate** موفق باشد چه نباشد) اجرا خواهند شد.

اعتبارسنجی به صورت خودکار اتفاق می‌افتد اما مدیریت داده‌های معتبر بر عهده‌ی اکشن مت است.

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

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

ما یک متدهای `Edit` دیگر با فعال `[HttpPost]` نیازمندیم. به `EmployeeController` برمی‌گردیم و کدهای ذخیره اطلاعات را در متدهای `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));
    }
}
```

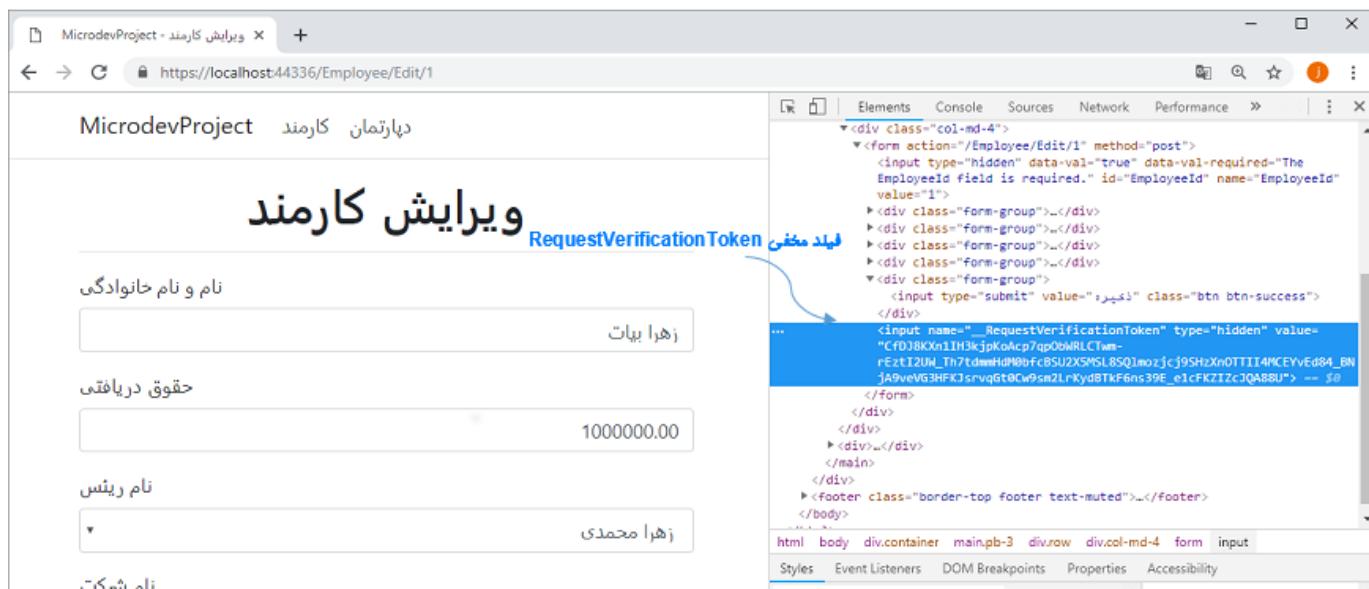
اگر `Id` دریافتی با `EmployeeId` مغایر باشد، متدهای `NotFound` برای نباید صفحه `NotFound` نمایش داده می‌شود.

اگر مدل ارسالی معتبر باشد، متدهای `UpdateAsync` سرویس صدازده می‌شود و `Employee` مارک در `View` به `employee` بروزرسانی شود و `Update` می‌شود و پاس داده خواهد شد.

در غیر این صورت به صفحه `Redirect` می‌شود.

ValidateAntiForgeryToken چیست؟

درین 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. در حقیقت 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.

متدهای حذف کارمند

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

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
        return NotFound();
    Employee employee = await _service.GetAllEmployeesAsync(id);
    if (employee == null)
        return NotFound();
    var isBoos = await _service.IsBoos(employee.EmployeeId);
    if (isBoos)
    {
        return View("DeleteFailed", employee);
    }
    return View(employee);
```

وگرنه `Id` دریافتی به متدهای `GetAllEmployeesAsync` گلاس سرویس پاس داده می‌شود و یک `Employee` بازگشت داده خواهد شد.

صفحه نمایش داده خواهد شد.

اگر `Id` null بود، صفحه `Employee` نمایش داده شود.

بررسی می‌گند که این شخص رئیس است؟ اگر رئیس باشد، وارد صفحه‌ی حذف نشود.

وگرنه `View` به `Employee` پاس داده می‌شود.

بعد از نوشتن متدهای `Delete` بالا در `EmployeeController`، باید در `View` به نام `Employee` یک `View` ایجاد نمایید و کدهای نمایش جزئیات کارمند را در آن قرار دهید.

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

```
@{
    ViewData["Title"] = "حذف کارمند";
```

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

```
<h3 class="d-flex justify-content-center">آیا از حذف این کارمند مطمئن هستید؟</h3>
<div class="d-flex justify-content-center" dir="rtl">
```

```
<br />
<form asp-action="Delete">
```

```

<input type="hidden" asp-for="EmployeeId" />
<table>
    <tr>
        <th><h3 class="control-label" dir="rtl"></h3></th>
        <th>@Model.EmployeeId</th>
    </tr>
    <tr>
        <th><h3 class="control-label" dir="rtl"></h3></th>
        <th>@Model.Name</th>
    </tr>
</table>
<br />
<input type="submit" value="حذف کارمند" class="btn btn-danger" /> |
<input type="submit" value="صفحه اصلی" class="btn btn-info" />

</form>
</div>

```

اگر کارمند فعلی رئیس باشد باید یک View دیگر هم با نام DeleteFail در مسیر `DeleteFail` در مسیر `Employee` ایجاد کنید.

```

@model MicrodevProject.Models.Employee

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

<h1 class="d-flex justify-content-center">حذف کارمند</h1>
<h3 dir="rtl" class="d-flex justify-content-center">این کارمند دارای سمت رئیس می باشد.</h3>
<h3 dir="rtl" class="d-flex justify-content-center">.این کارمند قابل حذف نیست</h3>
<table class="d-flex justify-content-center" dir="rtl">
    <tr>
        <th><h3 class="control-label" dir="rtl"></h3></th> : شماره کارمندی</th>
        <th>@Model.EmployeeId</th>
    </tr>
    <tr>
        <th><h3 class="control-label" dir="rtl"></h3></th> : نام و نام خانوادگی</th>
        <th>@Model.Name</th>
    </tr>
</table>
<br />
<a asp-action="Index" class="btn btn-info">صفحه اصلی</a>

```

حالا اپلیکیشن را اجرا و بر روی دکمه حذف یک از کارمندان که رئیس نیست کلیک کنید.

The screenshot shows a web page titled "حذف کارمند" (Delete Employee) with the sub-question "آیا از حذف این کارمند مطمئن هستید؟" (Are you sure you want to delete this employee?). It displays employee number 9, name Zahra Bayat, and her manager's name Zahra Mohammadi. Below the form are two buttons: "صفحه اصلی" (Main Page) in blue and "حذف کارمند" (Delete Employee) in red. The footer contains the copyright notice "© 2019 - MicrodevProject - Zahra Bayat".

بعد از حذف ، اینبار بر روی کارمندی که خود رئیس است کلیک کنید.

The screenshot shows a web page titled "حذف کارمند" (Delete Employee) with the sub-question "این کارمند دارای سمت رئیس می باشد." (This employee has the position of head). It displays employee number 1, name Zahra Bayat, and her manager's name Zahra Bayat. Below the form is a single "صفحه اصلی" (Main Page) button in blue. The footer contains the copyright notice "© 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 متده زیر را اضافه می نماییم.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
```

پاس دادن id ورودی به متده

DeleteConfirmedAsync

کلاس سرویس و حذف کارمند از

دیتابیس.

```

    await _service.DeleteConfirmedAsync(id);
    return RedirectToAction(nameof(Index));
}

```

بازگشت به صفحه

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

درج کارمند جدید

آخرین مطلب پروژه CRUD درج کارمند جدید می‌باشد.

به Employee Controller برمی‌گردیم و این‌بار متدهای Create را ایجاد می‌کنیم.

ما باید دو متدهای Create و Get را ایجاد کنیم. یکی برای فعل Get و دیگری برای فعل Post.

فعل Get برای متدهای Create

```

[HttpGet]
public IActionResult Create()
{
    ViewBag.Bosses = _service.GetDropDownEmployees();
    ViewBag.Departments = _service.GetDropDownDepartments();

    return View();
}

```

این متدهای GetDropDownDepartments و GetDropDownEmployees از کارمندان و شرکت‌ها را از متدهای Department و Bosses می‌آوردند و مقادیر آنها را در متغيرهای ViewBag ذخیره می‌کنند. قرار ViewBag از نوع Department و Bosses می‌باشد.

این مقادیر باید در View‌های DropDown نمایش داده شود.

یک View به نام Create در مسیر Employee → View 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" />
        </form>
    </div>
</div>

```

برای تولید URL Action ها در Form ها استفاده می شود. TagHelper

asp-for برای تولید input در Validation و Value است. Model Attributes برای Model

```
<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" Model برای تولید برچسب هایی بر مبنای
    <label asp-for="Salary" class="control-label"></label> می باشد.
    <input asp-for="Salary" class="form-control" />
    <span asp-validation-for="Salary" class="text-danger"></span>
</div>
<div class="form-group" نمایش کارمندان در
    <label class="control-label">رینس نام</label>
    DropDown یک <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" نمایش شرکت ها در
    DropDown یک <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"
    <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.

MicrodevProject دفترچه کارمند

افزودن کارمند

نام و نام خانوادگی
سازا بیات

حقوق دریافتی
1000000

نام ریئس
زهرا بیات

نام شرکت
مدیریت روشنمند

ذخیره

صفحه اصلی

© 2019 - MicrodevProject - Zahra Bayat

نوبت به نوشتن فعل POST برای متدهای 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: Employee
        public async Task<IActionResult> Index()
        {
            IEnumerable<Employee> employees = await
                _service.GetAllEmployeesAsync();
            return View(employees);
        }

        // GET: Employee/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: Employee/Create
        [HttpGet]
        public IActionResult Create()
        {
            ViewBag.Bosses = _service.GetDropDownEmployees();
            ViewBag.Departments = _service.GetDropDownDepartments();

            return View();
        }

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

    Employee employee = await _service.GetAllEmployeesAsync(id);
    if (employee == null)
    {
        return NotFound();
    }
    var isBoos = await _service.IsBoos(employee.EmployeeId);
    if (isBoos)
    {
        return View("DeleteFaild", employee);
    }

    return View(employee);
}

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

}
```

فصل ششم : ASP.NET Identity

Authorization و Authentication چیست؟ ➤

Principal چیست؟ ➤

User مدیریت ➤

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 سایتها مختلف به گوش می‌رسد. شاید شنیدن این جملات کمی نامید 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 داینامیک 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 لاین شده امکانات متفاوتی ارائه دهید.

بیایید کمی بیشتر به این موضوع بپردازیم و این قابلیت فوق العاده را به این اپلیکیشن اضافه نماییم. یک **ASP.NET Core Identity** است که قابلیت ورود به **ASP.NET Core Membership system** را اضافه می‌کند. می‌تواند با استفاده از دیتابیس **SQL Server** پیکربندی شود تا بدین وسیله نام کاربر، کلمه عبور و اطلاعات پروفایل را ذخیره نماید.

بیایید کمی بیشتر به این موضوع بپردازیم و این قابلیت فوق العاده را به این اپلیکیشن اضافه نماییم.

Authorization و Authentication چیست؟

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

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

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

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 اtribut **[Authorize]** به بالای اکشن‌های **Controller** صورت می‌پذیرد.

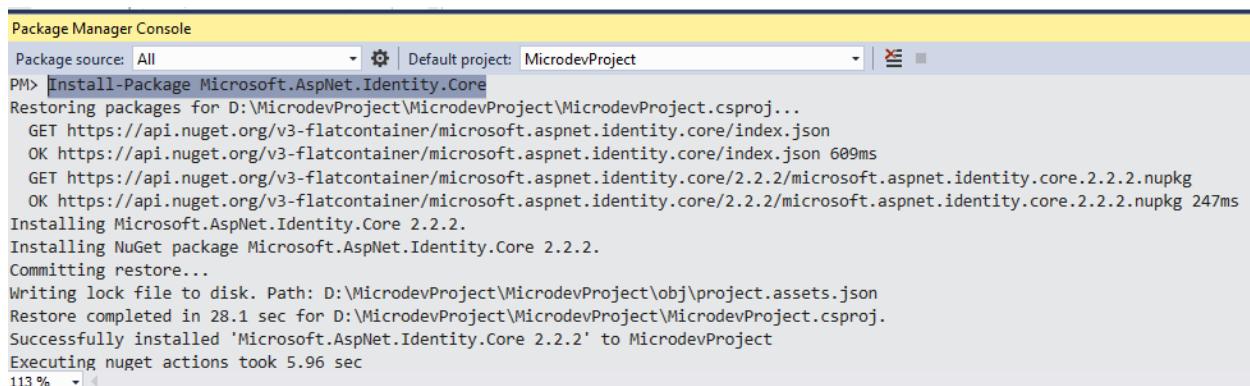
نکته!!

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

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

ابتدا در **Model ← Folder** یک کلاس به نام **User** ایجاد و در این کلاس از **IdentityUser** ارث بری نمایید.

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

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

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

برای سفارشی سازی **User** می‌توانید **Property**‌هاین را در اینجا اضافه نمایید.

```
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 EF Core از Identity باید به جای ارث بری ارث بری نمایید. DbContext از DbContext و IdentityDbContext از DbContext است.

برای لینکه EF Core بتواند از Identity پشتیبانی کند باید به جای DbContext از کلاس TUser ارث بری کند. کلاسی است که باید از IdentityUser ارث بری کرده باشد.

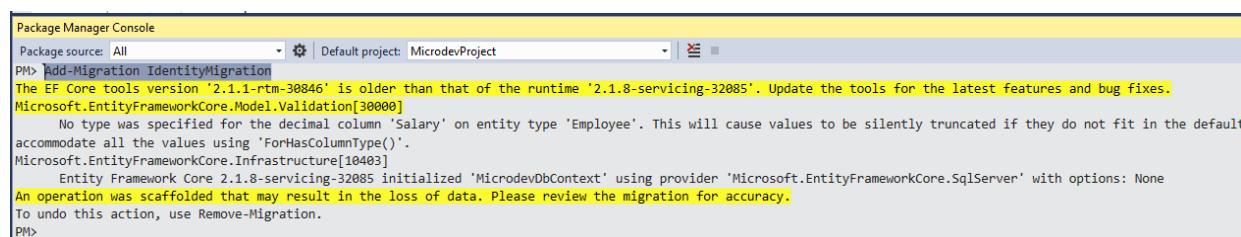
کلاس EF Core شامل DbSet<T> های ضروری است که User Entity را با استفاده از IdentityDbContext ذخیره می کند.

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

حالا همانطور که می دانید برای اعمال تغییرات باید Migration بزنید.

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

Add-Migration IdentityMigration



```
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-32085'. 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-32085 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.

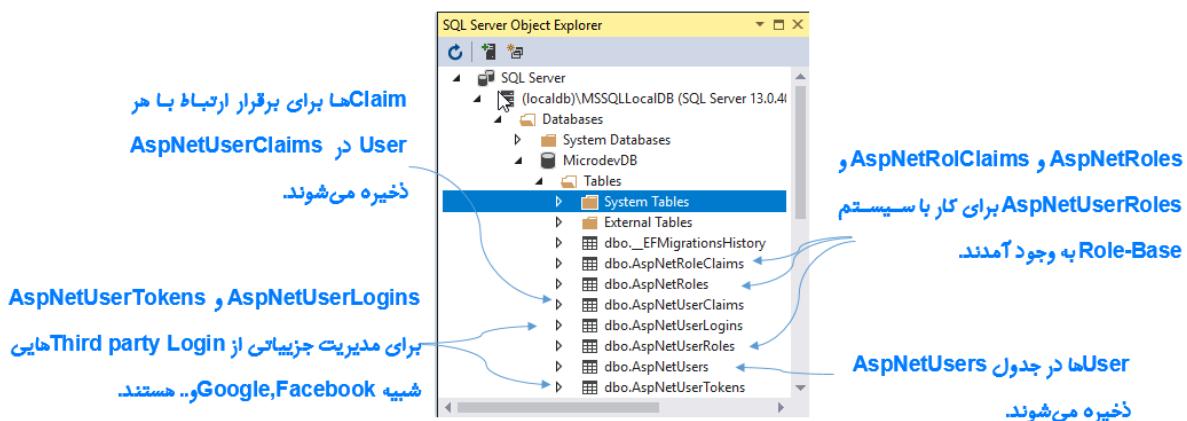
Update-Database

```

Package Manager Console
Package source: All | Default project: MicrodevProject | X
PM> Update-Database
The EF Core tools version '2.1.1-rtm-30846' is older than that of the runtime '2.1.8-servicing-32085'. 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-32085 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('[_EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
  Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
  SELECT OBJECT_ID('[_EFMigrationsHistory]');

```

بیایید با هم نمای دیتابیس و جداول اضافه شده را بینیم:



User مدیریت

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)           يافت User که ایمیل آن با
    {                           برابر باشد.
        user = new User();      اگر User یافت نشد یک نمونه از
        {
            FirstName = "Zahra",
            LastName = "Bayat",
            UserName = "BytZahra@Gmail.com",
            Email = "BytZahra@Gmail.com"   کلاس User ایجاد من کنیم.
        };
        سپس این User را توسط کلاس
        userManager.CreateAsync(user,           اگر User با موفقیت ایجاد نشود
        "P@ssw0rd!");                      یک اکسپشن پرتاب می شود.
        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="روشنده مدیریت"},           "روشنده مدیریت"
            new Department{Name="باز دکان"},                     "باز دکان"
            new Department{Name="پرداز ایده"},                  "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="افخاری 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);
    _context.Employees.ToList();
}
await _context.SaveChangesAsync();

}
}
}

```

حالا نوبت به رجیستر Identity در پروژه می‌رسد:

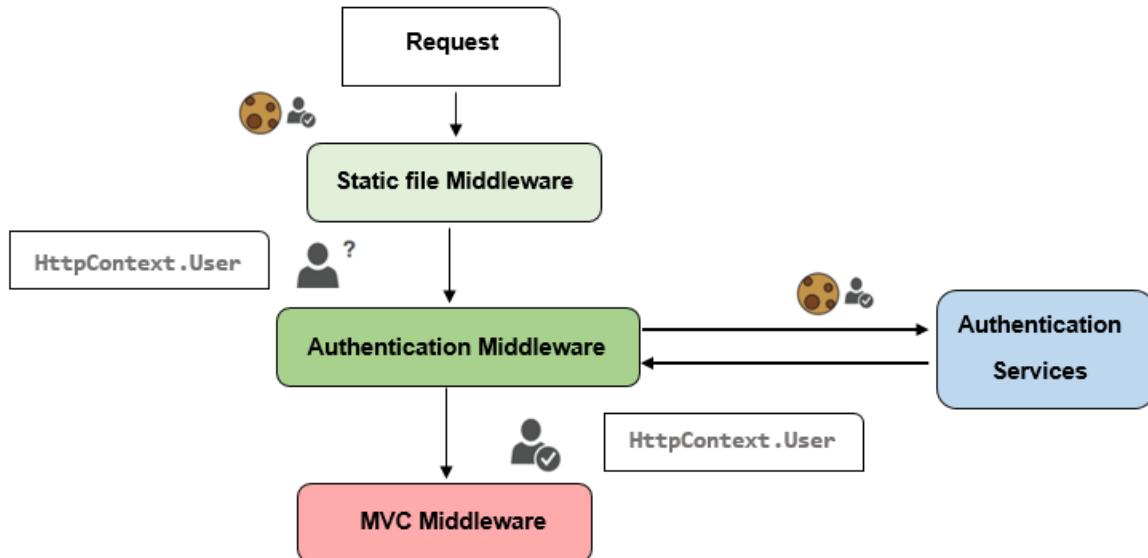
۱) پیکربندی AuthenticationMiddleware بسیار ساده است کافیست **Configure** را (قبل از **UseMvc**) در متدهای **UseAuthentication**

```

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



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

(۲) افروden سرویس `Authentication` و `Identity`

```

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

    services.AddTransient<Seeder>();
    services.AddTransient<IEmployeeService, EmployeeService>(); ← Identity System و پیکربندی

    DI به Identity
    services.AddIdentity<User, IdentityRole>(cfg => ← Role و انواع User
    {
        cfg.User.RequireUniqueEmail = true; ← پیکربندی Identity برای ذخیره
        ← داده هایش در EF Core
        cfg.User.AllEmailsMustBeConfirmed = true; ← داشتن ایمیل Unique برای User الزامیست.

        .AddEntityFrameworkStores<MicrodevDbContext>(); ← افزودن سرویس
        services.AddAuthentication(options => ← Authentication
        {
            options.DefaultScheme =
                CookieAuthenticationDefaults.AuthenticationScheme; ← برای تنظیم AddCookie سرویس
            options.ExternalAuthenticationSchemes.Add("MyExternalScheme"); ← جاری در کوکی من باشد.

            DI در
            services.AddMvc(); ← افزودن سرویس
        });
    });
}
    
```

نکاتی در مورد AddAuthentication!

✓ در کد `AddAuthentication` انجام می‌پذیرد و همانطور که در `Authentication` توسط `Asp.Net Core` انجام می‌پذیرد `AddAuthentication` تعدادی سرویس جهت تنظیم `Principal` جاری برای ها فراهم کرده است.

✓ یک `Authenticate` به نام `DefaultScheme` دارد، که نحوه‌ی `AddAuthentication` کردن کاربران را تعیین می‌کند.

✓ این `Authenticate` می‌تواند در کوکی، توکن و... باشد که در اینجا ما از کوکی استفاده کردہ‌ایم.

✓ قابلیت استفاده از توکن `bearer` برای `Authenticate` کردن API‌ها را هم دارد.

✓ در فضای نام زیر قرار دارد `CookieAuthenticationDefaults`

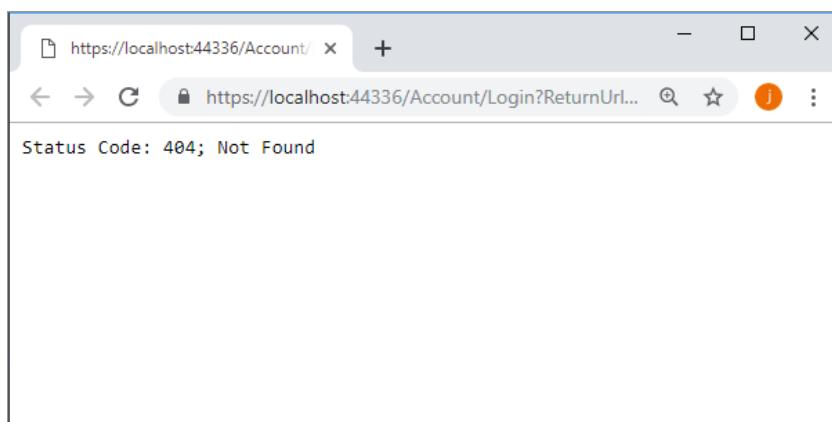
`Microsoft.AspNetCore.Authentication.Cookies;`

۳) استفاده از Authorize Attribute

در این مرحله بالای اکشن‌متد یک `[Authorize]` قرار دهید.

```
[Authorize]
// GET: Employee
public async Task<IActionResult> Index()
{
    IEnumerable<Employee> employees = await
        _service.GetAllEmployeesAsync();
    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 Controller‌ی به نام **Login** و View‌ی به نام **Account** در پروژه نداریم.

نکته!!

نمایش Status Code در این صفحه به این خاطر است که متدهای **Configure** را کمی تغییر دادیم و Handle‌ها را Status Code کردیم.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        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 کنترلری به نام **Login** با اکشن متدهای **AccountController** می‌کنیم.

```
using Microsoft.AspNetCore.Mvc;

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

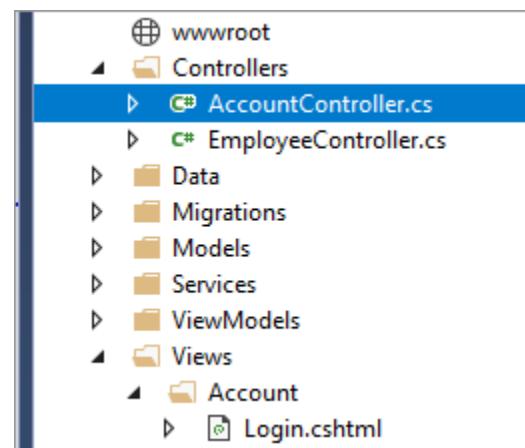
}

. مسئول رجیستر شدن کاربران و لایگین شدن آن‌هاست.

نکته!!

اگر کاربر لایگین نشده، بخواهد اکشنی را اجرا کند به صفحه لایگین منتقل خواهد شد، اما اگر لایگین شده باشد و دسترسی به اکشن‌متد مورد نظر نداشته باشد به صفحه Access Denied انتقال داده خواهد شد.(شما می‌توانید این صفحه را به پروژه خود اضافه نمایید.)

حالا در **Folder** یک **Folder** با نام **Account** و **View** با نام **Login** ایجاد می‌کنیم.



قبل از نوشتن کدهای صفحه‌ی **Login**، بباید تغییراتی را به **Layout** پردازه دهیم.

جهت اضافه کردن قابلیت لایگین، 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>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-
                controller="Departmant" 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">ورود به سایت</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 **Logout** خروج از سایت در نمایش داده شود، و گرنه **Button**‌های ورود به سایت و ایجاد حساب را خواهید دید.

حالا نیاز به یک **ViewModel** برای نگهداری اطلاعات **Login** داریم. در 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 = "بسپار 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 **ViewModels** شوید و فضای نام **ViewImport** را اضافه کنید:

```

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

```

حالا وارد **Login** View 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">
            نحوه نمایش خطاهای
            <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">
                        @Html.DisplayNameFor(m => m.RememberMe)
                    </label>
                </div>
            </div>
        </form>
    </div>
</div>

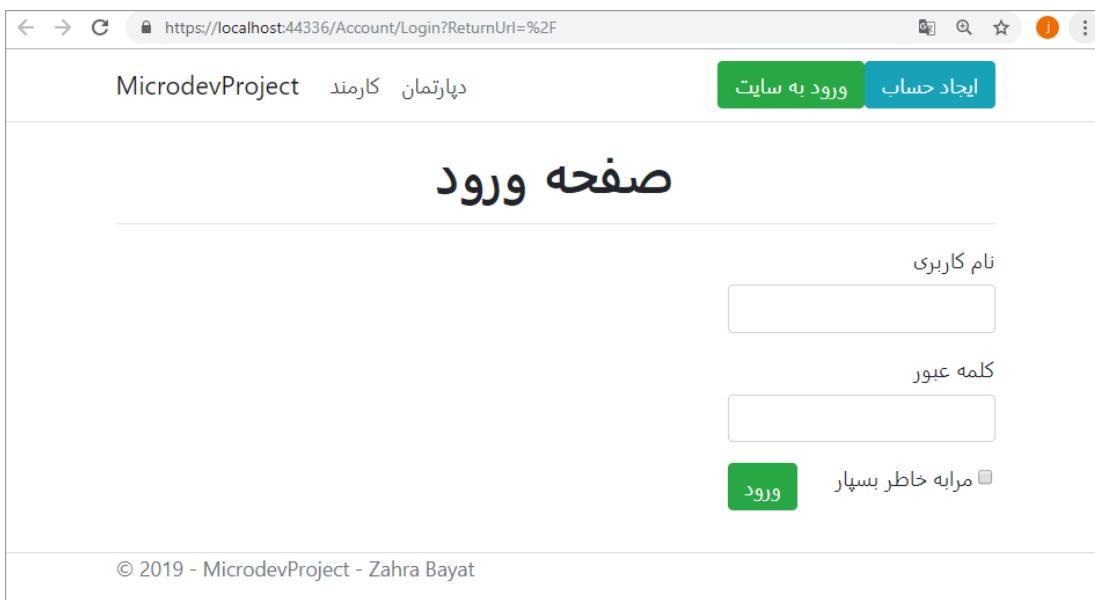
```

```

        </div>
    <div class="form-group">
        <input type="submit" value="ورود" class="btn btn-success" />
    </div>
</form>
</div>
</div>

```

حالا با اجرای برنامه وارد صفحه لاگین می‌شوید.



فرم لاگین ، درخواست کاربر برای ورود به سیستم را می‌گیرد و سپس اجاز دسترسی به کاربر را می‌دهد.

اگر می‌خواهید با کلیک بر روی دکمه ورود، لاگین شوید باید متده Post را برای اکشن Login بنویسید.

```

using MicrodevProject.Models;
using MicrodevProject.ViewModels;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

```

```
namespace MicrodevProject.Controllers
{
```

```
    public class AccountController : Controller
    {
```

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

برای ایجاد یک کاربر جدید می‌توانید از

کلاس UserManager<T> استفاده

کنید.

```
        public AccountController(
            SignInManager<User> signInManager,
```

کلاس login برای و پر کردن Cookies ها و همچنین لود logout

UserManager<User> userManager

کردن کاربر و چک کردن پسورد مورد استفاده قرار

می‌گیرد.

```

{
    _signInManager = signInManager;
    _userManager = userManager;
}

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

[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)           Username, password لیکن Sign in
    {
        var result = await
        _signInManager.PasswordSignInAsync(model.Username,
            model.Password,
            model.RememberMe,
            false);                      موفق باشد. Sign in و HttpContext.user Authentication Cookie
                                            تنظیم من شود.

        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Employee");
        }
    }

    ModelState.AddModelError("", "Failed to login");

    return View();
}

```

خوشحالم از اینکه توانستید قابلیت **Identity** را به پروژه اضافه و صفحه لایگین پروژه را راهاندازی کنید.

همراه ما باشید، می خواهیم کمی در مورد قابلیت **Claim Based** در **ASP.NET Core** صحبت کنم.

Claim چیست؟

یک جفت **Key- Value** (هر دو از نوع **string** هستند) است که مشخص کننده **Principal** می باشد.

Principal چیست؟

در **ASP.NET Core**، هر درخواست به یک کاربر مرتبط است، که به آن **Principal** گفته می شود. کاربری که نشده باشد، یک **Anonymous user** است. **Authenticate**

Property‌های اساسی مربوط به ASP.NET Core همانند Identity 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، می‌توانید 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 می‌توان Claim‌ی به نام NationalCode داشت که مشخص کننده کد ملی شما باشد.

پس گواهینامه مجازی مبتنی بر ادعاهای شماست. براساس این Claim‌ها، ارزش یک ادعا بررسی می‌شود و براساس این ارزش به شما اجازه داده می‌شود که دسترسی‌هایی داشته باشید.

افسر، صحت کارت شما را بررسی می‌کند و براساس اطلاعات درون کارت، اجازه رانندگی به شما می‌دهد.

یک Identity می‌تواند شامل Claim‌هایی با چندین مقدار باشد و حتی می‌تواند چندین Claim مشابه داشته باشد.

در ورژن‌های قبلی ASP.NET به جای Claim از Role استفاده می‌شد. هنوز هم می‌توانید از Roles استفاده کنید، اما بهتر است تا جایی که امکان دارد از Claim استفاده نمایید.

مثال زیر سعی کردیم این کلاس را شبیه‌سازی 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.

- این **Property** شامل تمام **Claim**‌های **Identity** است.
- این **Property** برای مشخص کردن نوع **Authentication** است. مثلاً **AuthenticationType** گواهینامه رانندگی که بالاتر توضیح دادیم، **AuthenticationType** است و در **ASP.NET** از طریق **Cookie**, **Bearer**, **Google** ... می‌توان صحت این **Property** را مشخص نمود.
- این **Property** مشخص می‌کند که کاربر **Authenticated** شده یا خیر؟
- تعدادی متدهای کاربردی هم برای **Claim**‌ها وجود دارد که در اینجا دو مورد آورده شده است. این متدها زمانی مفید است که کاربر **Autorize** شده باشد.

ایجاد یک Claim

تا اینجا با نحوه کارکرد **Claim** در **ASP.NET Core** آشنا شدید، حالا چطور می‌توان یک **Claim** ایجاد نمود؟ به مثال زیر توجه کنید. ما در این مثال می‌خواهیم بعد از لاگین شدن کاربر یک **Claim** به لیست **Claim**‌های آن اضافه نماییم:

```

[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Username,
            model.Password,
            model.RememberMe,
            false);
        يافتن کاربر با استفاده از
        ایمیل
        if (result.Succeeded)
        {
            var user = await _userManager.FindByEmailAsync(model.Username);

            await _userManager.AddClaimAsync(user, new
                Claim("EmployeeNumber", "nothing"));
            اضافه کردن Claim

            return RedirectToAction("Index", "Employee");
        }
    }
}

```

```
ModelState.AddModelError("", "Failed to login");  
return View();  
}
```

- اولین کاری که انجام دادیم، **User** جاری را با استفاده از ایمیل واکشی نمودیم.
- سپس یک **Claim** برای این کاربر درج می‌شود.

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

برای کاربر Claim

هر کاربر تعدادی **Claim** دارد، این **Claim**ها اطلاعاتی در مورد کاربر هستند (مثل نام، ایمیل و...) که شما می‌توانید هر جایی از آن‌ها استفاده کنید. کد پایین لیست تمام **Claim**‌های کاربر را نمایش می‌دهد. شما می‌توانید قطعه کد پایین را در هر جایی از پروژه اضافه نمایید.

```
@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 anonymous</div>  
}
```

ما در این پروژه تنها به **Claim**→**Name** نیاز داشتیم، که قبل از Layout اضافه نمودیم.

```
@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>
}

```

Claim Check افزودن

از **Claim** های کاربر جاری، جهت تعیین اینکه این کاربر اجازه اجرای چه اکشنی را دارد استفاده می‌کند. برای اینکه مشخص کنید چه **Claim** هایی برای اجرای یک اکشن متد نیاز است، می‌توانید از **Policy** ها استفاده کنید.

نیازمندی **Claim** ها مبتنی بر **Policy** است، که برنامه‌نویس باید **Policy** موردنیاز **Claim** را ایجاد و و رجیستر نماید.

افزودن **Policy** ها از طریق متد **AddPolicy** (در بدنه متد **AddAuthorization()** درون 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 **Claim** هایی است، که برای اجرا به آن نیاز دارد.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MicrodevDbContext>(option =>

        option.UseSqlServer(configuration.GetConnectionString("MicrodevConnect
            ion")));
    services.AddTransient<Seeder>();
    services.AddScoped<IEmployeeService, EmployeeService>();
    services.AddIdentity<User, IdentityRole>(cfg =>
    {
        cfg.User.RequireUniqueEmail = true;
    })
        .AddEntityFrameworkStores<MicrodevDbContext>();
    services.AddAuthentication(options =>
    {
        options.DefaultScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
    })
        .AddCookie();
    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 **AuthorizeAttribute** روی **Policy Property** قرار دهید.

```
[Authorize(Policy = "EmployeeOnly")]  
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);  
}
```

نکته!!

می تواند به کل Controller اعمال گردد.

```
[Authorize(Policy = "EmployeeOnly")]  
public class EmployeeController : Controller  
{  
    private readonly IEmployeeService _service;  
    public EmployeeController(IEmployeeService service)  
    {  
        _service = service;  
    }  
  
    //..  
}
```

شما اگر کنترلی دارید که دارای **AuthorizeAttribute** می باشد، و می خواهید اجازه **Anonymous access** در اکشن خاص را هم داشته باشید، می توانید از **AllowAnonymousAttribute** استفاده کنید.

```

[Authorize(Policy = "EmployeeOnly")]
public class EmployeeController : Controller
{
    private readonly IEmployeeservice _service;
    public EmployeeController(IEmployeeservice service)
    {
        _service = service;
    }

    public async Task<IActionResult> Index()
    {
        Ienumerable<Employee> employees = await _service.GetAllEmployeesAsync();
        return View(employees);
    }
}

این اکشن در هر صورتی اجرا می شود.

[AllowAnonymous]
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);
}

//..
}

```

ایجاد Logout

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

```

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

با یک HttpContext.user
جایگزین anonymous principal
منشود و خواهد شد.

```

همانطور که در کد بالا می بینید بعد از **Logout** شدن به صفحه **Index** بر می گرددید.

ثبت‌نام کاربر

صفحه Login/Logout شما با موفقیت ایجاد شد. اما متن‌افانه هنوز صفحه‌ای برای ثبت‌نام کاربر وجود ندارد.

در کنترلر Register یک متده است Account ایجاد نمایید.

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

این متده نمایش صفحه‌ی ثبت‌نام کاربر می‌باشد، اما چون هنوز View Register ایجاد نشده، اگر بر روی ایجاد حساب در navbar کلیک کنید، به خطاب خواهد خورد.

پس در مسیر Views/Account به نام View 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 class="form-group">
                <label asp-for="ConfirmPassword" class="control-label" style="FLOAT: right"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
            </div>
        </form>
    </div>
</div>
```

```

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

```

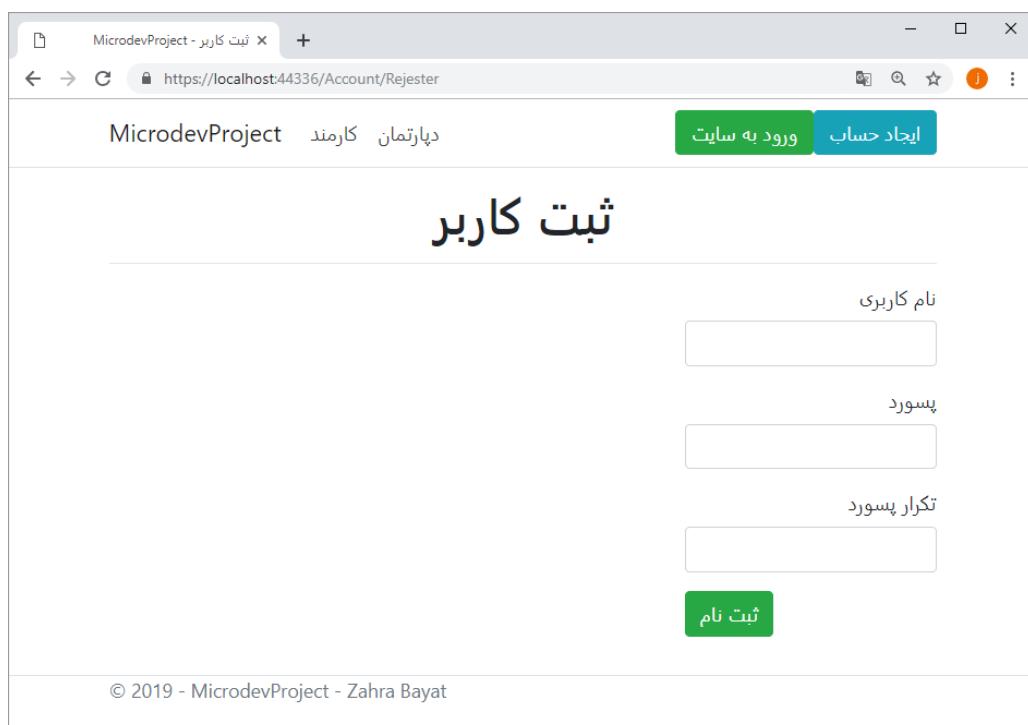
</span>
</div>
<div class="form-group">
<input type="submit" value="نام ثبت" class="btn btn-success" />
</div>
</form>
</div>
</div>

```

نکته!!

همانطور که مشاهده می‌کنید در View فوق از عبارت `using` برای Import Namespace کردن `ViewModel` حاوی `ViewImports.cshtml` استفاده نکرده‌ایم؛ زیرا این کار را یکبار درون فلیل `_ViewImports.cshtml` برای تمامی `View`‌ها 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` ایجاد کنید.

```

using System.ComponentModel.DataAnnotations;
namespace MicrodevProject.ViewModels

```

```

{
    public class RegisterUserViewModel
    {
        [EmailAddress]
        [Required, MaxLength(256), Display(Name = "نام کاربری")]
        public string Username { get; set; }

        [StringLength(100, ErrorMessage = "{0} حداقل و {2} حاکمتر کاراکتر باشد", MinimumLength = 6)]
        [Required, DataType(DataType.Password), Display(Name = "پسورد")]
        public string Password { get; set; }

        [DataType(DataType.Password), Compare(nameof(Password), ErrorMessage = "تکرار پسورد با هم متنطبق نیست"), Display(Name = "تکرار پسورد")]
        public string ConfirmPassword { get; set; }
    }
}

```

حالا نوبت به نوشتمن متدهای **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, Email= model.Username };

        IdentityResult result = await _userManager.CreateAsync(user,
            model.Password);
    }
    else
    {
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError("", error.Description)
        }
    }
}

```

فرم تنها محدود به **RegisterUserViewModel** است.

اگر ایجاد **User** موفق باشد، **Binding** معتبر باشد یک **User** ایجاد می‌شود.

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

اگر ایجاد **User** موفق باشد، **IdentityResult** را در دستور **CreateAsync** دریافت می‌کند.

اگر هر گونه خطایی رخداد کند، **ModelState** اضافه کردن **Claim** ("EmployeeNumber", "nothing") و **SignInManager** را در دستور **SignInAsync** دریافت می‌کند.

اگر خطاهای **ModelState** اضافه شود، **ModelError** را در دستور **TryAddModelError** دریافت می‌کند.

```

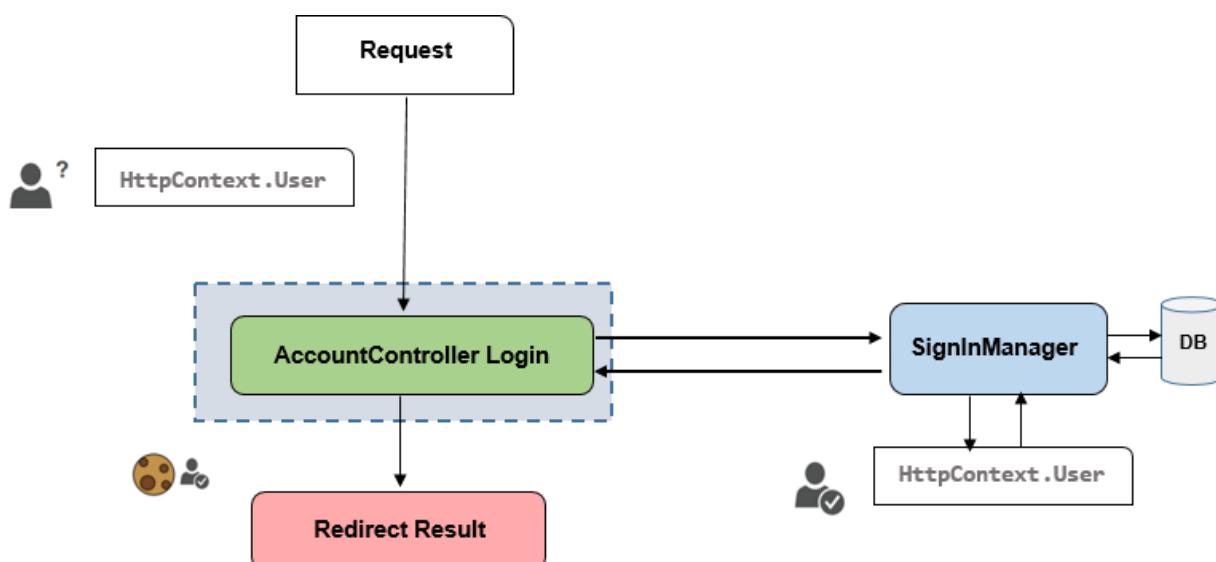
    return View();
}

```

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



کدهای `AccountController` بعد از تغییرات:

```

using MicrodevProject.Models;
using MicrodevProject.ViewModels;
using Microsoft.AspNetCore.Identity;

```

```
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using System.Threading.Tasks;

namespace MicrodevProject.Controllers
{
    public class AccountController : Controller
    {
        private readonly SignInManager<User> _signInManager;
        private readonly UserManager<User> _userManager;
        public AccountController(
            SignInManager<User> signInManager,
            UserManager<User> userManager
        )
        {
            _signInManager = signInManager;
            _userManager = userManager;
        }
        public IActionResult Login()
        {
            return View();
        }
        [HttpPost, ValidateAntiForgeryToken]
        public async Task<IActionResult> Login(LoginViewModel model)
        {
            if (ModelState.IsValid)
            {
                var result = await
                    _signInManager.PasswordSignInAsync(model.Username,
                    model.Password,
                    model.RememberMe,
                    false);
                if (result.Succeeded)
                {
                    var user = await
                        _userManager.FindByEmailAsync(model.Username);
                    await _userManager.AddClaimAsync(user, new
                        Claim("EmployeeNumber", "nothing"));

                    return RedirectToAction("Index", "Employee");
                }
            }
            ModelState.AddModelError("", "Failed to login");
            return View();
        }
        [HttpGet]
        public async Task<IActionResult> Logout()
        {
            await _signInManager.SignOutAsync();
            return RedirectToAction("Index", "Employee");
```

```

        }
    public IActionResult Register()
    {
        return View();
    }
    [HttpPost, ValidateAntiForgeryToken]
    public async Task<IActionResult> Register(RegisterUserViewModel model)
    {
        if (ModelState.IsValid)
        {
            User user = new User { UserName = model.Username, Email= model.Username };
            IdentityResult result = await _userManager.CreateAsync(user, model.Password);
            if (result.Succeeded)
            {
                await _userManager.AddClaimAsync(user, new Claim("EmployeeNumber", "nothing"));
                await _signInManager.SignInAsync(user, false);
                return RedirectToAction("Index", "Employee");
            }
            else
            {
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError("", error.Description);
                }
                return RedirectToAction("Index", "Employee");
            }
        }
        return View();
    }
}
}

```

کدهای EmployeeController بعد از تغییرات:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using MicrodevProject.Models;
using MicrodevProject.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
namespace MicrodevProject.Controllers
{
    public class EmployeeController : Controller
    {
        private readonly IEmployeeService _service;
        public EmployeeController(IEmployeeService service)

```

```
{  
    _service = service;  
}  
  
// GET: Employee  
[Authorize]  
public async Task<IActionResult> Index()  
{  
    IEnumerable<Employee> employees = await  
    _service.GetAllEmployeesAsync();  
    return View(employees);  
}  
  
// GET: Employee/Details/5  
[AllowAnonymous]  
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: Employee/Create  
[HttpGet]  
public IActionResult Create()  
{  
    ViewBag.Bosses = _service.GetDropDownEmployees();  
    ViewBag.Departments = _service.GetDropDownDepartments();  
    return View();  
}  
  
// POST: Employee/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: Employee/Edit/5
[Authorize(Policy = "EmployeeOnly")]
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: Employee/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: Employee/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    Employee employee = await _service.GetEmployeeAsync(id);
    if (employee == null)
    {
        return NotFound();
    }
}

```

```
var isBoos = await _service.IsBoos(employee.EmployeeId);
if (isBoos)
{
    return View("DeleteFaild", employee);
}

return View(employee);
}

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

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 ارزشتن 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 ویرایش های بعدی اصلاح کنم.
این تمام چیزی است که از شما انتظار دارم.

امیدوارم با تلاش های تک تک 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>

برای دانلود پروژه به آدرس زیر مراجعه کنید.

<https://github.com/ZahraBayatgh/MicrodevProject>

کتاب های نوشته شده:

