



www.MicroDev.ir

ASP.NET

C
5.0 re

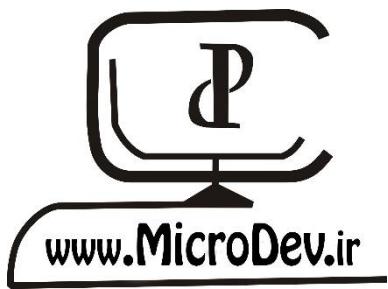
راهنمای عملی

Web API

مولفین:

زهرا بیات قلی لاله

علی بیات قلی لاله



راهنمای عملی

ASP.NET Core 5.0

Web API

زهرا بیات قلی‌لاله

علی بیات قلی‌لاله

راهنمای عملی

ASP.NET Core 5.0

Web API

مؤلفین : زهرا بیات قلی‌لاله – علی بیات قلی‌لاله

طراح جلد : زهرا بیات قلی‌لاله

مشخصات ظاهری : ۳۵۲ ص

سال انتشار: بهمن ۹۹

قیمت : رایگان

فهرست

۱۱	تقدیم به
۱۲	با تشکر
۱۳	درباره این کتاب
۱۴	فصل اول : تنظیمات پروژه
۱۵	تنظیمات پروژه
۱۵	ایجاد پروژه جدید
۱۷	تنظیمات فایل <code>launchSettings.json</code>
۱۹	کلاس <code>Startup</code> و <code>Program</code>
۲۱	کلاس <code>Startup</code>
۲۱	اکستنشن متدهای <code>CORS</code>
۲۳	اکستنشن متدهای <code>IIS</code>
۲۴	اعمال تنظیمات در <code>Startup</code>
۲۷	تنظیمات <code>Environment</code>
۳۰	فصل دوم : سرویس <code>Logger</code>
۳۱	پیکربندی سرویس <code>Logger</code>
۳۱	ایجاد پروژه های زیرساخت
۳۳	ایجاد اینترفیس <code>ILoggerManager</code> و نصب <code>NLog</code>
۳۴	پیاده سازی اینترفیس <code>ILoggerManager</code> و پیکربندی <code>Nlog.Config</code> فایل
۳۶	پیکربندی <code>Logger Service</code>

۳۸	و IoC چیست؟ DI
۴۱	تست سرویس با Logger
۴۴	فصل سوم : دیتابیس و Repository Pattern
۴۵	مدل دیتابیس و Repository Pattern
۴۵	ایجاد مدلها
۴۸	ایجاد کلاس Context
۴۹	چیست؟ ConnectionString
۵۱	رجیستر DbContext از طریق DI
۵۳	چیست؟ Migration
۵۵	چیست؟ Seed Data
۵۹	پیاده‌سازی Repository Pattern
۶۲	پیاده‌سازی کلاس‌های Repository
۶۴	ایجاد یک Repository Manager
۷۰	فصل چهارم : مسیریابی و REST
۷۱	کنترلرها و مسیریابی در Web API
۷۴	چیست؟ REST
۷۶	چیست؟ HTTP Method
۷۷	مقایسه بین PUT - POST - PATCH
۷۸	فرمت بازگشته REST
۷۸	قوانین نامگذاری مسیر اکشن متدها
۷۹	گرفتن اطلاعات شرکت‌ها از دیتابیس

۸۳	تست کردن اکشن متدهای GetCompanies
۸۴	کلاس‌های DTO در مقابل کلاس‌های Entity Model
۸۷	استفاده از ASP.NET Core در AutoMapper
۹۴	فصل پنجم : مدیریت Content Negotiation و Exception ها
۹۵	مدیریت Exception ها
۱۰۰	تست UseExceptionHandler
۱۰۲	گرفتن اطلاعات یک Resource از دیتابیس
۱۰۶	ارتباطات Web API در Parent/Child
۱۱۳	واکشی اطلاعات کارمند
۱۱۷	Content Negotiation چیست؟
۱۱۸	تغییر پیکربندی Response
۱۱۹	تست Content Negotiation
۱۲۰	محدود کردن Media Type
۱۲۱	ایجاد فرمت سفارشی
۱۲۵	فصل ششم : DELETE , POST و PUT
۱۲۶	مدیریت Post Request
۱۲۳	ایجاد Child Resource
۱۳۷	ایجاد Child Resource همراه با یک Parent
۱۳۹	ایجاد مجموعه‌ای از Resource ها
۱۴۵	ایجاد Model Binding چیست؟
۱۴۹	ایجاد اکشن متدهای Delete

۱۵۲	حذف یک Parent Resource همراه با Childها یش
۱۵۷	آپدیت Employee
۱۶۰	انواع روش‌های Update
۱۶۱	ایجاد Resource در هنگام آپدیت یک Resource
۱۶۵	فصل هفتم : Patch Request و اعتبارسنجی
۱۶۶	کار کردن با Patch Request
۱۶۸	اضافه کردن Employee Entity به Patch
۱۷۴	اعتبارسنجی چیست؟
۱۷۶	ولیدیشن در زمان ایجاد Resource
۱۸۳	اعتبارسنجی نوع int
۱۸۵	اعتبارسنجی برای PUT Request
۱۸۸	اعتبارسنجی برای PATCH Request
۱۹۴	فصل هشتم : برنامه‌نویسی Async و Action Filter ها
۱۹۵	برنامه‌نویسی Async چیست؟
۱۹۶	کلمه‌های کلیدی await و async
۱۹۷	Rيفكتور Repository
۲۰۱	Rيفكتور CompaniesController
۲۱۶	Action Filter چیست؟
۲۱۶	پیاده‌سازی Action Filter
۲۱۸	Action Filter سطح
۲۱۹	ترتیب اجرا شدن Filter ها

۲۲۰	Action Filter با اعتبار سنجی
۲۲۶	Action Filter در Dependency Injection
۲۳۳	فصل نهم : Paging, Filtering, Searching
۲۳۴	Paging چیست؟
۲۳۴	Paging پیاده سازی
۲۴۰	Paging ارتقای
۲۴۶	Filtering چیست؟
۲۴۷	پیاده سازی Filtering در ASP.NET Core
۲۴۹	ارسال و تست Filtering
۲۵۱	Searching چیست؟
۲۵۲	پیاده سازی جستجو در اپلیکیشن
۲۵۴	تست پیاده سازی Searching
۲۵۵	Data Shaping و Sorting فصل دهم :
۲۵۶	Sorting چیست؟
۲۵۶	پیاده سازی Sorting در ASP.NET Core
۲۶۲	تست Sorting
۲۶۳	Data Shaping چیست؟
۲۶۳	چطور Data Shaping را پیاده سازی کنیم؟
۲۷۲	API Versioning : فصل یازدهم
۲۷۳	API Versioning
۲۷۴	Versioning تست

۲۷۷	استفاده از URL Versioning
۲۷۸	HTTP Header Versioning
۲۷۹	منسوخ کردن Version
۲۸۱	فصل دوازدهم : Rate Limiting و Cache
۲۸۲	Caching چیست؟
۲۸۲	انواع Cache
۲۸۳	افزودن هدرهای Cache
۲۸۵	اضافه کردن cache-store
۲۸۸	Expiration Model چیست؟
۲۹۰	Validation Model چیست؟
۲۹۱	پیاده‌سازی اعتبارسنجی
۲۹۳	پیکربندی هدرهای Validation و Expiration
۲۹۵	Rate Limiting چیست؟
۲۹۶	Rate-Limit
۳۰۰	فصل سیزدهم : Identity و JWT
۳۰۱	Identity و JWT
۳۰۱	ASP.NET Identity چیست؟
۳۰۲	Authorization و Authentication چیست؟
۳۰۶	ایجاد جداول و اضافه کردن Roleها
۳۱۰	ایجاد User
۳۱۶	فرآیند Login

۳۱۷	JWT چیست؟
۳۱۹	پیکربندی JWT
۳۲۳	پیاده‌سازی Authentication
۳۳۳	اعتبارسنجی براساس Role
۳۳۶	فصل چهاردهم : ایجاد داکیومنت با Swagger
۳۳۷	داکیومنت API چیست؟
۳۴۷	Swagger چیست؟
۳۴۶	تنظیمات Swagger

تقدیم به

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

با تشکر

برنامه نویس‌ها معمولاً به داشتن یک دوست هنرمند افتخار می‌کنند. من هم خیلی خوشحالم که با یکی از بهترین گرافیست‌های کشورم یعنی خانم عسل ضیایی دوست هستم و تشکر می‌کنم که با ایده‌های زیباییش به هر چه بهتر شدن این کتاب کمک کردند.

درباره این کتاب

این کتاب برای برنامه‌نویسانی نوشته شده که می‌خواهند به صورت عملی، صفر تا صد ساخت Web API را برای ورود به بازار کار یاد بگیرند.

این روزها به دلیل استفاده روز افزون از اپلیکیشن‌های موبایل، نیاز به API‌هایی که امکان ارائه سرویس به هزاران کلاینت به طور همزمان را دارند به یک نیاز حیاتی تبدیل شده است.

اگر نگاهی به آگهی‌های شغلی بیندازید خواهید دید که سهم عمده‌ای از بازار برنامه‌نویسی نیز مربوط به توسعه‌ی API است.

از این رو در این کتاب سعی کردم برای برطرف کردن نیازهایی که در پژوهش‌های وجود دارد مثال بیاورم و تجربیاتم را با شما به اشتراک بگذارم. این کتاب می‌تواند یک نقشه راه برای استفاده در پژوهش‌های واقعی باشد.

فصل اول : تنظیمات پروژه

آنچه خواهید آموخت:

- برسی تنظیمات فایل **launchSettings.json**
- آشنایی با کلاس **Startup Program**
- اعمال تنظیمات **CORS** و **IIS** در پروژه
- برسی تنظیمات **Environment**

تنظیمات پروژه

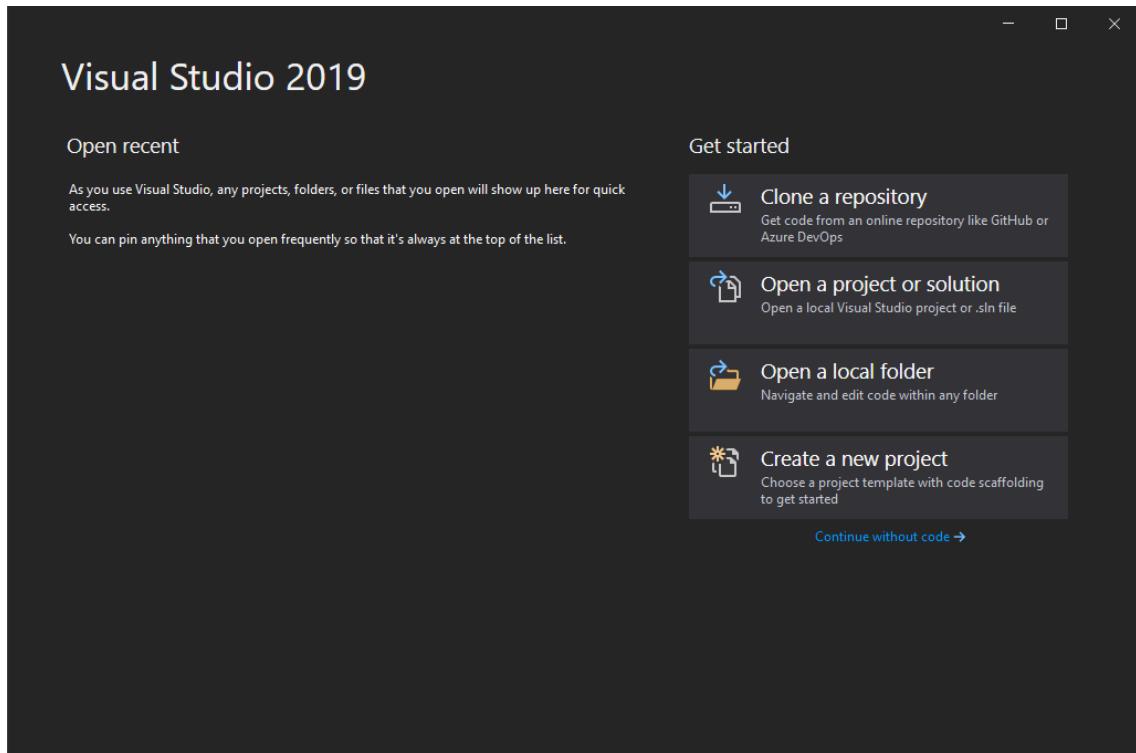
برای اینکه بتوانیم اپلیکیشن‌های خوبی را توسعه دهیم باید قبل از هر کاری، بدانیم که چطور اپلیکیشن و سرویس‌های درون آن را پیکربندی کنیم.

تنظیمات در.NET Core با چیزی که در پروژه‌های.NET Framework استفاده می‌کردید خیلی فرق دارد. در پروژه‌های.NET Core، دیگر خبری از فایل web.config نیست و ما از تنظیمات داخلی فریمورک استفاده می‌کنیم.

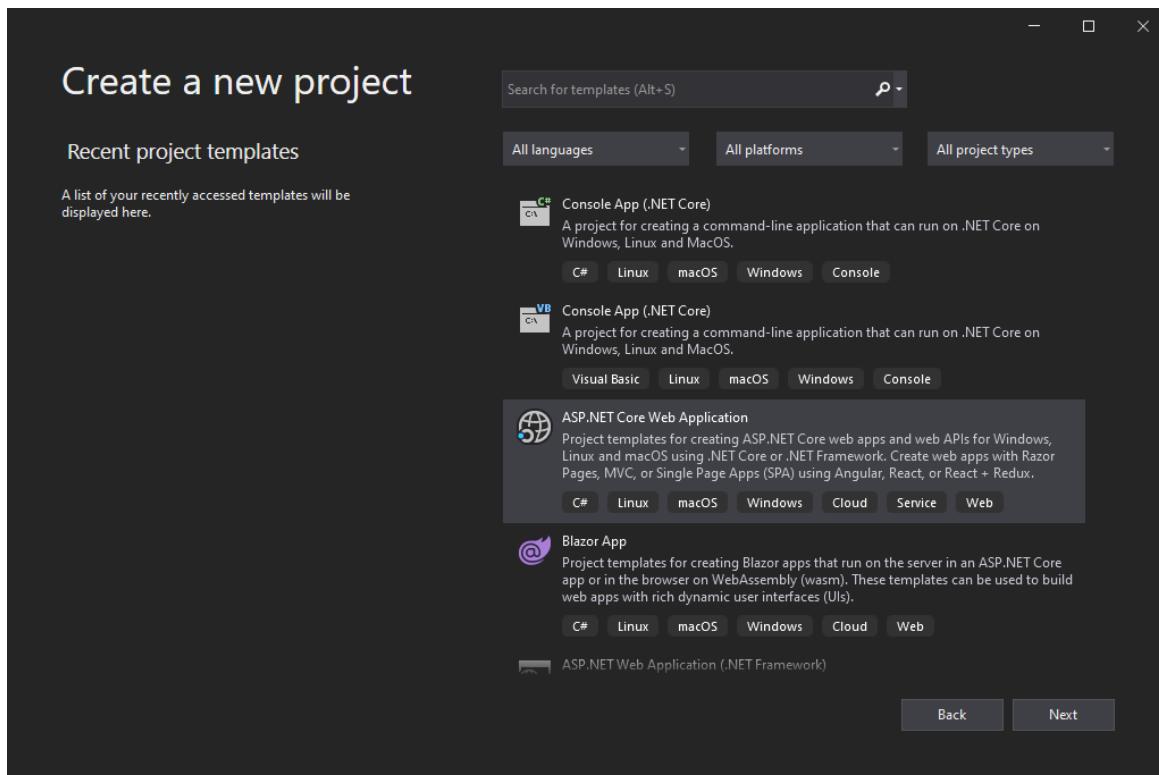
شما در این فصل با کلاس Startup و متدهای درون آن آشنا می‌شوید و یاد می‌گیرید چطور سرویس‌های خود را رجیستر کنید.

ایجاد پروژه جدید

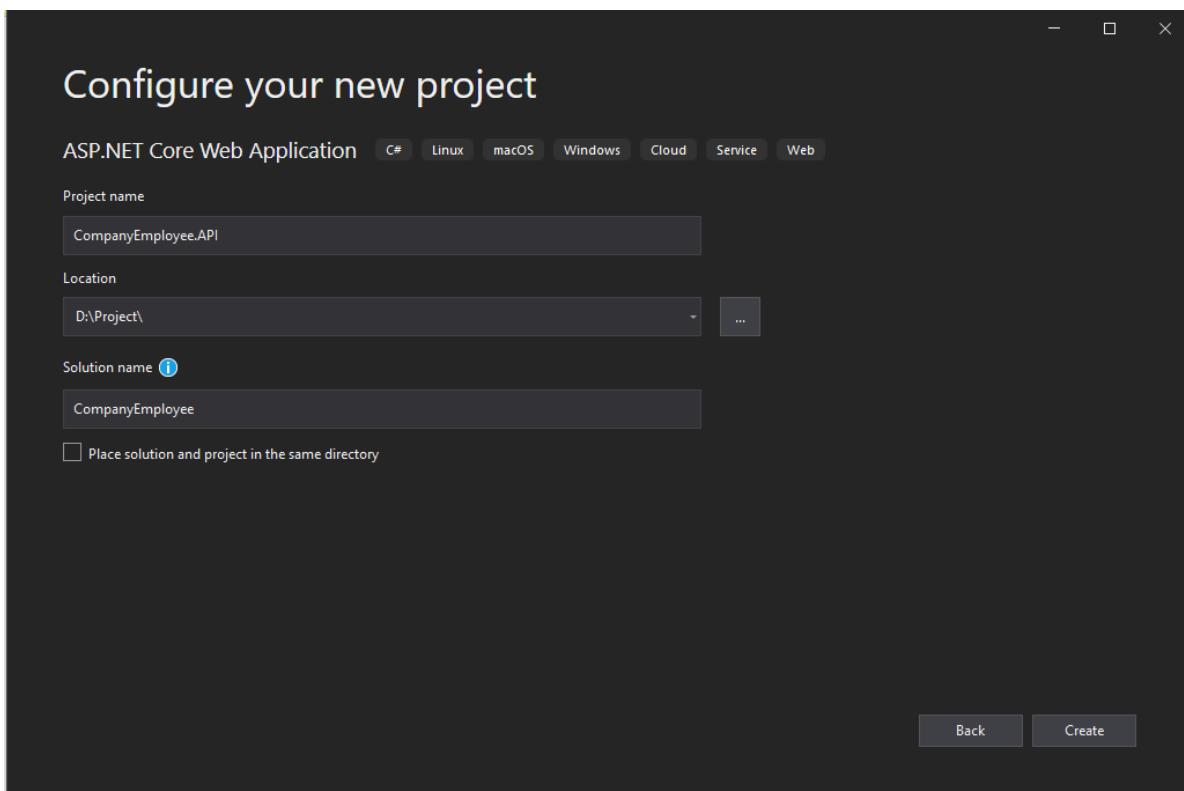
را باز کنید و روی Create a new project کلیک کنید.



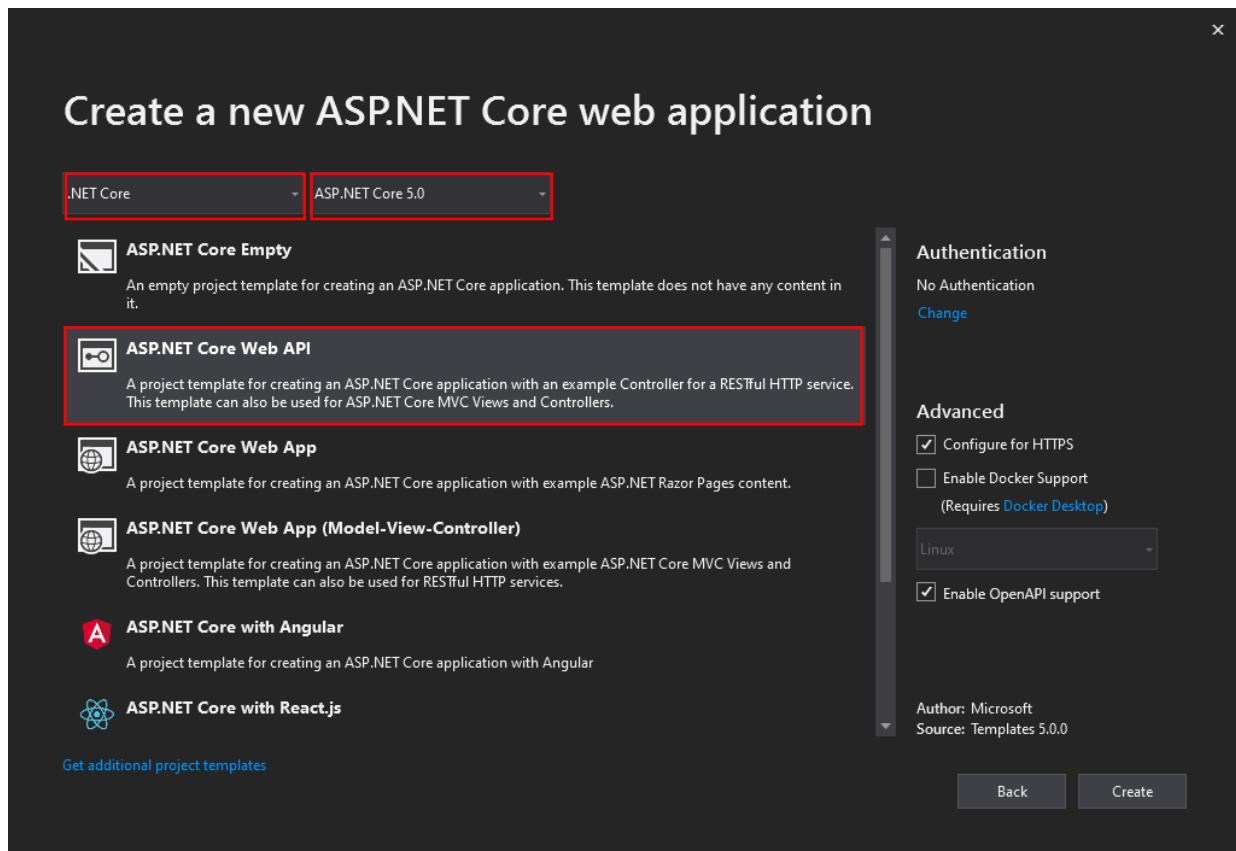
در کادر بعدی گزینه ASP.NET Core Web Application را انتخاب و بر روی Next بزنید.



حالا نام و مسیر پروژه را انتخاب کنید.



مرحله بعد انتخاب ASP.Net Core Web API و ASP.NET Core 5.0 ، .NET Core است.

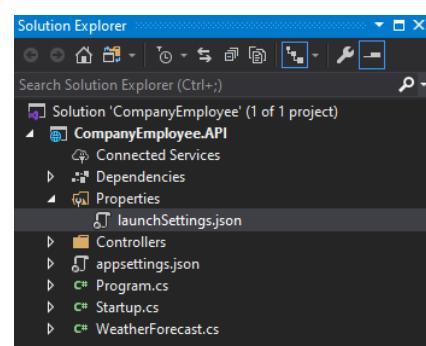


حالا بر روی Create کلیک کنید تا پروژه ایجاد شود.

تنظیمات فایل launchSettings.json

فایل launchSettings.json رفتار ASP.NET Core را در زمان راهاندازی اپلیکیشن تعیین می‌کند. این فایل شامل تنظیمات مربوط به IIS و اپلیکیشن‌های self-hosted است (Kestrel).

فایل launchSettings.json را می‌توان در بخش Properties (پنجره Solution Explorer) ببینید.



فایل : launchSettings.json

```

{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:58753",
      "sslPort": 44370
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "weatherforecast",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "CompanyEmployees": {
      "commandName": "Project",
      "launchBrowser": false,
      "launchUrl": "weatherforecast",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

بررسی فایل :

- در این فایل یک پرپرتی launchBrowser وجود دارد که تعیین می‌کند، زمان استارت اپلیکیشن، مرورگر باز شود یا خیر.
اگر برای چک کردن API از Postman استفاده می‌کنید و نیازی به مرورگر ندارید، پس مقدار این پرپرتی را می‌توانید به false تغییر دهید.

"launchBrowser": false,

- اگر چک باکس مربوط به HTTPS را در مرحله Setup تیک زده باشید باید در بخش HTTPS URLها یکی برای HTTP و دیگری برای applicationUrl است.

`"applicationUrl": "https://localhost:5001;http://localhost:5000"`

- در این فایل یک پرآپرتی sslPort وجود دارد که یک عدد می‌گیرد. این عدد پورت HTTPS را تنظیم و زمانیکه اپلیکیشن را با IISExpress اجرا کنید این پورت به اپلیکیشن داده خواهد شد.

`"sslPort": 44370`

نکته!!

توجه داشته باشید که این پیکربندی HTTPS، تنها در محیط Local معتبر است. پس زمانیکه برنامه را در سروری Deploy می‌کنید باید این HTTPS را مجدداً پیکربندی و در اینجا یک HTTPS معتبر قرار دهید.

- در اینجا یک Property با نام launchUrl وجود دارد که تعیین می‌کند کدام URL در ابتدای اپلیکیشن اجرا شود. البته برای اینکه این Property به درستی کار کند باید launchBrowser را نیز بر روی True تنظیم کنید.

بنابراین اگر پرآپرتی launchUrl بر روی weatherforecast باشد، زمانیکه برنامه اجرا شود، اپلیکیشن به مسیر زیر هدایت خواهد شد.

<https://localhost:5001/weatherforecast>

کلاس Startup و Program

تمام اپلیکیشن‌های ASP.NET Core همانند برنامه‌های Console، با یک کلاس Program.cs شروع می‌شوند. این کلاس یک متدهای Main دارد و نقطه ورود اپلیکیشن است. زمانی که اپلیکیشن Start شود، این متدهای اجرا خواهد شد.

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

namespace CompanyEmployee.API
```

```

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

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}

```

اگر با .NET Core 1.0 آشنايی داشته باشيد، می بینيد که اين کلاس از اين ورژن تاکنون خيلي کوچک شده است. بعضی قسمت ها مثل () UseKestrel() یا () UseIISIntegration() دیگر وجود ندارد چون متده CreateDefaultBuilder(args) برای خولانیي کد، تمام اين موارد را کپی سوله کرده است. البته شما هنوز هم می توانيد اين تنظیمات را دستی انجام دهيد.

اين متده کد () webBuilder.UseStartup<Startup> را صدا می زند تا کلاس Startup را Initialize کند. کلاس Startup در پروژه های ASP.NET Core Web API الزامي است. در اين کلاس ما تمامي سرويس های موردنیاز اپلیکیشن را پیکربندی می کنيم. پایین تر در مورد اين کلاس بیشتر توضیح داده شده است.

نکته !!

متده CreateDefaultBuilder(args) فلیل های پیشفرض، متغیر های پروژه و پیکربندی Logger را تنظیم می کند.

قبل از Bootstrapping انجام می شد و ما می توانستیم هر مشکلی که در طول Start اتفاق می افتاد را، با آن Log کنیم. این فرآیند در نسخه های قبلی سخت تر بود.

Startup کلاس

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

هرگونه پیکربندی که باید در زمان اجرای ASP.NET Core انجام شود، از این کلاس شروع خواهد شد. این کلاس یک Constructor و دو متده دارد که مسئول پیکربندی وب اپلیکیشن است.

- **متد ConfigureServices** : این متدهمانطور که از نامش معلوم است، برای رجیستر کردن سرویس ها می باشد.
سرویس کلاسی است که برخی از قابلیت‌ها را به اپلیکیشن اضافه می‌کند. یا به زبان ساده‌تر : هر کلاسی که اپلیکیشن به آن وابسته باشد (چه در اپلیکیشن مشخص شود و چه توسط فریمورک استفاده شود) باید در این متدرجیسترن شود.
 - **متد Configure** : با این متدمی توانیم **Middleware**‌های مختلف را به اپلیکیشن اضافه کنیم. **Middleware** تکه کدی است که می‌تواند **HTTP Request** و **HTTP Response** را پردازش، تغییر و در نهایت به **HTTP Response** بعدی دهد.

نکته!!

از آنجا که اپلیکیشن‌های بزرگتر، سرویس‌های بیشتری هم دارند پس حتماً کدهای آشفته زیادی در متدهای `ConfigureServices` خواهیم داشت. برای آنکه خوانایی این متدها بالاتر رود، می‌توانیم با استفاده از اکستنشن متدها، ساختاری تعریف کنیم و از این آشфтگی دور باشیم.

CORS متد تنظیمات اکستنشن

اکستنشن متد چیست؟

اکستنشن متدهای استاتیک این است که باید قبل از اولین پارامتر ورودی آن، کلمه کلیدی this را قرار دهید. این اولین پارامتر، نشان دهنده‌ی نوع داده‌ای است که این اکستنشن متدهای آن یک قابلیت اضافه می‌کند.

توجه داشته باشید که اکستنشن متدهای درون یک کلاس استاتیک تعریف شود.

خب حالا بباید کدنویسی را شروع کنیم تا ببینیم این چیزهایی که گفتیم چیست؟ به چه دردی میخورد؟ و چطور باید اضافه شود؟

یک فolder جدید با نام Infrastructure ایجاد کنید سپس درون این فolder یک فolder دیگر با نام Extensions اضافه نمایید.

حالا در اینجا یک کلاس استاتیک با نام ServiceExtensions ایجاد کنید.

```
namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions
    {
    }
}
```

خب حالا میتوانیم چیزهایی که در پروژه به آن نیاز داریم را با این کلاس پیاده‌سازی کنیم.

اولین کاری که میخواهیم انجام دهم پیکربندی CORS (Cross-Origin Resource Sharing) مکانیزم دسترسی دادن یا محدود کردن دسترسی به دومین‌های مختلف اپلیکیشن است.

اگر بخواهیم در اپلیکیشن درخواست‌هایی را از یک Domain به Domain دیگر بفرستیم، پیکربندی CORS الزامی است. بنابراین برای شروع میخواهیم کدی را اضافه کنم که به تمامی درخواست‌ها، از هر مبدا‌ی اجازه دسترسی به API‌های ما را بدهد.

```
using Microsoft.Extensions.DependencyInjection;

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions
    {
        public static void ConfigureCors(this IServiceCollection services)
        =>
            services.AddCors(options =>
            {
                options.AddPolicy("CorsPolicy", builder =>
                    builder.AllowAnyOrigin()
                );
            });
    }
}
```

```

        .AllowAnyMethod()
        .AllowAnyHeader());
    });
}

}

```

بررسی کد :

- در کد بالا از تنظیمات پایه‌ی CORS policy استفاده کردیم تا به هر Method و Origin دسترسی دهیم.

توجه داشته باشید این تنظیمات در محیط Production باید محدودتر باشد. بنابراین در این محیط به جای استفاده از AllowAnyOrigin() باید از WithOrigins("https://example.com") استفاده کنیم تا درخواست‌ها، تنها از منبع موردنظر اجازه دسترسی داشته باشد.

- همچنین () AllowAnyMethod به تمامی HTTP method اجازه دسترسی می‌دهد، در حالیکه ما می‌توانیم با استفاده از متدهای HTTP WithMethods("POST", "GET") تنها به method موردنظر اجازه دهیم.
- و می‌توانیم متدهای AllowAnyHeader را هم تغییر دهید تا فقط به هدرهای خاص اجازه دسترسی دهد. برای مثال :

```
WithHeaders("accept", "contenttype")
```

IIS اکستنشن متدهای تنظیمات

اپلیکیشن‌های ASP.NET Core به صورت پیش فرض Self-Hosted هستند و اگر بخواهیم اپلیکیشن خود را روی IIS هاست کنیم باید تنظیمات IIS integration را انجام دهیم. بنابراین برای انجام این کار ما کد زیر را به کلاس ServiceExtensions اضافه می‌کنیم.

```

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

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions

```

```

    {
        public static void ConfigureCors(this IServiceCollection
            services) =>
        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy", builder =>
                builder.AllowAnyOrigin()
                    .AllowAnyMethod()
                    .AllowAnyHeader());
        });
    }

    public static void ConfigureIISIntegration(this IServiceCollection
        services) => services.Configure<IISSettings>(options =>
    {
        });
    }
}

```

اعمال تنظیمات در Startup

خب تا اینجا ما اکستنشن متدهای خود را برای سازماندهی کد نوشتهیم. حالا برای اینکه اپلیکیشن از CORS و IIS integration پشتیبانی کند باید به کلاس Startup برگردیم تا این تنظیمات را در متدهای این کلاس اضافه کنیم.

```

using CompanyEmployee.API.Infrastructure.Extensions;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpOverrides;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.OpenApi.Models;

namespace CompanyEmployee.API
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
    }
}

```

```

{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
    services.ConfigureIISIntegration();
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title =
            "CompanyEmployee.API", Version = "v1" });
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c =>
            c.SwaggerEndpoint("/swagger/v1/swagger.json",
                "CompanyEmployee.API v1"));
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCors("CorsPolicy");
    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.All
    });

    app.UseRouting();
    app.UseAuthorization();
}

```

```

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}

```

بررسی کد :

- در کد بالا تنظیمات CORS و IIS را در متدهای ConfigureServices اضافه کردیم و سپس پیکربندی CORS در Pipeline را در متدهای Configure انجام دادیم.
- پیکربندی در .NET 5.0 با ورژن 3.1 فرق زیادی ندارد اما با این حال تغییراتی را در کلاس Startup می‌بینیم که بهتر است آن‌ها را با هم بررسی کنیم.
- در متدهای ConfigureServices AddControllers و AddSwaggerGen را می‌بینیم. متدهای AddControllers و AddSwaggerGen مانند ورژن قبل، وظیفه‌ی رجیستر کردن کنترلرهای ServiceCollection را برعهده دارد. این متدهای Viewها و Pageها کاری ندارد چون در پروژه Web API به آنها نیازی نداریم.
- متدهای AddSwaggerGen هم برای اضافه کردن Swagger جهت تست WebAPI است.
- در متدهای UseAuthorization و UseRouting متدهای Configure به ترتیب برای اضافه کردن قابلیت‌های Authorization و Routing به اپلیکیشن هستند.
- متدهای MapControllers هم وظیفه‌ی Routing را برعهده دارند.
- متدهای Forward و UseForwardedHeaders برای کردن هدرهای Proxy به ریکوئست جاری است.
- متدهای UseStaticFiles، استفاده از فایل‌های استاتیک را برای ریکوئست مهیا می‌کند. اگر مسیری را برای دایرکتوری فایل‌های استاتیک تنظیم نکنید، ریکوئست به صورت پیش‌فرض از فolder wwwroot استفاده می‌کند.

نکته!!

توجه داشته باشید که ترتیب اضافه کردن **Middleware**ها بسیار مهم است بنابراین متدهای **UseRouting** را باید بعد از **UseAuthorization** قرار دهید و همچنین متدهای **UseCors** یا **UseStaticFiles** را باید قبل از **UseRouting** صدا زده شوند.

تنظیمات Environment

ما در حال حاضر برنامه خود را در محیط Development توسعه می‌دهیم اما به محض اینکه برنامه خود را Publish کنیم، اپلیکیشن به محیط Production می‌رود.

محیط Production و Development باشد. Connection string، URLها، Portها، Passwordها و دیگر تنظیمات حساس آنها از هم مستقل باشد. بنابراین باید برای هر کدام از محیط‌ها، پیکربندی جداگانه‌ای داشته باشیم که این در .NET 5.0 ساده است.

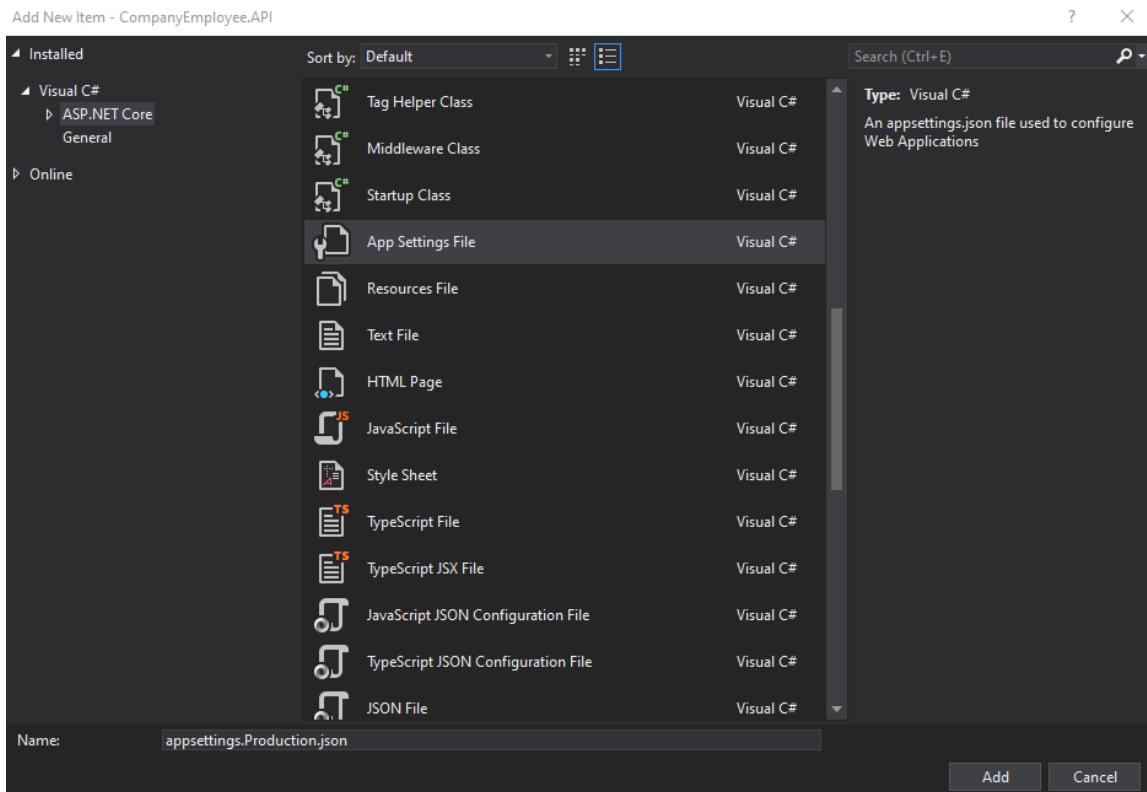
در اینجا بهتر است به سراغ فایل appsettings.json ببریم. این فایل حاوی تنظیمات ماست. اگر این فایل را Expand کنید می‌بینید که به صورت پیش فرض یک فایل appsettings.Development.json وجود دارد.



فایل‌های appsettings.json برای override کردن appsettings.{EnvironmentSuffix}.json اصلی استفاده می‌شود. ما همچنین می‌توانیم یک محیط خاص را در پروژه ایجاد کنیم. به طور مثال : محیط Production

برای ایجاد محیط Production باید بر روی پروژه راست کلیک کنید و از منو باز شده گزینه‌ی Add → New Item را انتخاب نمایید.

حالا از کادر باز شده App Settings File را انتخاب و نام این فایل را appsettings.Production.json بگذارید.



بعد از زدن Add یک محیط جدید اضافه خواهد شد.

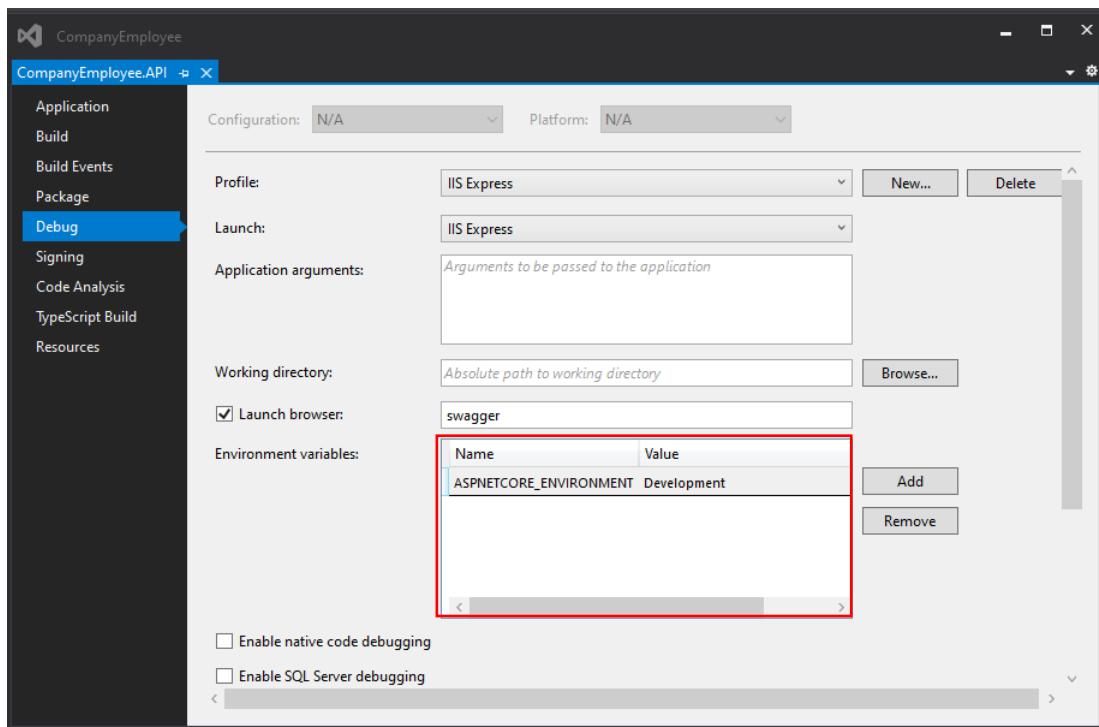


خب تا اینجا در مورد محیط‌های توسعه اپلیکیشن و نحوه ایجاد آن‌ها چیزهایی را یاد گرفتیم
اما چطور باید مشخص کنیم که اپلیکیشن در کدام یک از محیط‌ها اجرا شود؟

ASP.NET Core برای مشخص شدن ASPNETCORE_ENVIRONMENT متغیری است که اینجا در مورد محیط اپلیکیشن آن را جستجو خواهد کرد. اگر مقدار این متغیر برابر با Development باشد، ASP.NET فرض که در محیط Production گذاشت. اگر این Property وجود نداشته باشد برابر با true خواهد شد. این این دو روش تغییر داد :

این متغیر را می‌توانید از دو روش تغییر داد :

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



حالا در کادر Environment variables می‌توانید مقدار متغیر ASPNETCORE_ENVIRONMENT را تغییر دهید.

روش دوم: در فایل launchSettings.json مقدار متغیر ASPNETCORE_ENVIRONMENT را تغییر دهید.

```

20   "CompanyEmployee.API": {
21     "commandName": "Project",
22     "launchBrowser": true,
23     "launchUrl": "api/values",
24     "environmentVariables": {
25       "ASPNETCORE_ENVIRONMENT": "Development"
26     },
27     "applicationUrl": "https://localhost:5001;http://localhost:5000"
28   },
29   "Docker": {
30     "commandName": "Docker",
31     "launchBrowser": true,
32     "launchUrl": "{Scheme}://{ServiceHost}/api/values"
33   }
34 }
35

```

فصل دوم : سرویس Logger

آنچه خواهید آموخت:

- پیکربندی سرویس **Logger** چیست؟
- **IoC** و **DI**
- تست سرویس **Logger** با **Postman**

پیکربندی سرویس Logger

چرا پیام‌های Log در طول توسعه اپلیکیشن بسیار مهم هستند؟

اگر در زمان توسعه اپلیکیشن خطایی پیش بیاید، به راحتی می‌توانیم کدهایمان را دیباگ و مشکلی که پیش آمده را حل کنیم. اما دیباگ کردن در محیط Production ساده نیست به همین دلیل پیام‌های Log می‌تواند یک راه حل خوب برای فهمیدن مشکل و بررسی اکسپشن باشد.

علاوه بر این، با استفاده از Log می‌توان در زمانیکه به دیباگر دسترسی نداریم، به راحتی جریان برنامه را کنترل کنیم.

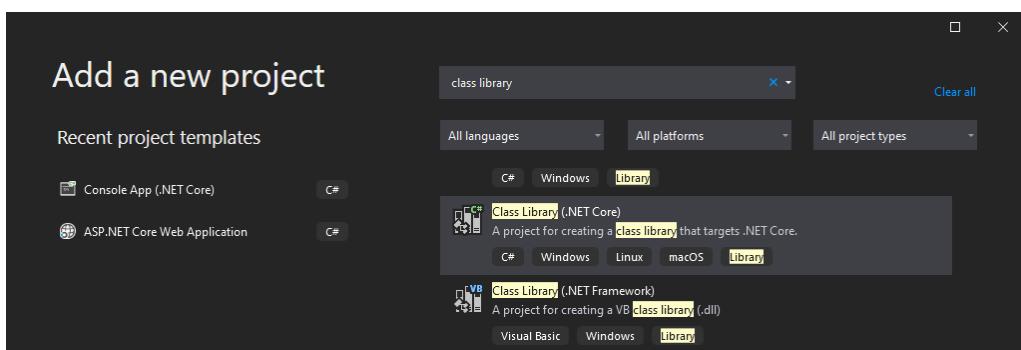
ما در این فصل می‌خواهیم با استفاده از پکیج NLog، یک سرویس سفارشی Log به پروژه اضافه کنیم.

ایجاد پروژه‌های زیرساخت

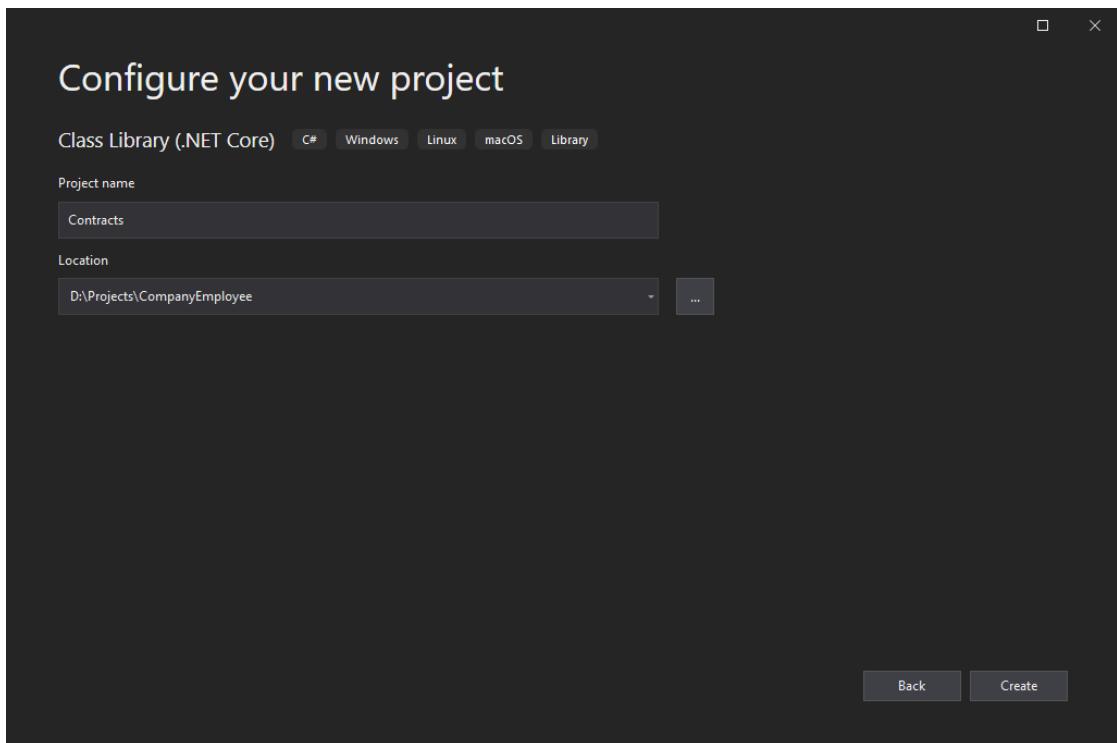
برای اضافه کردن Log به اپلیکیشن به دو پروژه جدید نیاز داریم.

- اولین پروژه را Contracts نامگذاری می‌کنیم چون قرار است اینترفیس‌های ما در این پروژه قرار گیرد. این پروژه قراردادهای کل اپلیکیشن را شامل می‌شود.
- پروژه دوم LoggerService نام دارد که در آن، منطق Logger می‌گیرد.

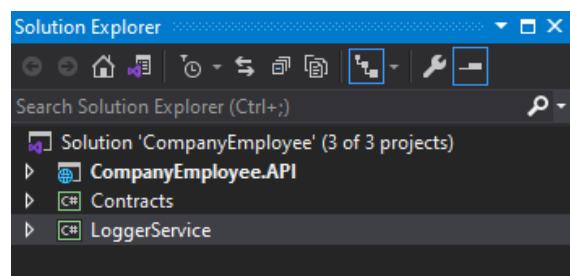
خب برای ایجاد یک پروژه جدید بر روی Solution Explorer راست کلیک و سپس گزینه‌ی Add → New Project را انتخاب کنید. حالا در کادر باز شده Core) Class Library (.NET را انتخاب نمایید.



در کادر بعد، نام آن را Contracts بگذارید و بر روی Create کلیک کنید.



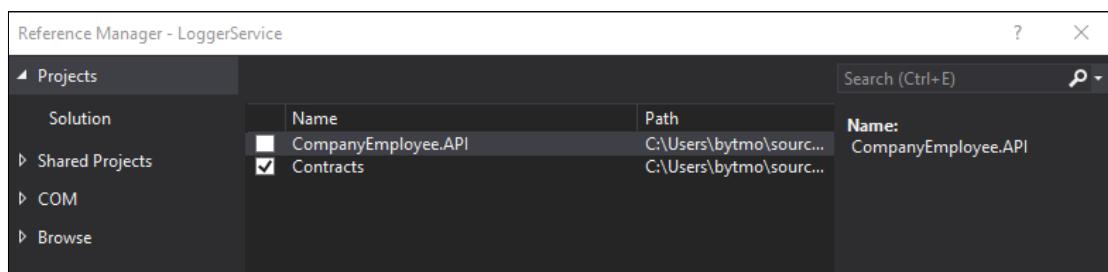
برای پروژه دوم هم همین مسیر را ادامه دهید اما نام آن را LoggerService بگذارید.



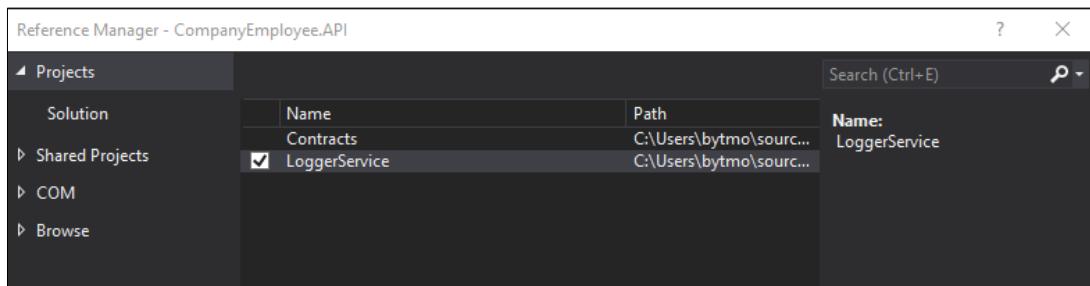
حالا باید پروژه‌ها به هم رفرنس داشته باشند.

برای این کار به Solution Explorer بروید.

- در پروژه LoggerService روی Dependencies راست کلیک و گزینه Add Project را انتخاب نمایید. سپس همانند تصویر پایین، چک باکس Contracts را تیک زده و بر روی OK کلیک کنید.



- در پروژه اصلی بر روی Dependencies راست کلیک و سپس گزینه Add Project را انتخاب و در پایان تیک Reference را بزنید. چون در پروژه Contracts رفنس پروژه Contracts را داریم، پس در اینجا نیازی به اضافه کردن LoggerService رفنس این پروژه نیست.



ایجاد اینترفیس ILogerManager و نصب NLog

در سرویس Logger، به متدهایی برای Log کردن پیام‌های Info و Debug و Warning و Error نیازمندیم. بنابراین باید در پروژه Contracts یک فolder با نام IServices ایجاد و سپس اینترفیسی با نام ILogerManager را در آن اضافه کنید.

```
namespace Contracts.IServices
{
    public interface ILogerManager
    {
        void LogInfo(string message);
        void LogWarn(string message);
        void LogDebug(string message);
        void LogError(string message);
    }
}
```

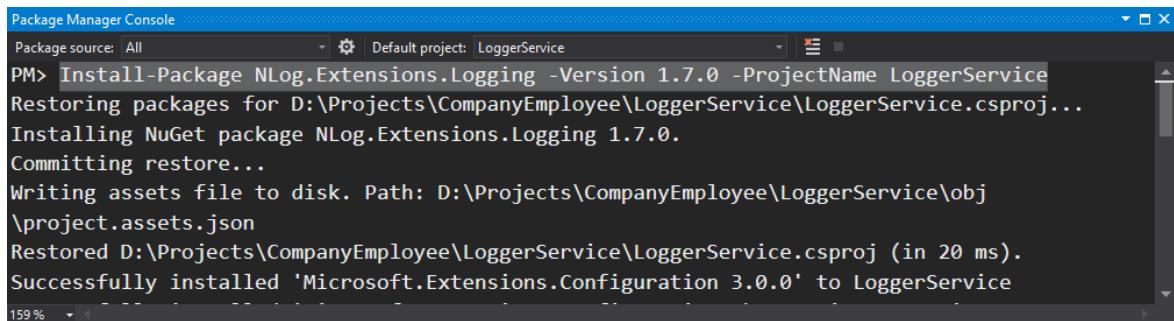
قبل پیاده‌سازی این اینترفیس، باید در پروژه LoggerService پکیج NLog را نصب کنید.

NLog پلتفرم Log برای .NET است که به ما کمک می‌کند تا پیام‌های خود را Log کنیم.

برای اضافه کردن NLog وارد Tools → NuGet Package Manager شوید سپس بر روی Package Manager Console کلیک نمایید.

حالا در اینجا عبارت `Install-Package NLog` را تایپ و در پایان Enter بزنید.

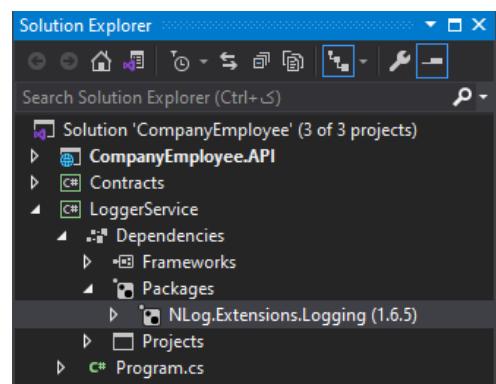
```
Install-Package NLog.Extensions.Logging -Version 1.7.0 -ProjectName  
LoggerService
```



The screenshot shows the Package Manager Console window with the command `Install-Package NLog.Extensions.Logging -Version 1.7.0 -ProjectName LoggerService` being run. The output shows the package is being restored, installed, and successfully added to the project.

```
PM> Install-Package NLog.Extensions.Logging -Version 1.7.0 -ProjectName LoggerService
Restoring packages for D:\Projects\CompanyEmployee\LoggerService\LoggerService.csproj...
Installing NuGet package NLog.Extensions.Logging 1.7.0.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\LoggerService\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\LoggerService\LoggerService.csproj (in 20 ms).
Successfully installed 'Microsoft.Extensions.Configuration 3.0.0' to LoggerService
159 %
```

بعد از چند ثانیه NLog در اپلیکیشن نصب می‌شود.



پیاده‌سازی اینترفیس `ILoggerManager` و پیکربندی فایل `Nlog.Config`

برای پیاده‌سازی اینترفیس `ILoggerManager` باشد در پروژه `LoggerService` یک کلاس با نام `LoggerManager` اضافه کنید.

```
using Contracts.IServices;
using NLog;

namespace LoggerService
{
    public class LoggerManager : ILoggerManager
    {
        private static ILogger logger =
            LogManager.GetCurrentClassLogger();
        public LoggerManager()
        {
        }

        public void LogDebug(string message)
        {
            logger.Debug(message);
        }
    }
}
```

```

public void LogError(string message)
{
    logger.Error(message);
}

public void LogInfo(string message)
{
    logger.Info(message);
}

public void LogWarn(string message)
{
    logger.Warn(message);
}
}

}

```

بررسی کد :

- همانطور که می‌بینید متدهای ما، فقط Wrapper NLog بر روی متدهای NLog می‌باشند. همچنانکه LogManager و ILoger هم دو بخشی از فضای نام NLog هستند.

NLog باید اطلاعاتی در مورد مکان فایل‌های Log، نام این فایل‌ها و حداقل سطحی که می‌خواهیم Log کنیم، را در اختیار داشته باشد. بنابراین در پروژه اصلی باید تمام این قراردادها را در یک فایل متنی به نام nlog.config تعریف کنیم.

خب در ریشه پروژه CompanyEmployee.API یک فایل nlog.config ایجاد و کدهای پایین را در آن اضافه نمایید.

```

<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      autoReload="true"
      internalLogLevel="Trace"
      internalLogFile="d:\Projects\LogsFolder\internal_logs\internallog.txt">
    <targets>
      <target name="logfile" xsi:type="File"

```

```

    fileName="d:\Projects\LogsFolder/logs\${shortdate}_logfile.tx
t"

layout="${longdate} ${level:uppercase=true} ${message}"/>

```

</targets>

<rules>

<logger name="*" minlevel="Debug" writeTo="logfile" />

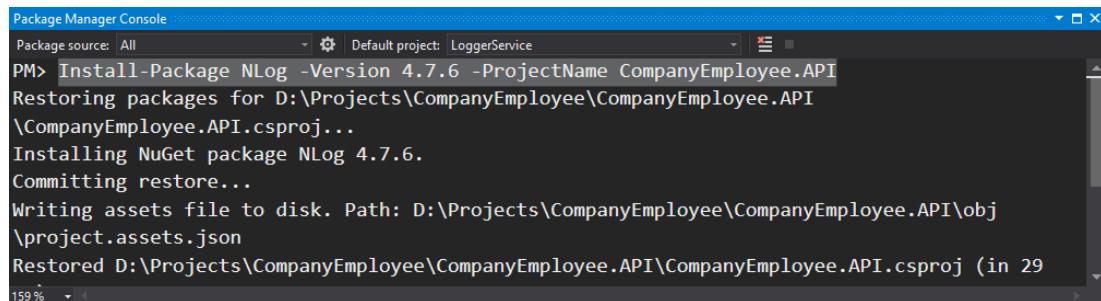
</rules>

</nlog>

پیکربندی Logger Service

پیکربندی سرویس Logger بسیار آسان است. ابتدا باید پکیج NLog را در پروژه اصلی نصب کنید.

`Install-Package NLog -Version 4.7.6 -ProjectName CompanyEmployee.API`



حالا `Namespace` های پایین را به کلاس `Startup` اضافه کنید.

```

using NLog;
using System.IO;

```

در پایان باید `Constructor` کلاس `Startup` را همانند کد پایین تغییر دهید.

```

public Startup(IConfiguration configuration)
{
    LogManager.LoadConfiguration(string.Concat(Directory.GetCurrentDirecto
ry(), "/nlog.config"));
    Configuration = configuration;
}

```

همانطور که در کد بالا می بینید، ما از متده `LoadConfiguration` کلاس `LogManager` استفاده کردیم، تا مسیر پیکربندی فایل `Nlog` را مشخص کنیم.

بعد از این مرحله باید سرویس ConfigureServices را در متده رجیستر کنیم. رجیستر کردن این سرویس می‌تواند به سه حالت انجام شود :

- ۱) از متده services.AddSingleton استفاده کنیم. با این متده، اولین بار که ریکوئستی دریافت کنیم، اپلیکیشن یک Instance از سرویس ایجاد می‌کند و ریکوئست‌های بعدی همه از همین Instance استفاده می‌کنند.
- ۲) از متده services.AddScoped استفاده کنیم. با این متده، هر بار که ریکوئستی دریافت کنیم، اپلیکیشن تنها یک Instance از سرویس بلبت آن ریکوئست ایجاد می‌کند. یعنی اگر در یک ریکوئست چند بار سرویس ما صدا زده شود ما هر بار همان Instance را می‌فرستیم.
- ۳) از متده services.AddTransient استفاده کنیم. این متده بر عکس متده است. یعنی اگر در یک ریکوئست چند بار سرویس ما صدا زده شود ما هم چند Instance از سرویس ایجاد می‌کنیم. خب حالا باید در کلاس ServiceExtensions یک اکستنشن متده جدید برای رجیستر کردن سرویس Logger اضافه کنیم.

```
using Contracts.IServices;
using LoggerService;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions
    {
        public static void ConfigureCors(this IServiceCollection services)
        =>
        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy", builder =>
            builder.AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());
        });
    }
}
```

```

public static void ConfigureServices(IServiceCollection services) =>
    services.Configure<IISSettings>(options =>
    {
        });
}

public static void ConfigureLoggerService(IServiceCollection services) => services.AddScoped<ILoggerManager, LoggerManager>();

}

}

```

الان باید اکستنشن متدها را در متدهای `ConfigureServices` و `ConfigureLoggerService` رجیستر کنیم.

```
services.ConfigureIISIntegration();
```

از این پس هر زمان که بخواهیم از سرویس `Logger` استفاده کنیم باید اینترفیس `ILoggerManager` را به `Constructor` کلاس موردنظر `Inject` کنیم، تا با اجرای اپلیکیشن، این سرویس را `Resolve` و `Log` را قابل دسترسی نماید.

این نوع `Inject` کردن را `Dependency Injection` می‌نامند. `Dependency Injection` به صورت توکار در `.NET Core` وجود دارد.

بیایید کمی بیشتر در مورد این موضوع صحبت کنیم.

DI و IoC چیست؟

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

Dependency Injection تکنیکی است که با استفاده از آن می‌توانیم آبجکت‌ها و وابستگی‌هایشان را از هم جدا کنیم. به بیان دیگر: به جای اینکه هر بار که به آبجکت نیاز است، آن را به صورت صریح در کلاس `new` کنیم، می‌توانیم یک بار از شی `Instance` بسازیم و سپس آن را به کلاس ارسال کنیم.

درک DI بسیار مهم است زیرا ASP.NET Core برای دریافت هر سرویسی از DI استفاده می‌کند.

DI یک Design Pattern است که قابلیت نوشتن کدهای Loosely Coupled را فراهم می‌کند. شاید بپرسید، کدهای Loosely Coupled چیست و به چه دردی می‌خورد؟

در دنیای نرمافزار، تغییرات بخش جدایی ناپذیر در اپلیکیشن‌های ماست. این تغییرات به مرور زمان باعث خراب شدن پایه‌های اپلیکیشن می‌شود. آقای رابت سسیل مارتین معروف به عموماً، 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 مژول نرمافزاری می‌بایست تنها یک دلیل برای تغییر داشته باشد.
- **اصل دوم O - OCP - Open Closed Principle** : مژول‌های نرمافزار باید برای تغییرات بسته و برای توسعه باز باشند.
- **اصل سوم L – LSP – Liskov Substitution Principle** : SubClass بتوانند جایگزین نوع پایه‌ی خود باشند.
- **اصل چهارم I – ISP – Interface Segregation Principle** : کلاینت‌ها نباید وابسته به متدهایی باشند که آنها را پیاده‌سازی نمی‌کنند.
- **اصل پنجم D – DIP – Dependency Inversion principle** : مهم‌ترین اصل که می‌گوید : مژول‌های سطح بالا نباید به مژول‌های سطح پایین وابسته باشند، هر دو باید به ابستركشن وابسته 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 مژول‌ها را کمتر می‌کنند. با کم شدن این وابستگی‌ها، تغییر یک مژول، تمام برنامه را تحت تاثیر قرار نمی‌دهد.

فریمورک ASP.NET Core برای مازولار بودن از بهترین شیوه‌ی مهندسی نرم‌افزار، یعنی اصول SOLID پیروی می‌کند. اصول SOLID یکی از بهترین تجربه‌ها در طول دوران برنامه‌نویسی شی گرا است.

مزایای DI :

• DI به اپلیکیشن این امکان را می‌دهد تا به صورت داینامیک با خودش لینک داشته باشد. DI Provider به این صورت عمل می‌کند که ریکوئست را دریافت و سپس با توجه به این ریکوئست، کلاس موردنظر را ایجاد می‌نماید.

به طور مثال: اگر یک کلاس به کلاس دیگری نیاز داشته باشد، DI آن را آماده می‌کند و دیگر نیازی نیست که در هر کلاس به صورت دستی این کلاس را new کنید و کدهای Tightly Coupled ایجاد نمایید.

• استفاده از DI، اپلیکیشن را Loosely Coupled مفهوم مهمی در برنامه‌نویسی شی گرا 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 ریکوئست، کلاس موردنظر را برگردانیم.

مثال: بخواهید در برنامه تجاری خود یک کارت اعتباری ساختگی به جای کارت اعتباری واقعی داشته باشید و این موضوع، از طریق تنظیمات مدیریت شود.

اصل DI، سیم کشی را به برنامه شما اضافه می‌کند. شما با این اصل می‌توانید پیچیدگی را کمتر کنید.

معمولاً از طریق Constructor، وابستگی خود را درخواست می‌کند. بنابراین نیاز به کلاسی است که مدیریت کلاس‌ها و وابستگی بین آن‌ها را از طریق Constructor فراهم کند. به این کلاس Container یا به اصطلاحی IoC Container گفته می‌شود.

در اصل یک IoC container یک Instance درخواست شده را فراهم می‌کند.

تست سرویس با Postman

وقت آن رسیده که این سرویس را با هم تست کنیم. بنابراین WeatherForecastController را باز و تغییرات پایین را به آن اعمال کنید.

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

namespace CompanyEmployee.API.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class WeatherForecastController : ControllerBase
    {
        private ILoggerManager _logger;
        public WeatherForecastController(ILoggerManager logger)
        {
            _logger = logger;
        }

        [HttpGet]
        public IEnumerable<string> Get()
        {
            _logger.LogInfo("Here is info message from our values controller.");
            _logger.LogDebug("Here is debug message from our values controller.");
            _logger.LogWarning("Here is warn message from our values controller.");
            _logger.LogError("Here is an error message from our values controller.");
        }
    }
}
```

```

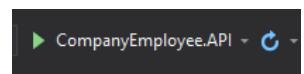
        return new string[] { "value1", "value2" };
    }
}

}

```

برای بررسی نتیجه، می‌خواهیم از یک ابزار فوق العاده به نام Postman استفاده کنیم. این ابزار در ارسال ریکوئست و نمایش Response‌ها به ما خیلی کمک می‌کند.

حالا اپلیکیشن را اجرا و در Postman آدرس پایین را وارد کنید.



<https://localhost:5001/weatherforecast>

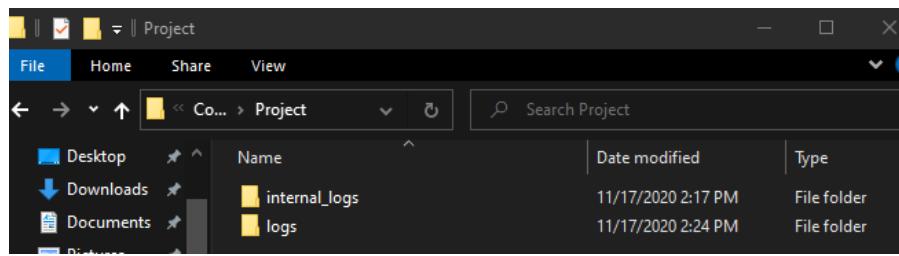
KEY	VALUE	DESCRIPTION
Key	Value	Description

```

1 [ 
2   "value1",
3   "value2"
]

```

بعد از اجرا، به مسیری که برای فایل nlog.config مشخص کردید بروید. در این فolder دو فایل logfile و internal_logs وجود دارد.



اگر فایل‌ها را باز کنید نتیجه پایین را می‌بینید.

```
2020-11-17 14:24:37.1767 INFO Here is info message from our values controller.  
2020-11-17 14:24:37.2179 DEBUG Here is debug message from our values controller.  
2020-11-17 14:24:37.2179 WARN Here is warn message from our values controller.  
2020-11-17 14:24:37.2361 ERROR Here is an error message from our values controller.
```

این تمام کاری بود که ما برای پیکربندی Logger باید انجام می‌دادیم.

فصل سوم : دیتابیس و Repository Pattern

آنچه خواهید آموخت:

- ارتباط با دیتابیس
- پیاده‌سازی **Repository Pattern** در اپلیکیشن
- **Seed Data** چیست؟

مدل دیتابیس و Repository Pattern

در این فصل می‌خواهیم نحوه ایجاد دیتابیس با رویکرد Code First، کار کردن با DbContext، Repository استفاده از Migration (برای انتقال مدل ایجاد شده به دیتابیس) و پیاده‌سازی Repository را بیاموزیم.

واسطی بین Data Access و لایه‌ی Domain Repository است. به عبارتی می‌توان گفت: پترنی است که همه‌ی ارتباطات با دیتابیس را کپسوله می‌کند و باعث می‌شود:

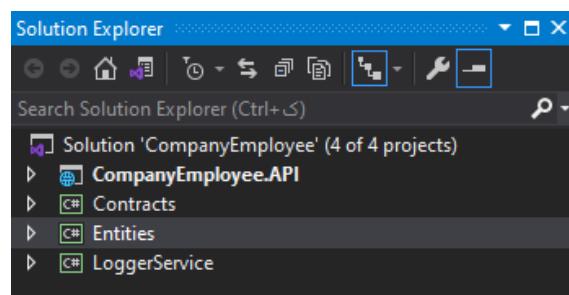
- دسترسی به داده‌ها متمرکز شود.
- نگهداری کد ساده‌تر شود.
- دسترسی به دیتابیس از Domain Model جدا شود.

در این پترن می‌توان رویکرد Loosely Coupled را جهت دسترسی به داده‌های دیتابیس اجرا کرد.

برای پیاده‌سازی این پترن، بهتر است (براساس اصل Separated Interface Pattern) اینترفیس Repository را از هم جدا کنیم تا کلاینت وابسته پیاده‌سازی نشود. خب بیایید ابتدا با کلاس مدل شروع کنیم.

ایجاد مدل‌ها

می‌خواهیم مثال دوم فصل را کامل کنیم بنابراین یک پروژه Class Library (.NET Core) جدید با نام Entities ایجاد کنید.



فراموش نکنید که در پروژه اصلی باید از این پروژه رفرنس بگیرید.

در این پروژه فolderی با نام Models ایجاد کنید تا Entity‌ها را درون آن قرار دهیم.

Entity Framework Core از آن‌ها برای Map کردن با جداول دیتابیس استفاده می‌کند. Entity‌های درون Property، با ستون‌های دیتابیس Map می‌شوند.

در فولدر Models دو کلاس با نام‌های **Company** و **Employee** ایجاد کنید.

کلاس Employee :

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

namespace Entities.Models
{
    public class Employee
    {
        [Column("EmployeeId")]
        public Guid Id { get; set; }

        [Required(ErrorMessage = "Employee name is a required field.")]
        [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30
        characters.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Age is a required field.")]
        public int Age { get; set; }

        [Required(ErrorMessage = "Position is a required field.")]
        [MaxLength(20, ErrorMessage = "Maximum length for the Position is
        20 characters.")]
        public string Position { get; set; }

        [ForeignKey(nameof(Company))]
        public Guid CompanyId { get; set; }

        public Company Company { get; set; }
    }
}
```

```
 }  
}
```

: Company کلاس

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;  
  
namespace Entities.Models  
{  
    public class Company  
    {  
        [Column("CompanyId")]  
        public Guid Id { get; set; }  
  
        [Required(ErrorMessage = "Company name is a required field." )]  
        [MaxLength(60, ErrorMessage = "Maximum length for the Name is 60  
        characters." )]  
        public string Name { get; set; }  
  
        [Required(ErrorMessage = "Company address is a required field." )]  
        [MaxLength(60, ErrorMessage = "Maximum length for the Address is  
        60 character" )]  
        public string Address { get; set; }  
  
        public string Country { get; set; }  
  
        public ICollection<Employee> Employees { get; set; }  
    }  
}
```

بررسی کد :

- ما دو کلاس Company و Employee ایجاد کردیم. پرایپریتی‌های این کلاس‌ها را برای Map کردن با ستون‌های جداول دیتابیس استفاده می‌کند.
 - آخرین پرایپری کلاس Company (یعنی Employees) و آخرین پرایپری کلاس Navigational Properties Company (یعنی Employee) هستند که هدفشان برقراری ارتباط بین مدل‌هاست.
 - همانطور که می‌بینید، چندین Attribute در Entity‌ها اضافه شده است.
- [Column] مشخص می‌کند که Property با یک نام دیگر در دیتابیس Map شود.
- [Required] ورود اطلاعات برای پرایپری را اجباری می‌کند.
- [MaxLength] حداکثر کاراکتری که پرایپری باید داشته باشد را مشخص می‌کند.

بعد از اینکه این مدل به دیتابیس انتقال یافت خواهیم دید که چطور این Attribute و ستون‌ها روی ناویگیشن Property تاثیر می‌گذارند.

ایجاد کلاس Context

برای ارتباط با دیتابیس و استفاده از Entity Framework Core باید یک کلاس ایجاد کنیم که از کلاس DbContext ارث بری کند. EF Core قلب DbContext است.

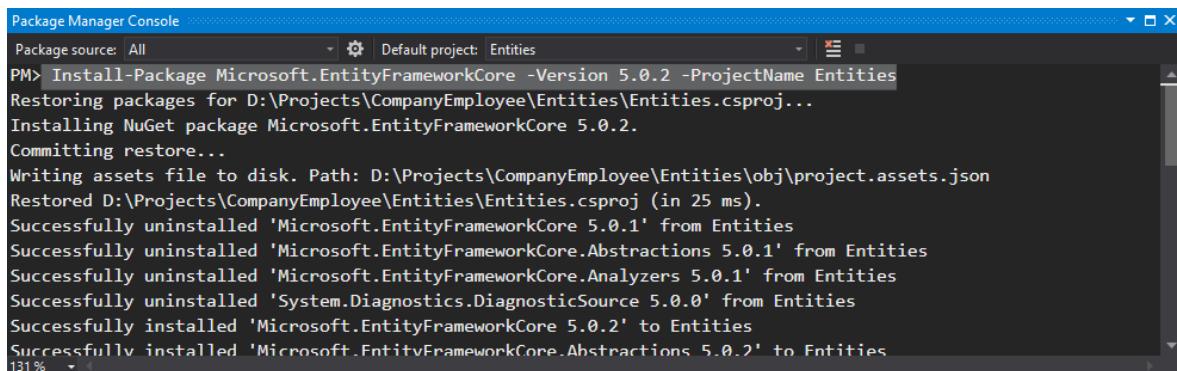
در کلاس‌های شما به صورت DbSet<T> تعریف می‌شود. T درواقع Entity می‌باشد. (Company یا Employee).

هر DbSet به یک جدول در دیتابیس Map خواهد شد تا EF Core بتواند با جداول دیتابیس ارتباط برقرار کند. پس شما باید بابت هر DbSet یک Entity داشته باشید.

برای ارث بری از کلاس DbContext، باید قبل از هر کاری پکیج Microsoft.EntityFrameworkCore را در پروژه Entities اضافه کنید.

برای انجام این کار وارد **Manger Package** Tools → NuGet Package Manager → Package مسیر شوید و دستور پایین را اجرا کنید.

```
Install-Package Microsoft.EntityFrameworkCore -Version 5.0.2 -ProjectName Entities
```



```
Package Manager Console
Package source: All Default project: Entities
PM> Install-Package Microsoft.EntityFrameworkCore -Version 5.0.2 -ProjectName Entities
Restoring packages for D:\Projects\CompanyEmployee\Entities\Entities.csproj...
Installing NuGet package Microsoft.EntityFrameworkCore 5.0.2.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\Entities\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\Entities\Entities.csproj (in 25 ms).
Successfully uninstalled 'Microsoft.EntityFrameworkCore 5.0.1' from Entities
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Abstractions 5.0.1' from Entities
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Analyzers 5.0.1' from Entities
Successfully uninstalled 'System.Diagnostics.DiagnosticSource 5.0.0' from Entities
Successfully installed 'Microsoft.EntityFrameworkCore 5.0.2' to Entities
Successfully installed 'Microsoft.EntityFrameworkCore.Abstractions 5.0.2' to Entities
131%
```

بعد از نصب این پکیج باید در ریشه پروژه Entities یک کلاس با نام CompanyEmployeeDbContext ایجاد کنید.

```
using Entities.Models;
using Microsoft.EntityFrameworkCore;

namespace Entities
{
    public class CompanyEmployeeDbContext : DbContext
    {
        public CompanyEmployeeDbContext(DbContextOptions options):
            base(options)
        {

        }

        public DbSet<Company> Companies { get; set; }
        public DbSet<Employee> Employees { get; set; }
    }
}
```

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 خواهد شد اما مکان دیتابیس باید روی سیستم شما یا هر جایی در سرور دیتابیس قرار گیرد.

برای مثال:

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

آن را درون فایل appsettings.json قرار دهیم تا بتوانیم بدون کامپایل مجدد، محل دیتابیس را در کامپیوترهای مختلف مشخص کنیم.

خب حالا بباید برای ارتباط با دیتابیس Connection String خود را مشخص کنیم. پس به پروژه اصلی بروید و فایل appsettings.Json را مانند فایل زیر تغییر دهید.

```
{
  "ConnectionStrings": {
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated
                     Security=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

رجیستر DbContext از طریق DI

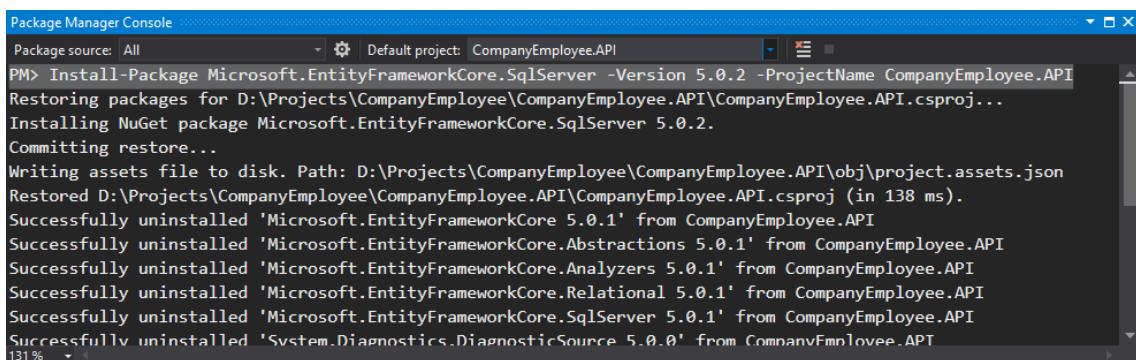
برای تکمیل پیکر بندی دیتابیس، یک مرحله دیگر به نام رجیستر کردن DbContext وجود دارد که باید انجام شود. اما قبل از رجیستر شدن این سرویس، باید نوع دیتابیس خود را مشخص کنید.

EF Core از طیف وسیعی از دیتابیس‌ها پشتیبانی می‌کند، شما می‌توانید با توجه به مهارت خود یکی را انتخاب کنید:

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

من در این کتاب می‌خواهم از SQL Server استفاده کنم پس پکیج `Microsoft.EntityFrameworkCore.SqlServer` را با دستور زیر نصب می‌کنم.

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 5.0.2 -  
ProjectName CompanyEmployee.API
```



The screenshot shows the Package Manager Console window with the following output:

```
Package Manager Console  
Default project: CompanyEmployee.API  
PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 5.0.2 -ProjectName CompanyEmployee.API  
Restoring packages for D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj...  
Installing NuGet package Microsoft.EntityFrameworkCore.SqlServer 5.0.2.  
Committing restore...  
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\project.assets.json  
Restored D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj (in 138 ms).  
Successfully uninstalled 'Microsoft.EntityFrameworkCore 5.0.1' from CompanyEmployee.API  
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Abstractions 5.0.1' from CompanyEmployee.API  
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Analyzers 5.0.1' from CompanyEmployee.API  
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Relational 5.0.1' from CompanyEmployee.API  
Successfully uninstalled 'Microsoft.EntityFrameworkCore.SqlServer 5.0.1' from CompanyEmployee.API  
Successfully uninstalled 'System.Diagnostics.DiagnosticSource 5.0.0' from CompanyEmployee.API  
131 %
```

خب حالا باید در کلاس `Startup`، کلاس `CompanyEmployeeDbContext` را همانند کلاس `LoggerManager` رجیستر کنیم. بنابراین کلاس `ServiceExtensions` را باز کنید و اکستنشن `ConfigureSqlContext` را به آن اضافه کنید.

```
using Contracts.IServices;  
using Entities;  
using LoggerService;  
using Microsoft.AspNetCore.Builder;  
using Microsoft.EntityFrameworkCore;
```

```

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions
    {
        public static void ConfigureCors(this IServiceCollection services)
        =>
        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy", builder =>
            builder.AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());
        });

        public static void ConfigureIISIntegration(this
IServiceCollection services) =>
        services.Configure<IISSettings>(options =>
        {
        });

        public static void ConfigureLoggerService(this IServiceCollection
services) => services.AddScoped<ILoggerManager, LoggerManager>();

        public static void ConfigureDbContext(this IServiceCollection
services, IConfiguration configuration) =>
        services.AddDbContext<CompanyEmployeeDbContext>(opts =>
        opts.UseSqlServer(configuration.GetConnectionString("sqlConnectio
n")));
    }
}

```

بررسی کد :

- در کد بالا با استفاده از متده `GetConnectionString` توانستیم مقدار `Connection` را از فایل `appsettings.json` بخوانیم.

- این متدهای string از ورودی میگیرد و این رشته را با Keyهای درون فایل appsettings.json مقایسه و در نهایت Value موردنظر را برمیگرداند.
- اگر روی این متدهای F12 بزنید، به تعریف این اکستنشن متدهای روید و کد پایین را خواهید دید.

```
// Summary:
//     Shorthand for GetSection("ConnectionStrings")[name]
//
// Parameters:
//     configuration:
//         The configuration.
//
//     name:
//         The connection string key.
//
// Returns:
//     The connection string.
public static string GetConnectionString(this IConfiguration configuration, string name)
{
    throw null;
}
```

- همانطور که در کد بالا میبینید، نام Connection String به عنوان Key به متدهای Connection String داده شده و یک Connection String را با استفاده از این نام برگرداند.
- پس از انجام مراحل بالا باید این اکستنشن متدهای درون ConfigureServices رجیستر کنیم.

`services.ConfigureSqlContext(Configuration);`

Migration چیست؟

- بعد از انجام مراحل بالا، نوبت به ایجاد دیتابیس میرسد. اما دیتابیس چطور ایجاد میشود؟
- یک روش خوب برای ایجاد دیتابیس، ودار کردن EF به ساخت دیتابیس است. ساده‌ترین روش EF برای انجام این کار، استفاده از Migration است.
- راهنمایی برای مدیریت جداول در دیتابیس میباشد. با Migration میتوانید بدون هیچ‌گونه دردسری، تغییرات را به جداول دیتابیس اعمال نمایید.
- قبل از ایجاد Migration دو کار باید انجام دهیم :

- چون مدل دیتابیس را در پروژه‌ی Entities ایجاد کردید و میخواهیم فolder Migration در پروژه اصلی باشد، پس باید متدهای ConfigureServices را کمی تغییر دهید.

```

public static void ConfigureServices(IServiceCollection services,
IConfiguration configuration) =>
services.AddDbContext<CompanyEmployeeDbContext>(opts =>
opts.UseSqlServer(configuration.GetConnectionString("sqlConnection"), b =>
b.MigrationsAssembly("CompanyEmployee.API")));

```

- قبل از اجرای دستور Migration باید پکیج پایین را نصب کنید.

**Install-Package Microsoft.EntityFrameworkCore.Tools -Version 5.0.2 -
ProjectName CompanyEmployee.API**

```

Package Manager Console
Package source: All | Default project: CompanyEmployee.API
PM> Install-Package Microsoft.EntityFrameworkCore.Tools -Version 5.0.2 -ProjectName CompanyEmployee.API
Restoring packages for D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj...
Installing NuGet package Microsoft.EntityFrameworkCore.Tools 5.0.2.
Committing restore...
Generating MSBuild file D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\CompanyEmployee.API.csproj.nuget.g.props.
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\project.assets.json.
Restored D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj (in 129 ms).
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Design 5.0.1' from CompanyEmployee.API.
Successfully uninstalled 'Microsoft.EntityFrameworkCore.Tools 5.0.1' from CompanyEmployee.API.
Successfully installed 'Microsoft.EntityFrameworkCore.Design 5.0.2' to CompanyEmployee.API.
Successfully installed 'Microsoft.EntityFrameworkCore.Tools 5.0.2' to CompanyEmployee.API.
131 %

```

خب حالا برای ایجاد Migration و تولید ساختار دیتابیس باید دستور Add-Migration را در Package Manager Console اجرا کنید.

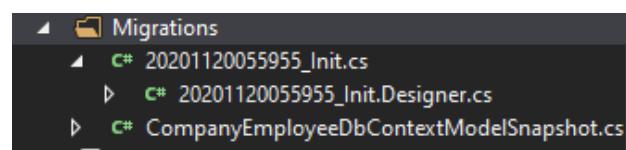
Add-Migration Init

```

Package Manager Console
Package source: All | Default project: CompanyEmployee.API
PM> Add-Migration Init
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>

```

با اجرای این دستور، یک فolder به نام Migrations به Solution اضافه شده است. در این فolder کلاسی وجود دارد که کد ایجاد دیتابیس و جداول های ما در آن قرار گرفته است.



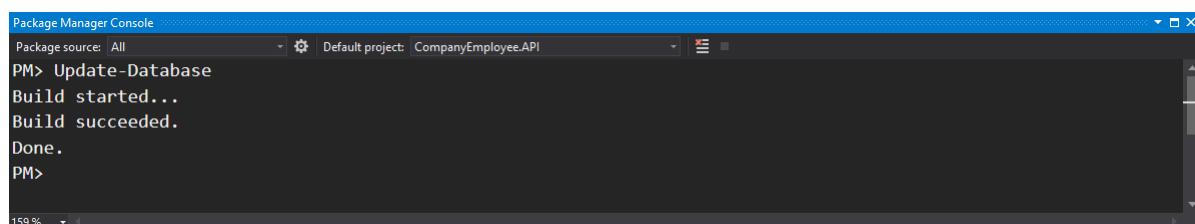
با این فایل ها می توانیم دیتابیس خود را ایجاد کنیم. سه روش برای اعمال کدهای Migration به دیتابیس وجود دارد:

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 Update دیتابیس استفاده کنید.

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

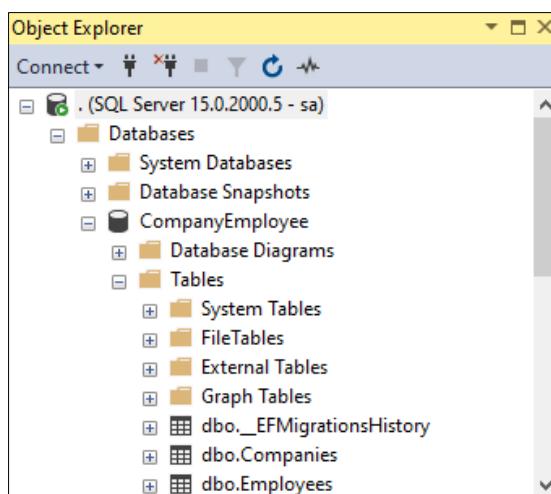


```
Package Manager Console
Package source: All
PM> Update-Database
Build started...
Build succeeded.
Done.
PM>
```

نکته!!

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

خب حالا ببینیم دیتابیس را با هم ببینیم.



Seed Data چیست؟

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

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

برای اعمال قابلیت Seed Data به اپلیکیشن :

مرحله اول : در پروژه Entities فolderی با نام Configuration ایجاد و در آن دو کلاس با نام‌های EmployeeConfiguration و CompanyConfiguration اضافه نمایید.

کلاس CompanyConfiguration :

```
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;

namespace Entities.Configuration
{
    public class CompanyConfiguration : IEntityTypeConfiguration<Company>
    {
        public void Configure(EntityTypeBuilder<Company> builder)
        {
            builder.HasData
            (
                new Company
                {
                    Id = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870"),
                    Name = "Raveshmand_Ltd",
                    Address = "Tehran,Tajrish",
                    Country = "Iran"
                },
                new Company
                {
                    Id = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3"),
                    Name = "Geeks_Ltd",
                    Address = "London",
                    Country = "English"
                }
            );
        }
    }
}
```

کلاس EmployeeConfiguration

```
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;

namespace Entities.Configuration
{
    public class EmployeeConfiguration : IEntityTypeConfiguration<Employee>
    {
        public void Configure(EntityTypeBuilder<Employee> builder)
        {
            builder.HasData
            (
                new Employee
                {
                    Id = new Guid("80abbca8-664d-4b20-b5de-024705497d4a"),
                    Name = "Zahra Bayat",
                    Age = 26,
                    Position = "Backend developer",
                    CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
                },
                new Employee
                {
                    Id = new Guid("86dba8c0-d178-41e7-938c-ed49778fb52a"),
                    Name = "Ali Bayat",
                    Age = 30,
                    Position = "Backend developer",
                    CompanyId = new Guid("c9d4c053-49b6-410c-bc78-2d54a9991870")
                },
                new Employee
                {
                    Id = new Guid("021ca3c1-0deb-4af8-ae94-2159a8479811"),
                    Name = "Sara Bayat",
                    Age = 35,
                    Position = "Frontend developer",
                    CompanyId = new Guid("3d490a70-94ce-4d15-9494-5248280c2ce3")
                }
            );
        }
    }
}
```

```
}
```

مرحله دوم : برای فراخوانی این پیکربندی، باید کلاس CompanyEmployeeDbContext را تغییر دهیم.

```
using Entities.Configuration;
using Entities.Models;
using Microsoft.EntityFrameworkCore;

namespace Entities
{
    public class CompanyEmployeeDbContext : DbContext
    {

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

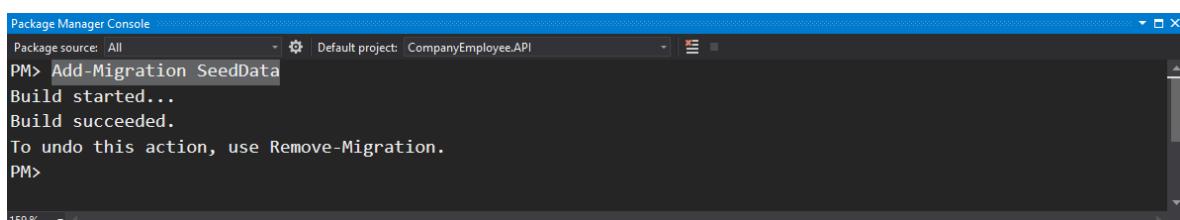
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.ApplyConfiguration(new CompanyConfiguration());
            modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
        }

        public DbSet<Company> Companies { get; set; }
        public DbSet<Employee> Employees { get; set; }

    }
}
```

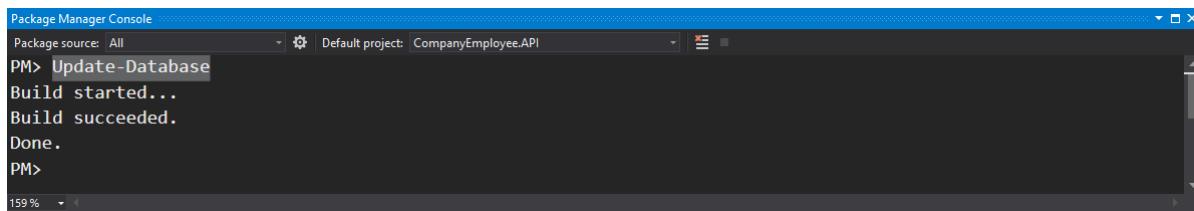
مرحله سوم : برای Seed کردن دیتا در دیتابیس یک Migration ایجاد کنید.

Add-Migration SeedData



```
Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Add-Migration SeedData
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

Update-Database



```
Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Update-Database
Build started...
Build succeeded.
Done.
PM>
159 %
```

با این کار تمام داده‌ها از فایل‌های پیکربندی به جداول انتقال یافت.

پیاده‌سازی Repository Pattern

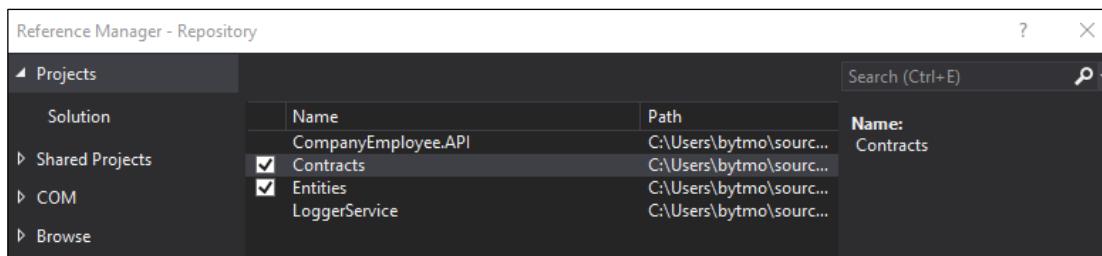
پس از برقراری ارتباط با دیتابیس، زمان آن فرارسیده تا برای داشتن متدهای CRUD یک Repository جنریک ایجاد کنیم.

- ابتدا در مسیر Contracts / IServices یک اینترفیس به نام IRepositoryBase<T> ایجاد کنید.

```
using System;
using System.Linq;
using System.Linq.Expressions;

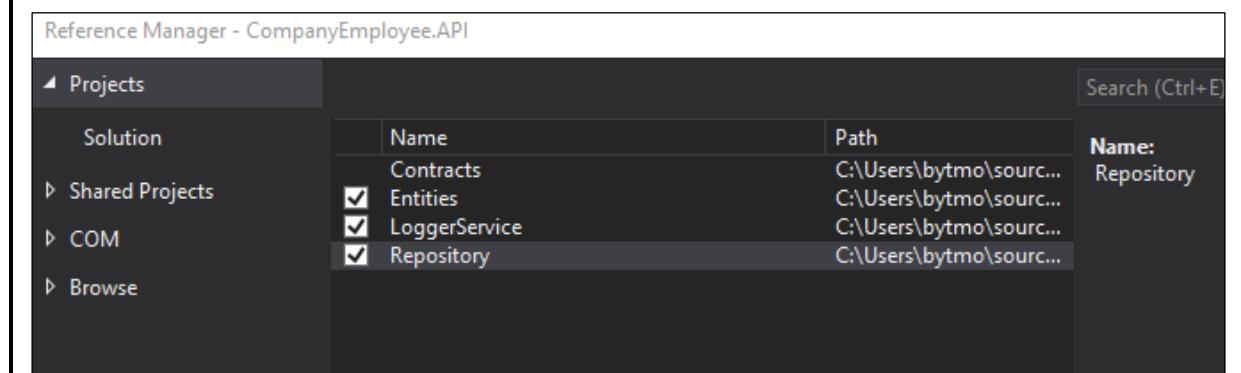
namespace Contracts.IServices
{
    public interface IRepositoryBase<T> where T : class
    {
        IQueryable<T> FindAll(bool trackChanges);
        IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression,
                                     bool trackChanges);
        void Create(T entity);
        void Update(T entity);
        void Delete(T entity);
    }
}
```

- بعد از ایجاد این اینترفیس، نیاز به یک پروژه جدید با نام Repository داریم. این پروژه Class Entities و Contracts رفنس داشته باشد. پس یک پروژه از نوع Library (.NET Core) ایجاد کنید.



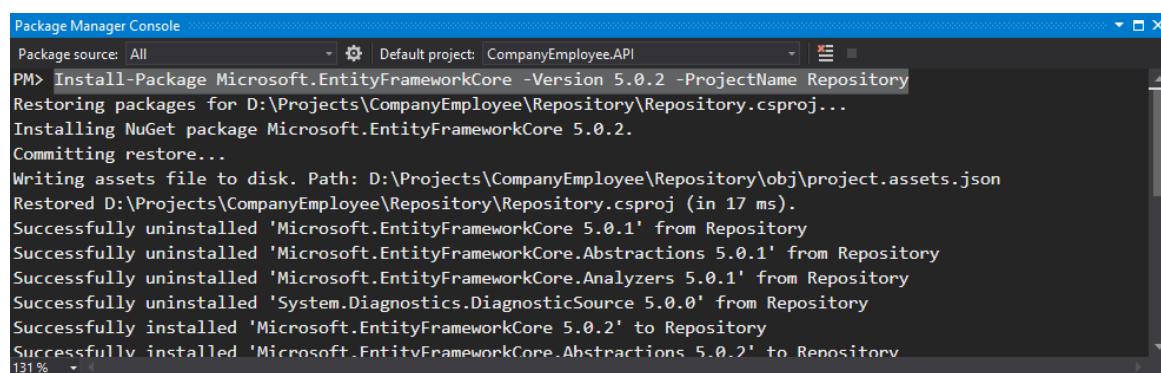
نکته!!

فراموش نکنید که پروژه اصلی باید از این پروژه رفرنس بگیرد.



- برای پیادهسازی IRepositoryBase، یک فolder با نام Repository در پروژه RepositoryBase ایجاد کنید. در این فolder باید یک کلاس Abstract Repository داشته باشید؛ اما قبل از ایجاد این کلاس باید پکیج EF Core را اضافه نمایید.

**Install-Package Microsoft.EntityFrameworkCore -Version 5.0.2 -
ProjectName Repository**



: کدهای کلاس RepositoryBase

```
using Contracts.IServices;
using Entities;
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;
```

```

using System.Linq.Expressions;

namespace Repository.Repositories
{
    public abstract class RepositoryBase<T> : IRepositoryBase<T> where T
        : class
    {
        protected CompanyEmployeeDbContext _companyEmployeeDbContext;
        public RepositoryBase(CompanyEmployeeDbContext
            companyEmployeeDbContext)
        {
            _companyEmployeeDbContext = companyEmployeeDbContext;
        }
        public IQueryable<T> FindAll(bool trackChanges) =>
        !trackChanges ?
        _companyEmployeeDbContext.Set<T>()
        .AsNoTracking() :
        _companyEmployeeDbContext.Set<T>();

        public IQueryable<T> FindByCondition(Expression<Func<T, bool>>
        expression,
        bool trackChanges) =>
        !trackChanges ?
        _companyEmployeeDbContext.Set<T>()
        .Where(expression)
        .AsNoTracking() :
        _companyEmployeeDbContext.Set<T>()
        .Where(expression);

        public void Create(T entity) =>
        _companyEmployeeDbContext.Set<T>().Add(entity);

        public void Update(T entity) =>
        _companyEmployeeDbContext.Set<T>().Update(entity);

        public void Delete(T entity) =>
        _companyEmployeeDbContext.Set<T>().Remove(entity);
    }
}

```

}

بررسی کد :

- کلاس RepositoryBase و اینترفیس IRepositoryBase با نوع جنریک T کار می‌کنند. جنریک به ما امکان می‌دهد تا کلاس‌هایی تعریف کنیم که نوع فیلدها، پارامترها و متدهایشان در زمان استفاده تعیین شوند.
- در کد بالا پارامتر trackChanges را می‌بینید. از این پارامتر برای بالا رفتن کوئری‌ها استفاده کردیم. وقتی که ما AsNoTracking استفاده کنیم، این پارامتر بر روی false تنظیم می‌شود و کوئری به EF Core اطلاع می‌دهد که نیاز به ردیابی تغییرات Entity‌ها نیست. این باعث افزایش سرعت یک کوئری می‌شود.

پیاده‌سازی کلاس‌های Repository

حالا نوبت ایجاد کلاس‌هایی است که می‌خواهند از کلاس RepositoryBase ارث بری کنند. قبل از ایجاد این کلاس‌ها باید بدانیم، که هر کلاس می‌تواند علاوه بر ارث بری از کلاس RepositoryBase، یک اینترفیس مخصوص به خود نیز داشته باشد. بنابراین ما باید منطقی را که برای همه ریپازیتوری‌هایمان یکی است را از ریپازیتوری کلاس‌های کاربر جدا کنیم. خب با این اوصاف در پروژه Contracts (فولدر IServices) دو اینترفیس با نام‌های IEmployeeRepository و ICompanyRepository ایجاد کنید.

: ICompanyRepository اینترفیس

```
namespace Contracts.IServices
{
    public interface ICompanyRepository
    {
    }
}
```

: IEmployeeRepository اینترفیس

```
namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
    }
}
```

```
}
```

حالا نوبت ایجاد کلاس‌های Repository در فolder Repository است.

کدهای کلاس **CompanyRepository**

```
using Contracts.IServices;
using Entities;
using Entities.Models;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext): base(companyEmployeeDbContext)
        {
        }
    }
}
```

کدهای کلاس **EmployeeRepository**

```
using Contracts.IServices;
using Entities;
using Entities.Models;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
        IEmployeeRepository
    {
        public EmployeeRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext): base(companyEmployeeDbContext)
        {
        }
    }
}
```

}

پس از این مرحله، ایجاد Repository تکمیل می‌شود اما هنوز کارهای دیگری هم هست که در ادامه با هم انجام می‌دهیم.

ایجاد یک Repository Manager

همانطور که می‌دانید خروجی برخی از API‌ها در اپلیکیشن، شامل ترکیب داده‌های چندین است. برای مثال Resource :

API که خروجی آن تمام شرکت‌هایی است که کارمندان بالای ۳۰ سال دارند.

در چنین حالتی باید از هر دو ریپازیتوری Instance داشته باشیم و داده‌های خود را از Resource‌ها بدست آوریم.

شاید با دو کلاس، مشکل چندانی با این مسئله وجود نداشته باشد اما اگر منطق ما نیاز به ترکیب ۵ یا ۶ کلاس داشته باشد مطمئناً این موضوع خیلی پیچیده می‌شود.

راحل این مشکل، یک کلاس RepositoryManager است که درون آن از تمام کلاس‌های Instance یک Repository وجود داشته باشد.

این کلاس دو قابلیت به اپلیکیشن ما اضافه می‌کند :

۱. به هر Repository که نیاز داشته باشیم به راحتی دسترسی داریم.
۲. ما متدهای Create, Update و Delete را در RepositoryBase داشتیم اما تا زمانی که متدهای SaveChanges صدای نشود این متدها هیچ کاری در دیتابیس انجام نمی‌دهند. کلاس RepositoryManager این کار را به خوبی هندل می‌کند.

خب برای اینکه به این قابلیت برسیم باید اینترفیس جدیدی با نام IRepositoryManager در پروژه فولدر (Services) Contract ایجاد کنیم.

```
namespace Contracts.IServices
{
    public interface IRepositoryManager
    {
        ICompanyRepository Company { get; }
        IEmployeeRepository Employee { get; }
    }
}
```

```
        void Save();  
    }  
}
```

حالا در پروژه RepositoryManager (فولدر Repositories) یک کلاس جدید با نام Repository ایجاد کنید.

```
using Contracts.IServices;  
using Entities;  
  
namespace Repository.Repositories  
{  
    public class RepositoryManager : IRepositoryManager  
    {  
        private CompanyEmployeeDbContext _repositoryContext;  
        private ICompanyRepository _companyRepository;  
        private IEmployeeRepository _employeeRepository;  
  
        public RepositoryManager(CompanyEmployeeDbContext repositoryContext)  
        {  
            _repositoryContext = repositoryContext;  
        }  
        public ICompanyRepository Company  
        {  
            get  
            {  
                if (_companyRepository == null)  
                    _companyRepository = new  
                        CompanyRepository(_repositoryContext);  
  
                return _companyRepository;  
            }  
        }  
  
        public IEmployeeRepository Employee  
        {  
            get  
            {  
                if (_employeeRepository == null)
```

```

        _employeeRepository = new
            EmployeeRepository(_repositoryContext);
        return _employeeRepository;
    }

    public void Save() => _repositoryContext.SaveChanges();
}

}

```

بررسی کد :

- همانطور که می‌بیند ما Repository‌هایی ایجاد کردیم که ما را در معرض دید قرار می‌دهند.
- همچنین برای اینکه بتوانیم بعد از اتمام تغییرات در یک شی، عمل ذخیره سازی را داشته باشیم؛ متدهای Save را در اینجا قرار دادیم.

این روش خوبی است زیرا می‌توانیم چند کار را در یک اکشن انجام دهیم. به عنوان مثال:

دو شرکت اضافه کنیم سپس دو کارمند تغییر دهیم و یک شرکت حذف کنیم و در پلیان با یک Save تمام تغییرات اعمال شود. در این صورت یا همه تغییرات انجام می‌شود یا هیچ تغییری اعمال نمی‌شود.

بعد از این تغییرات باید این کلاس رجیستر شود تا بتوان این کلاس را در کنترلرهایمان(یا داخل کلاس‌هایی که منطق بیزینس ما را تشکیل می‌دهند) Inject کنیم.

خب برای رجیستر شدن این کلاس، متدهای پایین را در کلاس ServiceExtensions اضافه کنید.

```

using Contracts.IServices;
using Entities;
using LoggerService;
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

```

```

using Repository.Repositories;

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ServiceExtensions
    {
        public static void ConfigureCors(this IServiceCollection
            services) =>
        services.AddCors(options =>
        {
            options.AddPolicy("CorsPolicy", builder =>
                builder.AllowAnyOrigin()
                    .AllowAnyMethod()
                    .AllowAnyHeader());
        });

        public static void ConfigureIISIntegration(this IServiceCollection
            services) =>
        services.Configure<IISSettings>(options =>
        {
        });

        public static void ConfigureLoggerService(this IServiceCollection
            services) => services.AddScoped<ILoggerManager, LoggerManager>();

        public static void ConfigureDbContext(this IServiceCollection
            services, IConfiguration configuration) =>
        services.AddDbContext<CompanyEmployeeDbContext>(opts =>
            opts.UseSqlServer(configuration.GetConnectionString("sqlConnectio
                n")));
    }

    public static void ConfigureRepositoryManager(this
        IServiceCollection services) =>
        services.AddScoped< IRepositoryManager, RepositoryManager>();
    }
}

```

و در متده `ConfigureServices` کلاس `Startup` در بالای `services.AddController` خط کد پایین را اضافه کنید.

```
services.ConfigureRepositoryManager();
```

حالا برای استفاده از این Repository، تمام کاری که باید انجام دهیم این است که سرویس Inject را در کنترلر RepositoryManager کنیم.

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

namespace CompanyEmployee.API.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class WeatherForecastController : ControllerBase
    {
        private ILoggerManager _logger;
        private readonly IRepositoryManager _repository;

        public WeatherForecastController(ILoggerManager logger, IRepositoryManager repository)
        {
            _logger = logger;
            _repository = repository;
        }

        [HttpGet]
        public IEnumerable<string> Get()
        {
            // _repository.Company.AnyMethodFromCompanyRepository();
            // _repository.Employee.AnyMethodFromEmployeeRepository();

            _logger.LogInfo("Here is info message from our values controller.");
            _logger.LogDebug("Here is debug message from our values controller.");
            _logger.LogWarning("Here is warn message from our values controller.");
            _logger.LogError("Here is an error message from our values controller.");
            return new string[] { "value1", "value2" };
        }
    }
}
```

}

بررسی کد :

- همانطور که می‌بینید ما Repository را در کنترلر Inject کردیم. این روش در اپلیکیشن‌های کوچک خوب است اما برای اپلیکیشن‌هایی با مقیاس بزرگتر باید یک لایه بیزینس بین کنترلرها و منطق ریپازیتوری ایجاد کنیم و سرویس RepositoryManager درون لایه بیزنس Inject شود. با این کار کنترلر سبک و آزاد از منطق ریپازیتوری می‌شود و همچنین کد ما قابل نگهداری و قابل استفاده مجدد خواهد شد.

فصل چهارم : مسیریابی و REST

آنچه خواهید آموخت:

- کنترلرها و مسیریابی در Web API
- قوانین نامگذاری مسیر اکشن متدهای REST
- فرمت بازگشتی REST
- چیست؟ HTTP Method

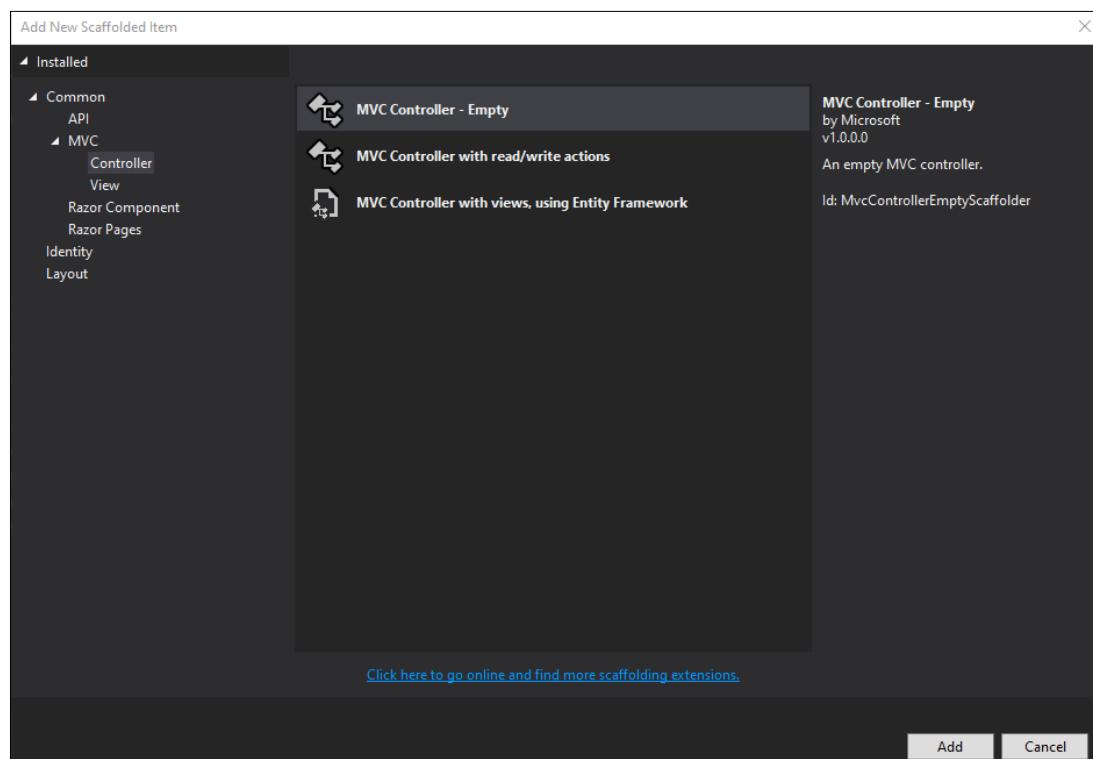
هدف همهی ما اضافه کردن منطق بیزینسی به اپلیکیشن است اما از آنجاییکه HTTP Request‌ها نقش مهمی در Web API بازی می‌کنند باید کمی درباره کلاس‌های کنترلر و مسیریابی صحبت کنیم.

کنترلرها و مسیریابی در Web API

کنترلر نقطه‌ی تعامل کاربر با اپلیکیشن است که:

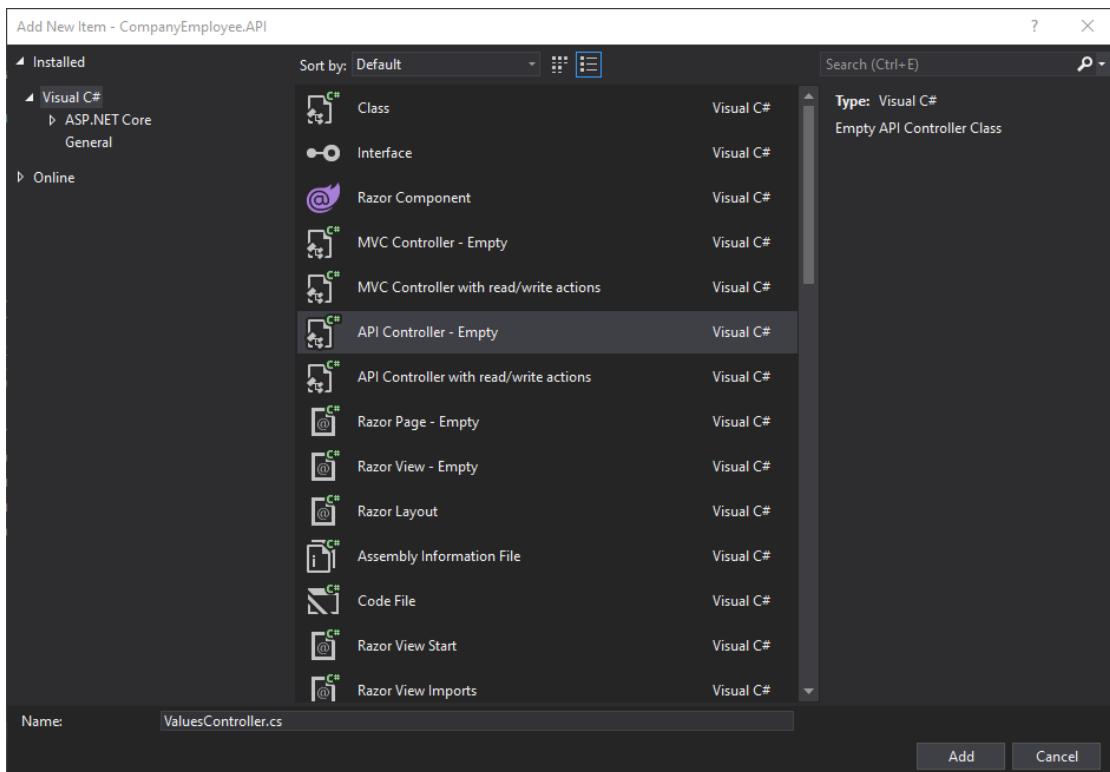
- ریکوئست را دریافت می‌کند.
- بسته به ماهیت ریکوئست داده‌های درخواست شده را از مدل می‌گیرد یا داده‌های مدل را آپدیت می‌کند.
- در پایان ریسپانسی را به کلاینت برمی‌گرداند.

برای ایجاد کنترلر، در پروژه اصلی بر روی فolder Controllers راست کلیک و سپس Add→Controller را انتخاب نمایید.



حالا کلاس API Controller-Empty را انتخاب و بر روی Add کلیک کنید.

در کادر بعد نام آن را CompaniesController بگذارید.



کنترلر ما باید شبیه کد پایین باشد.

```
using Microsoft.AspNetCore.Mvc;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
    }
}
```

بررسی کد :

- همانطور که می‌بینید کلاس ControllerBase بالا، از کلاس ControllerBase ارث بری می‌کند. این کلاس تمام متدهای ضروری برای کلاس CompaniesController را فراهم می‌کند.
- در کد بالا شما اtribut Route را می‌بینید.

```
[Route("api/[controller]")]
```

این، اtribیوت مسیریابی است که ریکوئست‌های ورودی را به اکشن متدهای درون Web API Controller می‌رساند. به محض ارسال یک HTTP Request به اپلیکشن، فریم ورک MVC آن ریکوئست را تجزیه و سپس تلاش می‌کند تا آن را با یک اکشن در کنترلر مطابقت دهد.

دو روش برای تعریف مسیریابی وجود دارد :

(۱) مسیریابی مبتنی بر **Convention**

(۲) مسیریابی مبتنی بر **Attribute**

مسیریابی مبتنی بر Convention قراردادهایی را برای مسیرهای URL ایجاد می‌کند. هنگامی که یک URL صدا زده می‌شود، موتور مسیریابی تلاش می‌کند تا متن URL را (در مکان {controller}) با یک Controller تعریف شده در وب اپلیکیشن مطابقت دهد.

اگر موتور مسیریابی نتواند چیزی برای تطبیق بیابد، خطا نمایش داده خواهد شد. در غیر این صورت نتیجه مسیریابی، یک اکشن‌متدهای Controller مرتبط با آن خواهد بود.



https://localhost:44342/Employee/GetEmployee?employeeid=321230

بخش اول بخش دوم بخش سوم

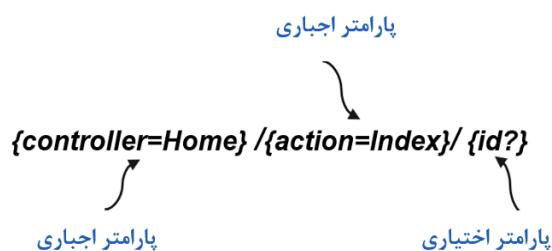
- بخش اول با نام کنترلر مپ می‌شود.
- بخش دوم با نام اکشن مپ می‌شود.
- بخش سوم یک پارامتر اختیاری به نام id را مشخص می‌کند که وارد کردن آن اجباری نیست اما در صورت وجود، مسیریاب مقداری برای پارامتر id ضبط می‌کند.

ما می‌توانیم این نوع مسیریابی را در متدهای Startup Configure کلاس ببینیم.

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

نکته!!

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



در روش بالا باید Controllerها و اکشن‌متدهای خود را تعریف کنید تا ریکوئست دریافتی AttributeRouting مطابق با قرارداد حرکت و بالاخره مسیریابی موردنظر خود را بیابد. اما در باید برای مپ کردن مسیرها عبارت [Route] را بر روی کنترلرها یا اکشن‌متدهای خود قرار دهید.

این روش، انعطاف‌پذیری بیشتری نسبت به روش قبل دارد و برای زمانیکه از Web API استفاده می‌کنید بسیار کارآمد است.

همانطور که بالاتر دیدید، معمولاً مسیر پایه را بالای کلاس کنترلر می‌گذاریم بنابراین برای اکشن متدهای خاص هم، باید مسیرها را درست بالای آن‌ها قرار دهیم:

زمانیکه با پروژه Web API کار می‌کنید، پیشنهاد تیم ASP.NET Core این است که از مسیر پایه مبتنی بر Attribute استفاده کنید.

جست REST

REST یک سبک معماری است که ارتباط کامپیوترهای راه دور را با استفاده از پروتکل HTTP برقرار می‌کند.

اینترنت پر از نظرات این چنینی است اما REST دقیقاً چیست و چه مشکلی را حل می‌کند؟

قبل از بررسی REST، بباید برخی تصورات غلط را پاک کنیم.

- اگر API از HTTP Method هایی مثل GET یا POST استفاده کرد، دلیل بر REST بودن آن نیست.
- اگر یک API خروجی JSON برگرداند، دلیل بر REST بودن آن نیست زیرا متدهای REST هم می‌توانند JSON برگردانند.
- اگر یک API عملیات CRUD را پشتیبانی کرد، باز هم دلیل بر REST بودن آن نیست.

اما RESTful API دقیقاً باید چه ویژگی داشته باشد؟

۱. RESTful API ها باید از HTTP method ها استفاده کنند.
۲. می‌توانند عملیات CURD را مدل نمایند.
۳. می‌توانند JSON هم برگردانند.

اما REST چیست؟

REST مدل کردن^۱ Resource هاست.

REST انجام عملیاتی بر روی Resource را، برای کلاینت فراهم و در نهایت یک Status Code به کاربر برمی‌گرداند. بنابراین از همین جاست که مخفف "Representational State Transfer" می‌باشد.

به طور مثال:

شما در یک RESTful API یک User اضافه می‌کنید و این API یک کد 201 برمی‌گرداند.

۱. آبجکت‌هایی هستند که در طراحی API استفاده می‌کنید. مثل: User, Order و ...

HTTP Method چیست؟

ها افعالی هستند که عملیات Resource‌ها را مشخص می‌کنند. احتمالاً شما HTTP Method و HTTP POST را شنیده و احتمالاً با این افعال کار کرده باشید. بیایید نگاهی دقیق‌تر به این افعال بیندازیم.

• GET : رایج‌ترین و ساده‌ترین HTTP Verb است که می‌توانید با استفاده از آن

Resource موردنظرتان را واکشی کنید. این Verb به صورت Read Only است و نباید

هیچ گونه State اپلیکیشن را تغییر دهد. به طور مثال:

یک کلاینت نمی‌تواند با استفاده از Get یک Resource را حذف یا آپدیت نماید.

• POST : برای ایجاد یا آپدیت بخشی از Resource استفاده می‌شود. اگر

دارای Id باشد این Verb یک Resource را آپدیت می‌کند و گرنه یک

Resource جدید را ایجاد خواهد کرد.

• PUT : هم مانند POST، برای ایجاد یا آپدیت Resource استفاده می‌شود.

اینجا یک سوال ذهن ما را درگیر می‌کند. برای ایجاد یا آپدیت Resource باید

از POST استفاده کرد یا از PUT ؟

برای پاسخ به این سوال باید معنی Idempotent بودن متده را بدانیم.

Idempotent چیست؟

Idempotent یعنی اگر متده هر چند بار هم صدای زده شود، تنها یک تاثیر بر روی

Resource بگذارد. یا به عبارتی اجرای متده هیچ گونه Side Effectی نداشته باشد.

برای مثال :

اگر یک متده Get را، صد بار هم صدای زده باز نماییم؛ تنها یک نتیجه را می‌گیرید و خروجی‌ها

هیچ فرقی با هم ندارند. بنابراین متده Get به عنوان Idempotent شناخته می‌شود.

حالا فرق POST و PUT چیست؟

ریکوئست POST - PUT نیست. یعنی اگر برای ایجاد یک User ده بار

ریکوئست POST ارسال کنیم، ده User جدید اضافه خواهد شد.

در حالیکه ریکوئست‌های POST - PUT هستند. یعنی اگر برای ایجاد یک User

یک ریکوئست PUT ارسال کنیم و سپس ۹ بار دیگر این ریکوئست را تکرار کنیم، این

ریکوئست تنها یک تاثیر را خواهد داشت، چون در بار اول یک User جدید اضافه خواهد شد و ۹ بار دیگر آن User آپدیت می‌شود.

همین مسئله باعث می‌شود ما برای ایجاد یا آپدیت Resource به سمت PUT حرکت کنیم. PUT برخلاف POST – Idempotent است و با اجرای مجدد یک درخواست ناموفق، یک Resource جدید ایجاد نمی‌شود بلکه یک آپدیت کامل انجام خواهد شد. • PATCH برای آپدیت قسمتی از Resource استفاده می‌شود. PATCH نیز Idempotent نیست زیرا اگر همزمان چندین کلاینت با هم شروع به آپدیت قسمتی از Resource کنند و آپدیت هر کدام از آن‌ها متفاوت باشد، کلاینت و سرور از Sync بودن خارج می‌شوند. به طور مثال :

فرض کنید دو کلاینت، همزمان اطلاعات یک کارمند را ویرایش می‌کنند. کلاینت اول نام کارمند را تغییر می‌دهد و کلاینت دوم دپارتمان کارمند را عوض می‌کند.

حالا بعد از آپدیت، این دو کلاینت اطلاعاتی که دریافت می‌کنند با اطلاعات اولیه آن‌ها متفاوت خواهد بود و کلاینت‌ها نمی‌دانند وضعیت واقعی Resource، در سرور چیست.

این موضوع دقیقاً تفاوت بین PATCH و PUT است. در PUT اطلاعات به صورت کامل آپدیت می‌شود و کلاینت دقیقاً می‌داند که اطلاعات این Resource در سرور به چه صورت است اما در Patch این طور نیست.

DELETE : DELETE ، یک Resource را با استفاده از یک ID حذف می‌کند. • هم Idempotent است زیرا هیچ گونه Side Effect ندارد.

مقایسه بین PUT - POST - PATCH

- از نظر فنی، POST برای آپدیت کامل و یا ناقص، PUT برای آپدیت کامل و PATCH برای آپدیت ناقص مورد استفاده قرار می‌گیرد.
- و POST و PATCH هر دو Idempotent-PUT نیستند اما Idempotent است.
- در RESTful API، فقط POST می‌تواند Response Body را برای کلاینت ارسال کند. PATCH و PUT فقط می‌توانند هدرهای Created 201 یا No Content 204 را برگردانند.

تفاوت بین این سه متد باعث ایجاد دو دیدگاه شده است :

۱. دیدگاه اول استفاده از هر سه Verb : برای ایجاد از POST، برای آپدیت

کامل از PUT و برای آپدیت ناقص از PATCH استفاده کنید.

این دیدگاه به تعریف HTTP نزدیکتر است اما یک نکته منفی دارد و آن این است که باید مسیرهای زیادی در برنامه قرار دهید و واقعاً در API‌ها نیازی به تمایز بین آپدیت کامل و ناقص نیست.

۲. دیدگاه دوم ساده نگه داشتن برنامه است: در این دیدگاه، هم برای آپدیت

و هم برای ایجاد Resource از POST استفاده می‌شود. این دیدگاه برای کلاینت ساده‌تر است چون کلاینت نیازی به تصمیم‌گیری بین آپدیت ناقص و کامل ندارد.

فرمت بازگشتی REST

یک API RESTful API می‌تواند XML، JSON یا هرچیزی را به طور کامل برگرداند. اما JSON یک فرمت محبوب برای Web API است. بیایید نگاهی دقیق‌تر به مزایا و معایب RESTful API JSON صحبت کنیم.

• یکی از بزرگترین دلایلی که باعث شد JSON نسبت به XML محبوبیت بیشتری

بیابد این است که، خواندن آن برای انسان راحت‌تر است.

• خواندن JSON برای ماشین هم ساده‌تر است زیرا نیازی به XML parsing ندارد و خواندن و نوشتن JSON تقریباً در هر سیستم عاملی آسان است.

• برای تعریف JSON هیچ Schema وجود ندارد بنابراین ساده‌تر از XML است و راحت‌تر می‌توان آن را تغییر داد. البته همین مورد می‌تواند به عنوان یک عیب هم تلقی شود چون هیچ ساختاری برای کلاینت وجود ندارد.

قوایین نامگذاری مسیر اکشن متد

اسمی که در URI استفاده می‌شود نمایانگر Resource است و به کاربر کمک می‌کند تا بفهمد با چه Resource کار می‌کند.

نام Resource در URI، همیشه باید یک اسم باشد. این بدان معناست که به طور مثال، اگر می‌خواهیم اکشن متده را، برای بدست آوردن اطلاعات شرکت‌ها ایجاد کنیم باید آن را با Companies GetCompanies اجتناب کنیم.

بخش مهم دیگری که باید به آن توجه کنیم، سلسله مراتب بین Resource‌هاست. در این اپلیکیشن، Company به عنوان Entity اصلی و Employee به عنوان یک Entity وابسته است. وقتی می‌خواهیم یک مسیر برای Entity وابسته ایجاد کنیم، باید قرارداد را کمی متفاوت‌تر دنبال کنیم.

/api/principalResource/{principalId}/dependentResource.

به طور مثال:

از آنجا که هیچ کارمندی نمی‌تواند بدون شرکت باشد پس مسیر برای Resource کارمندان باید به صورت پایین نوشته شود.

api/companies/{companyId}/employees/

با توجه به این توضیحات باید با ریکوئست Get شروع کنیم.

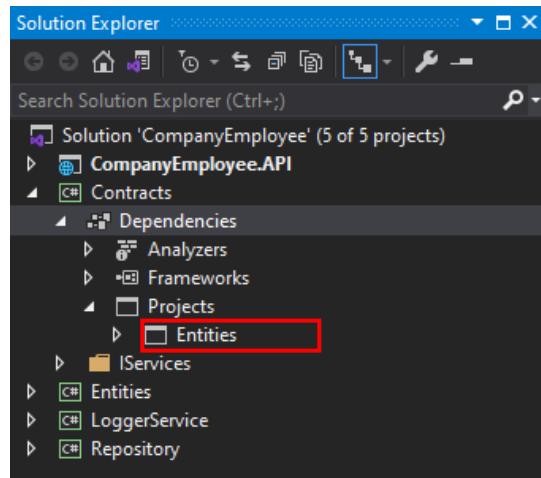
گرفتن اطلاعات شرکت‌ها از دیتابیس

اولین کاری که می‌خواهیم انجام دهیم تغییر مسیر از [Route("api/[controller]")] به [Route("api/[companies]")] است. با اینکه مسیر اول خیلی خوب کار می‌کند اما با مثال CompaniesController دوم، به صورت صریح مشخص می‌کنیم که مسیریاب باید به کلاس اشاره کند.

حال زمان آن رسیده که برای برگرداندن اطلاعات تمام شرکت‌ها اولین اکشن متده را بنویسیم.

گام اول : در اینترفیس ICompanyRepository متدی به نام GetAllCompanies اضافه کنید.

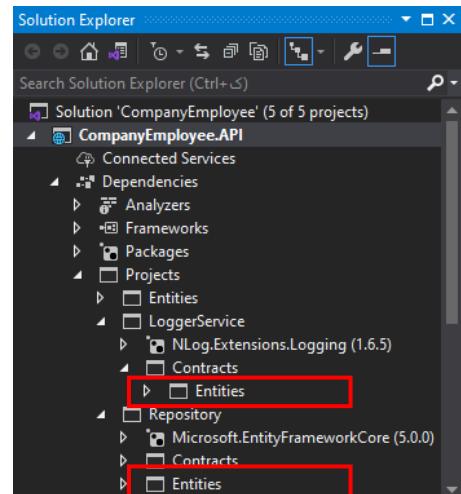
برای نوشتمن این متده باید به پروژه Contracts یک رفرنس از پروژه Entities اضافه کنیم.



توجه!!

در اینجا می‌خواهیم توجه شما را به یک نکته مهم جلب کنم.

پروژه اصلی به پروژه‌های Entities و Repository و LoggerService رفرنس دارد. از آنجاییکه یک رفرنس از پروژه Contracts دارند، پس پروژه اصلی ما از طریق Repository و LoggerService به پروژه Entities دسترسی دارد.



بنابراین می‌توانیم رفرنس پروژه Entities را از پروژه اصلی حذف کنیم.

: ICompanyRepository اینترفیس

```
using Entities.Models;
using System.Collections.Generic;

namespace Contracts.IServices
{
```

```

public interface ICompanyRepository
{
    IEnumerable<Company> GetAllCompanies(bool trackChanges);
}
}

```

گام دوم : حالا باید این اینترفیس را در کلاس CompanyRepository پیاده سازی کنیم.

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using Repository.Repositories;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext): base(companyEmployeeDbContext)
        {
        }

        public IEnumerable<Company> GetAllCompanies(bool trackChanges) =>
            FindAll(trackChanges)
                .OrderBy(c => c.Name)
                .ToList();
    }
}

```

گام سوم : در پایان باید در CompaniesController اطلاعات شرکت ها را با استفاده از متدهای GetAllCompanies برگردانیم.

```

using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using System;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]

```

```

[ApiController]
public class CompaniesController : ControllerBase
{
    private readonly IRepositoryManager _repository;
    private readonly ILoggerManager _logger;

    public CompaniesController(IRepositoryManager repository,
        ILoggerManager logger)
    {
        _repository = repository;
        _logger = logger;
    }

    [HttpGet]
    public IActionResult GetCompanies()
    {
        try
        {
            var companies =
                _repository.Company.GetAllCompanies(trackChanges: false);

            return Ok(companies);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong in the
                {nameof(GetCompanies)} action {ex} ");

            return StatusCode(500, "Internal server error");
        }
    }
}
}

```

بررسی کد :

- اول از همه، سرویس RepositoryManager و Logger را درون Constructor تزریق کردیم.
- سپس اتریبوت [HttpGet] را بالای اکشن متدهای GetCompanies گذاشتیم، تا عملکرد این متدهای را مشخص کنیم. با این کار این اکشن متدهای را با ریکوئست Get مپ می‌کنیم.

- درون بدنیه اکشن متدها برای Log کردن پیامها و گرفتن دیتا از کلاس ریپازیتوری از سرویس‌های Inject شده استفاده کردیم.
- ما اینترفیس IActionResult را در انواع اکشن متدها استفاده می‌کنیم تا اکشن متدهای بتوانند یک Status Code را همراه با نتیجه متدهای برگرداند. در اینجا متدهای OK اطلاعات تمام شرکت‌ها را همراه با 200 Status Code برمی‌گرداند.
- می‌خواهیم اگر یک اکسپشن رخ دهد، 500 Status Code را (که نمایانگر خطای داخلی سرور است) برگردانیم. پس در قسمت Catch این Status Code را Return می‌کنیم.
- چون بالای اکشن متدهای [Route] وجود ندارد پس مسیر این متدهای برابر api/companies است. این مسیر همان مسیری است که بالای کنترلر قرار گرفته است.

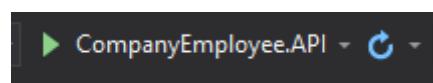
تست کردن اکشن متدهای GetCompanies

کلید F5 را بزنید تا برنامه اجرا شود. بعد از اجرا شدن، چک کنید که آیا اپلیکیشن به آدرس https://localhost:5001 گوش می‌دهد یا خیر.

```

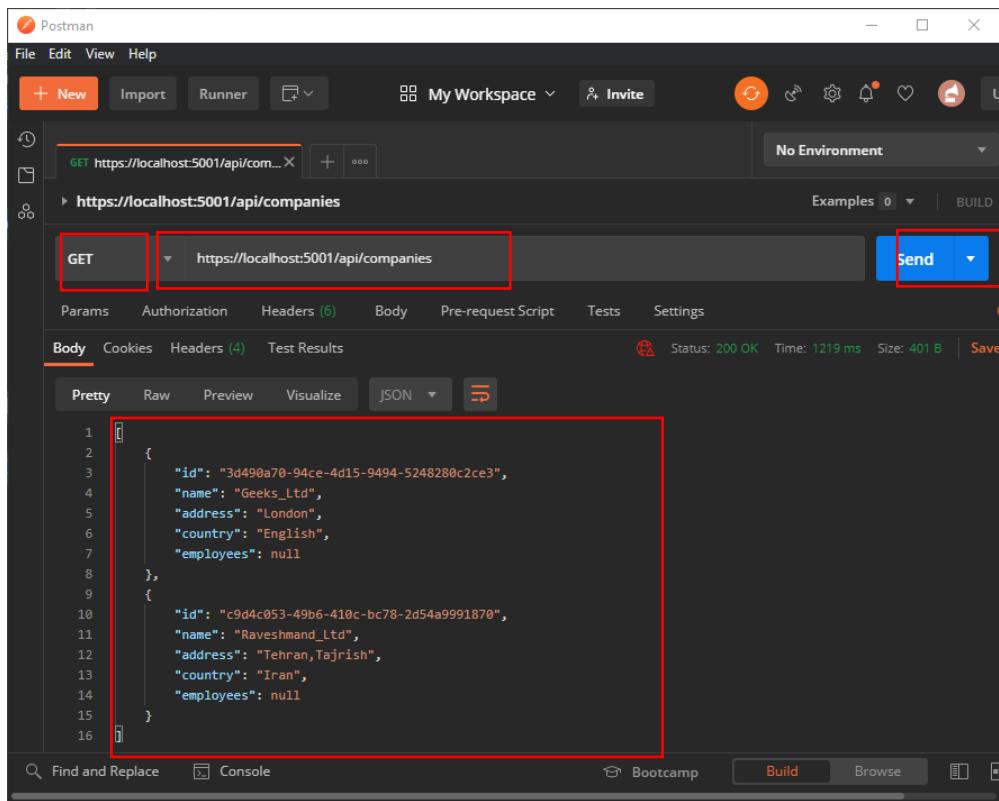
Info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
Info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
Info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
Info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
      +-----+
  
```

اگر اینگونه نباشد احتمالاً آن را در حالت IIS اجرا کرده‌اید. پس اپلیکیشن را از حالت اجرا خارج کنید و دوباره همانند تصویر، در مد CompanyEmployees اجرا کنید.



اکنون می‌توانیم از Postman برای تست نتیجه استفاده کنیم.

<https://localhost:5001/api/companies>



عالی شد. همه چیز مطابق با کدی که نوشتیم کار کرد.

کلاس‌های DTO در مقابل کلاس‌های Entity Model

کلاسی است که برای انتقال داده، بین اپلیکیشن کلاینت و سرور از آن استفاده می‌شود. این کلاس به ما امکان می‌دهد تا، ساختار داده‌ای Backward Compatible و مطابق با نیاز کلاینت ایجاد کنیم.

یعنی اینکه اگر Entity API را تغییر کند، چون همیشه دیتاهایی که کلاینت برای ما می‌فرستد باید با مدلی که ما از ورودی می‌گیریم یکی باشد. به بیان دیگر:

اگر به هر دلیلی نیاز باشد دیتابیس را تغییر دهیم، باید پر اپرتی‌های مدل ورودی هم تغییر کند. همانطور که می‌دانید تغییر مدل دیتابیس به این معنی نیست که اپلیکیشن سمت کلاینت هم باید تغییر کند. بنابراین اگر DTO داشته باشیم، نتیجه‌ای که کلاینت داشته باشد، مثل گذشته می‌ماند و تنها Entity ما تغییر می‌کند.

ما در کد بالا برای مپ کردن ریکوئست به دیتابیس، مستقیماً از Company Entity استفاده و نتیجه را به کلاینت برمی‌گردانیم. برگرداندن Entity به عنوان Response یک Bad Practice است چون هیچ Backward Compatibility نداریم و گاهی شاید نخواهیم همه پراپرتی‌ها یک Entity را به عنوان نتیجه برگردانیم. به عنوان مثال:

همانطور که می‌دانید Entity‌های ما دارای Navigation Property هایی هستند که شاید نخواهیم این پراپرتی‌ها را در ورودی بفرستیم.

پس تا اینجا متوجه شدیم که استفاده از DTO باعث می‌شود، کد ما قابل نگهداری و بهتر باشد. خب حالا بباید در پروژه Entities یک فolder به نام DataTransferObjects ایجاد و در آن یک کلاس با نام CompanyDto اضافه کنیم.

```
using System;

namespace Entities.DataTransferObjects
{
    public class CompanyDto
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public string FullAddress { get; set; }
    }
}
```

بررسی کد :

- در این کلاس پراپرتی Employees را حذف کردیم چون در اکشن متدهای GetCompanies و DeleteCompany نداریم.
- یک پراپرتی FullAddress در این کلاس اضافه کردیم تا مقدار پراپرتی‌های Address و Country موجود در کلاس Company را به هم بچسبانیم و در این پراپرتی بریزیم.
- همچنین در این کلاس، هیچ اtribut اعتبارسنجی استفاده نکردیم چون این کلاس فقط برای برگرداندن Response به کلاینت مورد استفاده قرار می‌گیرد پس اtribut اعتبارسنجی الزامی نیست.

باید اکشن متدهیم GetCompanies را با هم تغییر دهیم.

```
using Contracts.IServices;
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Linq;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;

        public CompaniesController(IRepositoryManager repository,
        ILoggerManager logger)
        {
            _repository = repository;
            _logger = logger;
        }

        [HttpGet]
        public IActionResult GetCompanies()
        {
            try
            {
                var companies =
                    _repository.Company.GetAllCompanies(trackChanges: false);

                var companiesDto = companies.Select(c => new CompanyDto
                {
                    Id = c.Id,
                    Name = c.Name,
                    FullAddress = string.Join(' ', c.Address, c.Country)
                }).ToList();
            }
            return Ok(companiesDto);
        }
    }
}
```

```

        catch (Exception ex)
    {
        _logger.LogError($"Something went wrong in the
{nameof(GetCompanies)} action {ex} ");

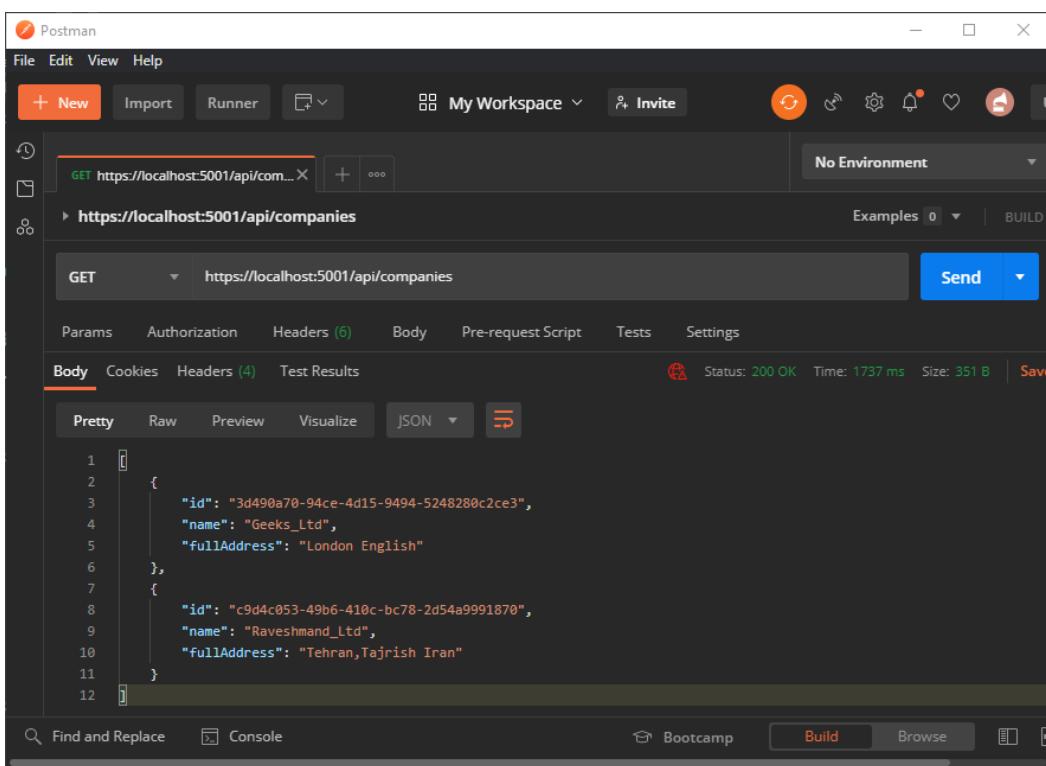
        return StatusCode(500, "Internal server error");
    }
}

}
}
}

```

حالا اپلیکیشن را استارت و ریکوئست پایین را در Postman اجرا کنید.

<https://localhost:5001/api/companies>



استفاده از AutoMapper در ASP.NET Core

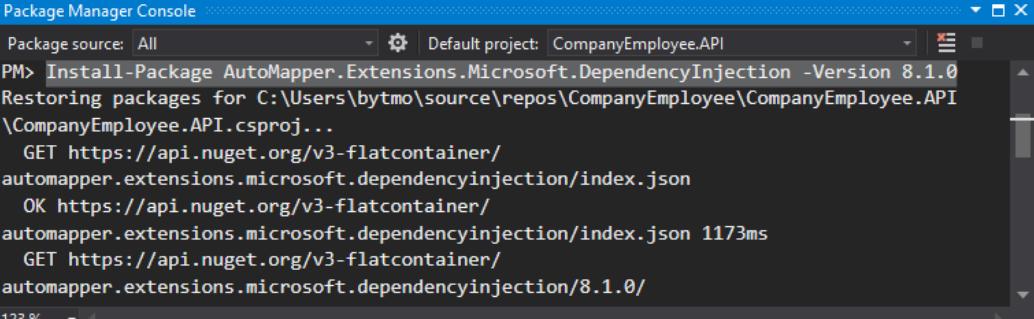
اگر به مپ شدن کد، در اکشن متدهای `GetCompanies` توجه کنید، میبینید که عمل مپ کردن پراپرتیها را به صورت دستی انجام دادیم. مطمئناً این نوع مپ کردن، برای چند فیلد مشکل ندارد اما اگر تعداد این پراپرتیها زیاد شود کدی شلوغ و آشفته خواهیم داشت.

یک روش بهتر برای مپ کردن کلاس‌ها، استفاده از کتابخانه `Automapper` است.

کتابخانه AutoMapper به ما کمک می کند تا آبجکت ها را در اپلیکیشن مپ کنیم و کدی خواناتر و قابل نگهداری داشته باشیم.

برای استفاده از این کتابخانه اولین قدم نصب AutoMapper، در پروژه است پس پنجره Package Manager Console CompanyEmployee.API را باز کنید و دستور پایین را اجرا نمایید.

```
Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection -  
Version 8.1.0 -ProjectName CompanyEmployee.API
```



The screenshot shows the Package Manager Console window with the command entered in the input field. The output pane displays the progress of the package restoration, including network requests for the package index and its dependencies.

بعد از نصب، باید این کتابخانه را در متدهای ConfigureServices رजیستر کنیم.

کدهای کلاس Startup :

```
using CompanyEmployee.API.Infrastructure.Extensions;  
using Microsoft.AspNetCore.Builder;  
using Microsoft.AspNetCore.Hosting;  
using Microsoft.AspNetCore.HttpOverrides;  
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
using Microsoft.OpenApi.Models;  
using NLog;  
using System.IO;  
using AutoMapper;  
  
namespace CompanyEmployee.API  
{  
    public class Startup
```

```

{
  public Startup(IConfiguration configuration)
  {
    LogManager.LoadConfiguration(string.Concat(Directory.GetCurrentDirectory(),
      "/nlog.config"));
    Configuration = configuration;
  }

  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.ConfigureCors();
    services.ConfigureIISIntegration();
    services.ConfigureLoggerService();
    services.ConfigureSqlContext(Configuration);
    services.ConfigureRepositoryManager();
    services.AddAutoMapper(typeof(Startup));
    services.AddControllers();

    services.AddSwaggerGen(c =>
    {
      c.SwaggerDoc("v1", new OpenApiInfo { Title =
        "CompanyEmployee.API", Version = "v1" });
    });
  }

  public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
      app.UseSwagger();
      app.UseSwaggerUI(c =>
        c.SwaggerEndpoint("/swagger/v1/swagger.json",
          "CompanyEmployee.API v1"));
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
  }
}

```

```

    app.UseCors("CorsPolicy");
    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.All
    });

    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}
}

```

پس از رجیستر شدن این کتابخانه، باید یک کلاس Profile ایجاد کنیم تا در آنجا عمل مپ شدن آبجکت مبدا و مقصد مشخص شود.

بنابراین یک کلاس با نام MappingProfile در فولدر Infrastructure (پروژه) ایجاد کنید و کدهای زیر را درون آن بنویسید.

```

using AutoMapper;
using Entities.DataTransferObjects;
using Entities.Models;

namespace CompanyEmployee.API.Infrastructure
{
    public class MappingProfile : Profile
    {
        public MappingProfile()
        {
            CreateMap<Company, CompanyDto>()
                .ForMember(c => c.FullAddress,
                    opt => opt.MapFrom(x => string.Join(' ', x.Address,
                        x.Country)));
        }
    }
}

```

}

بررسی کد :

- باید کلاس Profile از کلاس MappingProfile موجود در کتابخانه AutoMapper استفاده کنیم تا شی منبع و ارث بری کند.
 - در Constructor این کلاس، باید از متدهای CreateMap استفاده کنیم.
 - مقصود را برای مپ شدن مشخص شود.
 - از آنجاییکه ما یک پراپرتی FullAddress در DTO داریم و اطلاعات آن باید شامل دو پراپرتی Address و Country باشد، پس باید قوانین اضافه شدن مپ را در متدهای ForMember مشخص کنیم.
- حالا می‌توانیم این سرویس را همانند سایر سرویس‌ها در کنترلر Inject کنیم.

```
using AutoMapper;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public CompaniesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }
    }
}
```

```

    }

    [HttpGet]
    public IActionResult GetCompanies()
    {
        try
        {
            var companies =
                _repository.Company.GetAllCompanies(trackChanges: false);

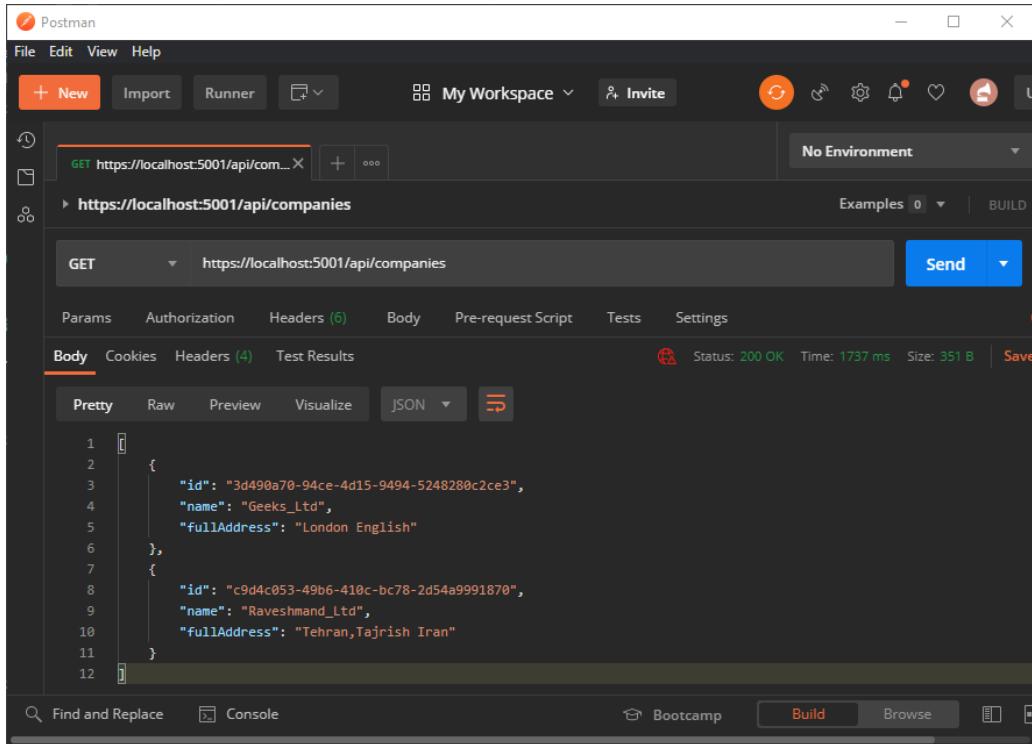
            var companiesDto =
                _mapper.Map<IEnumerable<CompanyDto>>(companies);

            return Ok(companiesDto);
        }
        catch (Exception ex)
        {
            _logger.LogError($"Something went wrong in the
                {nameof(GetCompanies)} action {ex} ");
            return StatusCode(500, "Internal server error");
        }
    }
}
}

```

خیلی عالی شد. باید دوباره این ریکوئست را در Postman تست بزنیم.

<https://localhost:5001/api/companies>



می بینم که همه چیز مانند قبل، به درستی کار می کند با این تفاوت که الان کد تمیزتر شده است.

فصل پنجم : مدیریت Exception ها و Content Negotiation

آنچه خواهید آموخت:

- مدیریت Exception ها
- ارتباطات Web API در Parent/Child
- Content Negotiation چیست؟
- تغییر پیکربندی Response
- محدود کردن Media Type
- ایجاد فرمت سفارشی

مدیریت Exception ها

یکی از جنبه‌های مهم یک اپلیکیشن، مدیریت Exception‌هاست. 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 Feature یک Option 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 :

• Exception ها

• و خطاهای Status code

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

اگر یک Exception در یک Middleware اتفاق بیفتد، در سراسر Pipeline منتشر می‌شود. اگر آن را مدیریت نکند، وب سرور، Status code 500 را به کاربر برمی‌گرداند.

گاهی اوقات هیچ Exception‌ی رخ نمی‌دهد اما Middleware ممکن است کد خطایی را ایجاد کند. یکی از مواردی که ممکن است دیده باشید، زمانیست که مسیر ریکوئست Handle

نمی‌شود. در این وضعیت Pipeline خطای 404 را برمی‌گرداند. در نتیجه صفحه خطای 404 به کاربر نمایش داده می‌شود.

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

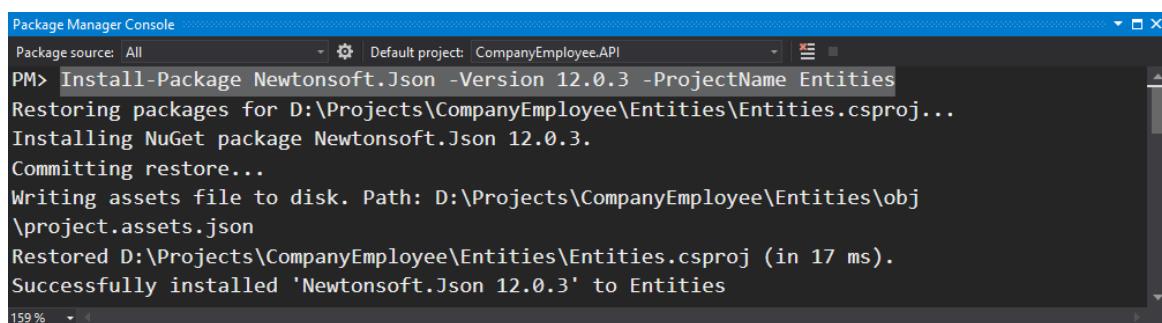
برای حل این مشکلات، Error handling middleware تلاش می‌کند قبل از اینکه اپلیکیشن چیزی را به کاربر نمایش دهد، Response را تغییر دهد.

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

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

در ASP.NET Core یک Middleware داخلی به نام UseExceptionHandler وجود دارد که می‌توانیم از آن، برای مقابله با Exception استفاده کنیم.
ابتدا در پروژه Entities پکیج پایین را نصب کنید.

Install-Package Newtonsoft.Json -Version 12.0.3 -ProjectName Entities



```
PM> Install-Package Newtonsoft.Json -Version 12.0.3 -ProjectName Entities
Restoring packages for D:\Projects\CompanyEmployee\Entities\Entities.csproj...
Installing NuGet package Newtonsoft.Json 12.0.3.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\Entities\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\Entities\Entities.csproj (in 17 ms).
Successfully installed 'Newtonsoft.Json 12.0.3' to Entities
```

حالا در این پروژه یک فolder جدید به نام ErrorModel ایجاد و در آن کلاسی با نام ErrorDetails اضافه نمایید.

```
using Newtonsoft.Json;

namespace Entities.ErrorModel
{
    public class ErrorDetails
```

```

        public int StatusCode { get; set; }
        public string Message { get; set; }
        public override string ToString() =>
            JsonConvert.SerializeObject(this);
    }

}

```

این کلاس برای نگهداری جزئیات پیام خطای بوجود آمده، استفاده می‌شود. برای ادامه در فولدر Extensions پروژه اصلی، یک کلاس جدید به نام ExceptionMiddlewareExtensions.cs اضافه کنید.

```

using Contracts.IServices;
using Entities.ErrorModel;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Http;
using System.Net;

namespace CompanyEmployee.API.Infrastructure.Extensions
{
    public static class ExceptionMiddlewareExtensions
    {
        public static void ConfigureExceptionHandler(this IApplicationBuilder app, ILoggerManager logger)
        {
            app.UseExceptionHandler(appError =>
            {
                appError.Run(async context =>
                {
                    context.Response.StatusCode =
                        (int) HttpStatusCode.InternalServerError;
                    context.Response.ContentType = "application/json";
                    var contextFeature =
                        context.Features.Get<IExceptionHandlerFeature>();
                    if (contextFeature != null)
                    {
                        logger.LogError($"Something went wrong:
{contextFeature.Error}");
                    }
                });
            });
        }
    }
}

```

```

        await context.Response.WriteAsync(new
            ErrorDetails()
        {
            StatusCode = context.Response.StatusCode,
            Message = "Internal Server Error."
        }.ToString());
    });
});
}
}
}

```

: بررسی کد :

- در این کلاس یک اکستنشن متدهایجاد کردیم که در آن UseExceptionHandler رجیستر و سپس Status Code و نوع محتوای Response را پرمی‌کند.
- در پایان پیغام خطا را Log و ریسپانس را می‌فرستد.

برای استفاده از این اکستنشن متدهاید که پایین را در Startup کلاس Configure اضافه کنید.

```

using CompanyEmployee.API.Infrastructure.Extensions;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpOverrides;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Contracts.IServices;
using Microsoft.OpenApi.Models;
using NLog;
using System.IO;
using AutoMapper;

namespace CompanyEmployee.API
{
    public class Startup
    {

```

```

public Startup(IConfiguration configuration)
{
    LogManager.LoadConfiguration(string.Concat(Directory.GetCurrentDirectory(), "/nlog.config"));
    Configuration = configuration;
}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureCors();
    services.ConfigureIISIntegration();
    services.ConfigureLoggerService();
    services.ConfigureSqlContext(Configuration);

    services.ConfigureRepositoryManager();
    services.AddAutoMapper(typeof(Startup));
    services.AddControllers();

    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title =
            "CompanyEmployee.API", Version = "v1" });
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env, ILoggerManager logger)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c =>
        c.SwaggerEndpoint("/swagger/v1/swagger.json",
            "CompanyEmployee.API v1"));
    }

    app.ConfigureExceptionHandler(logger);

    app.UseHttpsRedirection();
}

```

```
        app.UseStaticFiles();

        app.UseCors("CorsPolicy");

        app.UseForwardedHeaders(new ForwardedHeadersOptions
        {
            ForwardedHeaders = ForwardedHeaders.All
        });

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

UseExceptionHandler تست

برای تست این Middleware کد try-catch را از متدهای GetCompanies حذف و سپس برای شبیه سازی یک پیام خطا، اکسپشن پایین را به کد اضافه کنید.

```
using AutoMapper;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;
```

```

public CompaniesController(IRepositoryManager repository,
ILoggerManager logger,
IMapper mapper)
{
    _repository = repository;
    _logger = logger;
    _mapper = mapper;
}

[HttpGet]
public IActionResult GetCompanies()
{
    var companies =
        _repository.Company.GetAllCompanies(trackChanges: false);

    var companiesDto =
        _mapper.Map<IEnumerable<CompanyDto>>(companies);

    throw new Exception("Exception");

    return Ok(companiesDto);
}
}
}
}

```

حالا با Postman ریکوئست پایین را تست کنید.

<https://localhost:5001/api/companies>

The screenshot shows the Postman interface with a failed request to `https://localhost:5001/api/companies`. The response status is `500 Internal Server Error`. The JSON body of the response is highlighted with a red box and contains the following content:

```

{
  "StatusCode": 500,
  "Message": "Internal Server Error."
}

```

گرفتن اطلاعات یک Resource از دیتابیس

تا اینجا با ریکوئست GET آشنا شدیم. حالا می‌خواهیم اکشن متدهای برای گرفتن اطلاعات یک شرکت بنویسیم.

اولین گام اضافه کردن یک متدهای CompanyRepository است. باید اینترفیس ICompanyRepository را همانند کد پایین تغییر دهیم.

```
using Entities.Models;
using System.Collections.Generic;
using System;

namespace Contracts.IServices
{
    public interface ICompanyRepository
    {
        IEnumerable<Company> GetAllCompanies(bool trackChanges);
        Company GetCompany(Guid companyId, bool trackChanges);
    }
}
```

خب حالا باید این اینترفیس را در کلاس CompanyRepository پیاده‌سازی کنیم.

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext)
            : base(companyEmployeeDbContext)
        { }
```

```

    public IEnumerable<Company> GetAllCompanies(bool trackChanges) =>
        FindAll(trackChanges)
            .OrderBy(c => c.Name)
            .ToList();

    public Company GetCompany(Guid companyId, bool trackChanges) =>
        FindByCondition(c => c.Id.Equals(companyId), trackChanges)
            .SingleOrDefault();
}

}

```

در پایان باید اکشن متد GetCompany را در CompanyController اضافه کنید.

```

using AutoMapper;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public CompaniesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }

        [HttpGet]
        public IActionResult GetCompanies()
        {

```

```

        var companies =
            _repository.Company.GetAllCompanies(trackChanges: false);

        var companiesDto =
            _mapper.Map<IEnumerable<CompanyDto>>(companies);

        return Ok(companiesDto);
    }

    [HttpGet("{id}")]
    public IActionResult GetCompany(Guid id)
    {
        var company = _repository.Company.GetCompany(id, trackChanges:
false);
        if (company == null)
        {
            _logger.LogError($"Company with id: {id} doesn't exist in
the database.");

            return NotFound();
        }
        else
        {
            var companyDto = _mapper.Map<CompanyDto>(company);

            return Ok(companyDto);
        }
    }
}

```

: بررسی کد :

- مسیر این اکشن متدهای /api/companies و /api/companies/id است. عبارت /api/companies قسمتی از مسیر ریشه ا است (بالای کنترلر) و id قسمتی از اtribut اکشن است.
- این اکشن متدهای اطلاعات یک شرکت را با توجه به id ورودی، در دیتابیس جستجو میکند. اگر این شرکت وجود نداشت با استفاده از متدهای NotFound کد 404 را

برمی‌گردند در غیر این صورت اطلاعات شرکت را به CompanyDto مپ می‌کند و همراه با کد 200 به کلاینت برمی‌گرداند.

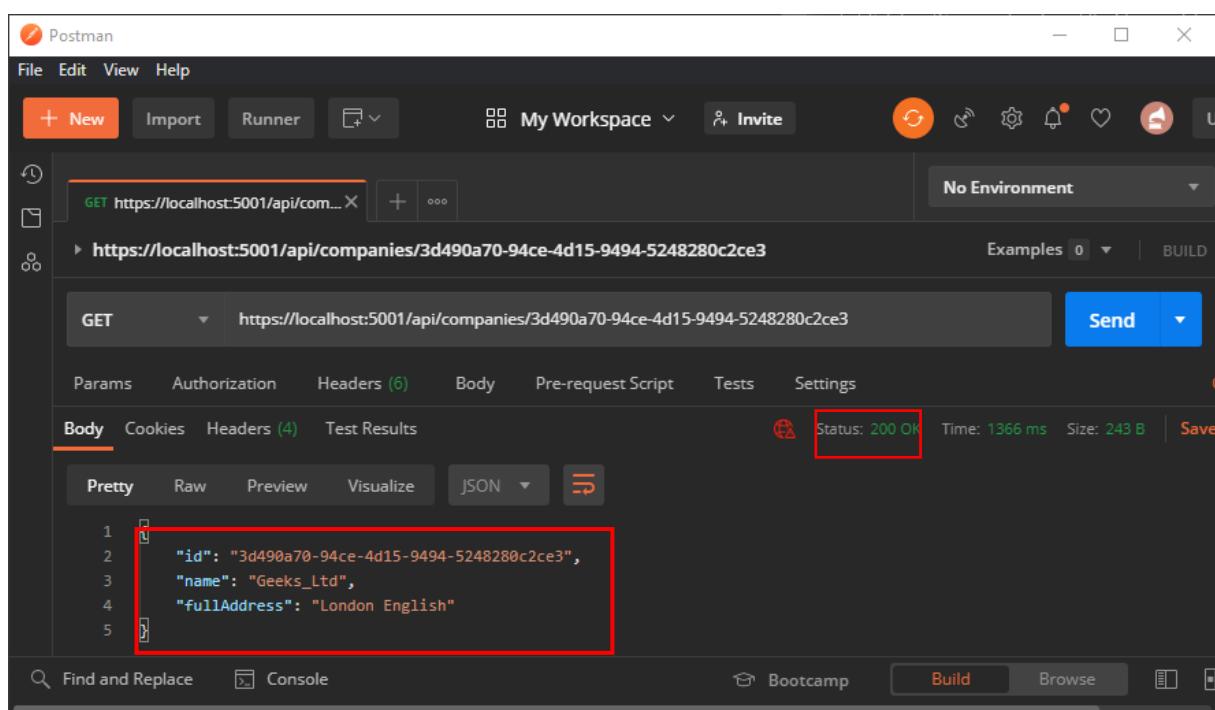
- در ASP.NET Core، تعدادی متدهای وجود دارد که تنها با دیدن نام آن‌ها می‌توانیم بفهمیم که چه کاری انجام می‌دهند. به طور مثال :

متدهای OK برای نتیجه خوب است و Status Code 200 را برمی‌گرداند. و یا متدهای NotFound برای نتیجه‌ای است که پیدا نشده و (Status Code 404) را برمی‌گرداند.

بیایید اینبار یک ریکوئست معتبر و یک ریکوئست نامعتبر را با Postman ارسال کنیم، تا نتیجه هر کدام از متدهای ASP.NET Core را ببینیم.

ریکوئست معتبر :

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>

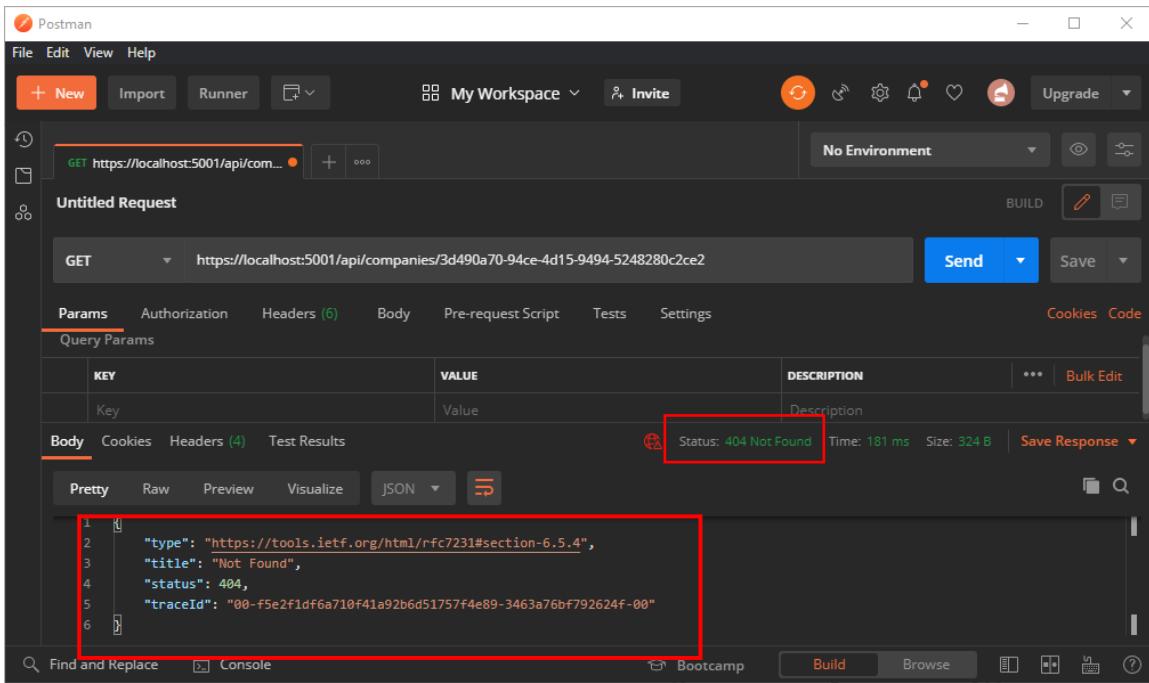


The screenshot shows the Postman interface with a successful GET request to the specified URL. The response status is 200 OK, and the JSON data returned is highlighted with a red box.

```
1 "id": "3d490a70-94ce-4d15-9494-5248280c2ce3",
2 "name": "Geeks_Ltd",
3 "fullAddress": "London English"
```

ریکوئست نامعتبر :

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce2>



ارتباطات Web API در Parent/Child

تا اینجا فقط با مدل Parent Entity Company که یک کار کردیم. اما همانطور که می دانید، هر شرکت تعدادی کارمند مرتبط دارد که وابسته شرکت هستند. هر کارمند باید با یک شرکت خاص در ارتباط باشد بنابراین باید URI های این کارمندان از این طریق ایجاد شوند.

خب بیایید در فolder Controllers یک کنترلر جدید با نام EmployeesController ایجاد کنیم.

```
using AutoMapper;
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies/{companyId}/employees")]
    [ApiController]
    public class EmployeesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public EmployeesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
```

```

    {
        _repository = repository;
        _logger = logger;
        _mapper = mapper;
    }
}

```

بررسی کد :

- همه شما با این کدها آشنا هستید. تنها نکته این که، Route بالای کنترلر است که کمی متفاوت می‌باشد. همانطور که گفتیم یک کارمند نمی‌تواند بدون شرکت باشد و این دقیقاً چیزی است که ما از طریق این URI نشان می‌دهیم.
- برای واکشی یک یا چند کارمند از دیتابیس، باید پارامتر CompanyId را مشخص کنیم. همانطور که می‌دانید این چیزی است که در تمام اکشن‌ها مشترک می‌باشد به همین دلیل ما این مسیر را به عنوان مسیر ریشه تعیین می‌کنیم.

قبل از ایجاد یک اکشن متد برای واکشی کارمندان هر شرکت، باید تغییراتی در اینترفیس IEmployeeRepository

```

using Entities.Models;
using System;
using System.Collections.Generic;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        IEnumerable<Employee> GetEmployees(Guid companyId, bool
trackChanges);
    }
}

```

حالا باید این اینترفیس را در کلاس EmployeeRepository پیاده‌سازی کنیم.

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using System;

```

```

using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
        IEmployeeRepository
    {
        public EmployeeRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext): base(companyEmployeeDbContext)
        {
        }

        public IEnumerable<Employee> GetEmployees(Guid companyId, bool
            trackChanges) => FindByCondition(e =>
            e.CompanyId.Equals(companyId), trackChanges).OrderBy(e => e.Name);

    }
}

```

در پایان باید متدها GetEmployeesForCompany را در EmployeesController اضافه کنیم.

```

using AutoMapper;
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using System;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies/{companyId}/employees")]
    [ApiController]
    public class EmployeesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public EmployeesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
        {
            _repository = repository;

```

```

        _logger = logger;
        _mapper = mapper;
    }

    [HttpGet]
    public IActionResult GetEmployeesForCompany(Guid companyId)
    {
        var company = _repository.Company.GetCompany(companyId,
trackChanges: false);

        if (company == null)
        {
            _logger.LogError($"Company with id: {companyId} doesn't exist
in the database.");
        }

        return NotFound();
    }

    var employeesFromDb =
        _repository.Employee.GetEmployees(companyId, trackChanges:
false);

    return Ok(employeesFromDb);
}
}

}

```

: بررسی کد :

- این متده بسیار ساده است و در آن کدی نیست که تاکنون ندیده باشیم. اما دو نکته وجود دارد که بهتر است با هم بررسی کنیم :
- این اکشن متده یک پارامتر companyId از ورودی میگیرد که این پارامتر، از مسیر اصلی مپ میشود. به همین دلیل ما این پارامتر را در اتریبوت [HttpGet] قرار ندادیم.
- خروجی این متده یک Entity است و همانطور که قبلا گفتیم، بهتر است خروجی که به کلاینت برگردانده میشود یک DTO باشد. بنابراین یک کلاس با نام EmployeeDto در فolder DataTransferObjects اضافه کنید.

```

using System;

namespace Entities.DataTransferObjects
{
    public class EmployeeDto
    {
        public Guid Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public string Position { get; set; }
    }
}

```

حالا باید یک MappingProfile در کلاس MappingProfile اضافه کنیم.

```
CreateMap<Employee, EmployeeDto>();
```

در پایان باید اکشن خود را تغییر دهیم.

```

using AutoMapper;
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using Entities.DataTransferObjects;
using System.Collections.Generic;
using System;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies/{companyId}/employees")]
    [ApiController]
    public class EmployeesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public EmployeesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
        }
    }
}

```

```

        _mapper = mapper;
    }

[HttpGet]
public IActionResult GetEmployeesForCompany(Guid companyId)
{
    var company = _repository.Company.GetCompany(companyId,
trackChanges: false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't exist
in the database.");
    }

    return NotFound();
}

var employeesFromDb =
_repository.Employee.GetEmployees(companyId, trackChanges:
false);

var employeesDTO =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

return Ok(employeesDTO);
}
}

}

```

بعد از انجام شدن این کارها می‌توانیم ریکوئستی با یک CompanyId معتبر ارسال کنیم.

<https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees>

Postman screenshot showing a successful GET request to `https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees`. The response status is 200 OK, and the body contains two employee objects:

```
1 [
2   {
3     "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
4     "name": "Ali Bayat",
5     "age": 30,
6     "position": "Backend developer"
7   },
8   {
9     "id": "80abbca8-664d-4b20-b5de-024705497d4a",
10    "name": "Zahra Bayat",
11    "age": 26,
12    "position": "Backend developer"
13 }
14 ]
```

حالا با یک CompanyId نا معتبر این را تست بزنیم.

<https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991873/employees>

Postman screenshot showing a 404 Not Found error for the URL `https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991873/employees`. The response status is 404 Not Found, and the body contains the following JSON object:

```
1 [
2   {
3     "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
4     "title": "Not Found",
5     "status": 404,
6     "traceId": "00-dc493daa327ab942a9a93d63bd2d24d5-19bef148ff6b1345-00"
7   }
8 ]
```

واکشی اطلاعات کارمند

برای نوشتن این اکشن متدهای همانند بخش قبلی ابتدا اینترفیس IEmployeeRepository را تغییر دهیم.

```
using Entities.Models;
using System;
using System.Collections.Generic;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);

        Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);
    }
}
```

سپس این متدهای را در EmployeeRepository پیاده‌سازی کنیم.

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>, IEmployeeRepository
    {
        public EmployeeRepository(CompanyEmployeeDbContext companyEmployeeDbContext)
            : base(companyEmployeeDbContext)
        { }

        public IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges) => FindByCondition(e =>
            e.CompanyId.Equals(companyId), trackChanges).OrderBy(e => e.Name);
    }
}
```

```

        public Employee GetEmployee(Guid companyId, Guid id, bool
trackChanges) => FindByCondition(e =>
e.CompanyId.Equals(companyId) && e.Id.Equals(id),
trackChanges).SingleOrDefault();

    }

}

در پایان اکشن متدها در EmployeesController را در GetEmployeeForCompany اضافه نمایید.

using AutoMapper;
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using Entities.DataTransferObjects;
using System.Collections.Generic;
using System;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies/{companyId}/employees")]
    [ApiController]
    public class EmployeesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;
        public EmployeesController(IRepositoryManager repository,
        ILoggerManager logger,
        IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }

        [HttpGet]
        public IActionResult GetEmployeesForCompany(Guid companyId)
        {
            var company = _repository.Company.GetCompany(companyId,
            trackChanges: false);

            if (company == null)

```

```

    {
        _logger.LogInfo($"Company with id: {companyId} doesn't
exist in the database.");
    }

    var employeesFromDb =
_repository.Employee.GetEmployees(companyId, trackChanges:
false);

    var employeesDTO =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDTO);
}

[HttpGet("{id}")]
public IActionResult GetEmployeeForCompany(Guid companyId, Guid id)
{
    var company = _repository.Company.GetCompany(companyId,
trackChanges: false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't
exist in the database.");
        return NotFound();
    }

    var employeeDb = _repository.Employee.GetEmployee(companyId,
id, trackChanges: false);

    if (employeeDb == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in
the database.");
    }

    return NotFound();
}

```

```

        var employee = _mapper.Map<EmployeeDto>(employeeDb);

        return Ok(employee);
    }
}

```

عالی شد.

ریکوئست پایین را در Postman تست کنید.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a>

The screenshot shows the Postman application interface. A GET request is made to the URL <https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a>. The response status is 200 OK, and the response body is a JSON object:

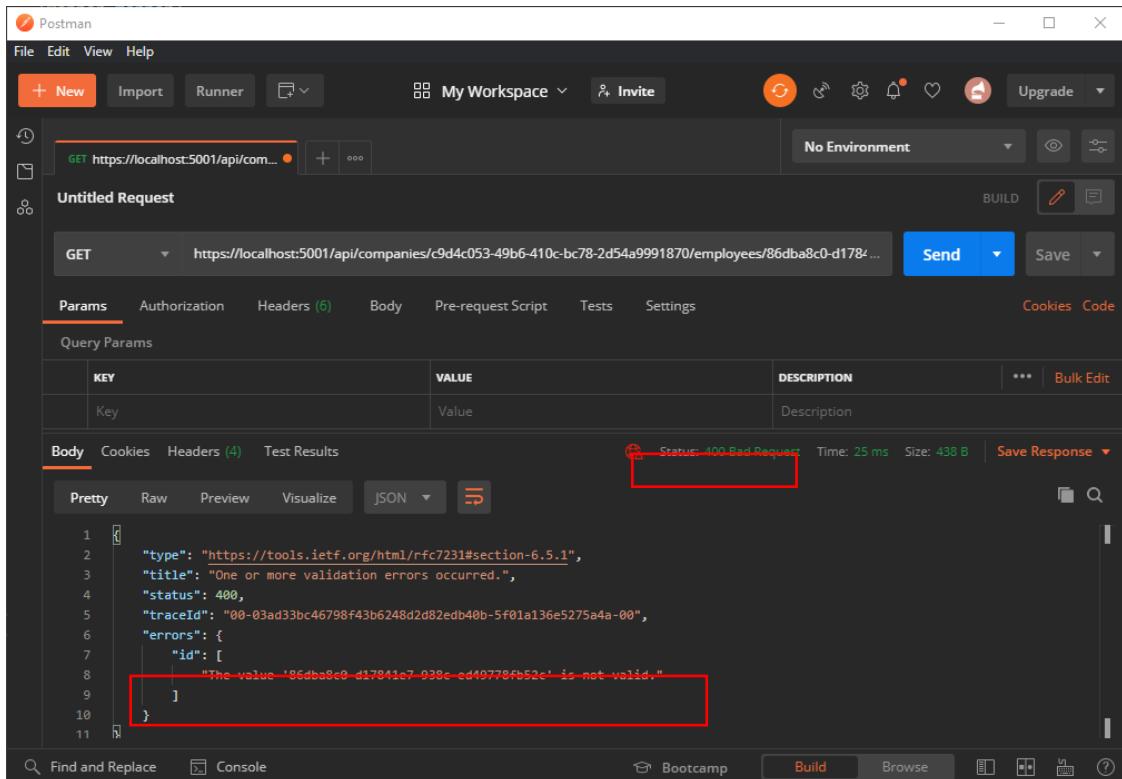
```

1  [
2   "id": "80abbca8-664d-4b20-b5de-024705497d4a",
3   "name": "Zahra Bayat",
4   "age": 26,
5   "position": "Backend developer"
6 ]

```

سپس این اکشن متده را با ریکوئست نامعتبر تست کنید.

<https://localhost:5001/api/companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/86dba8c0-d17841e7-938c-ed49778fb52c>



تا الان Response‌هایی با فرمت JSON از API گرفتیم، اما اگر بخواهیم از فرمت دیگری مانند

XML پشتیبانی کنیم چه باید کرد؟

پاسخ این سوال را Content Negotiation می‌دهد. با ما همراه باشید تا در این باره بیشتر بدانیم

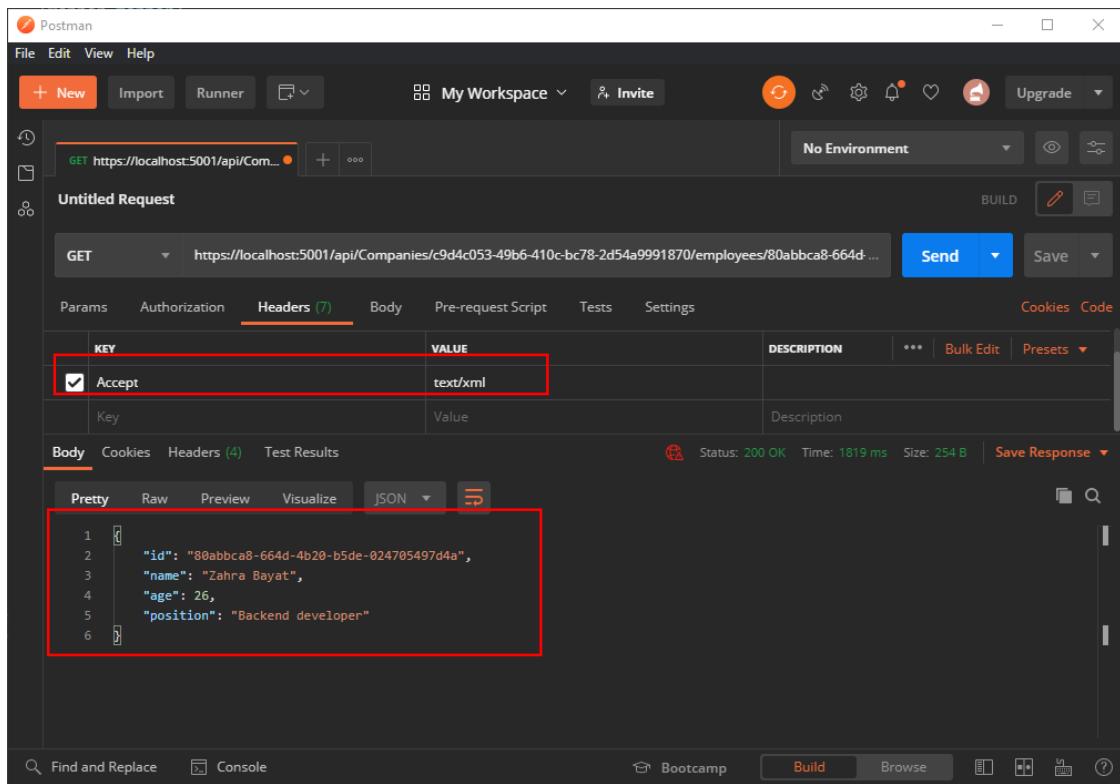
Content Negotiation چیست؟

یک ویژگی HTTP است که با آن می‌توانیم فرمت Response را مشخص کنیم. به عبارتی، پروسه‌ی انتخاب فرمت Response در زمان‌هایی که چندین فرمت در اختیار داریم، را Content Negotiation می‌گویند.

به طور پیش‌فرض ASP.NET Core Web API فرمت JSON را به عنوان نتیجه برمی‌گرداند.

ما می‌توانیم با دیدن ریسپانس تصویر پایین، این موضوع را متوجه شویم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a>



همانطور که می‌بینید، هدر Accept را بر روی Text/XML تنظیم کردیم، تا سرور را مجبور کنیم خروجی XML/Text را برگرداند؛ اما باز هم نتیجه JSON را می‌بینید. این به این دلیل است که باید فرمات‌های سرور را پیکربندی کنیم تا بتوانیم فرمات Response را تغییر دهیم.

تغییر پیکربندی Response

در یک سرور نمی‌توان به صراحت فرمات ریسپانس JSON را تغییر داد اما می‌توانید از طریق متدهای پیکربندی گزینه‌های Override AddControllers را کنید.

- اول از همه باید به یک سرور بگوییم که هدر Accept را قبول کند.
- سپس متدهای AddXmlDataContractSerializerFormatters را اضافه کنیم تا از فرمات‌های XML پشتیبانی کند.

خب متدهای AddControllers را در همانند کد پایین تغییر دهید.

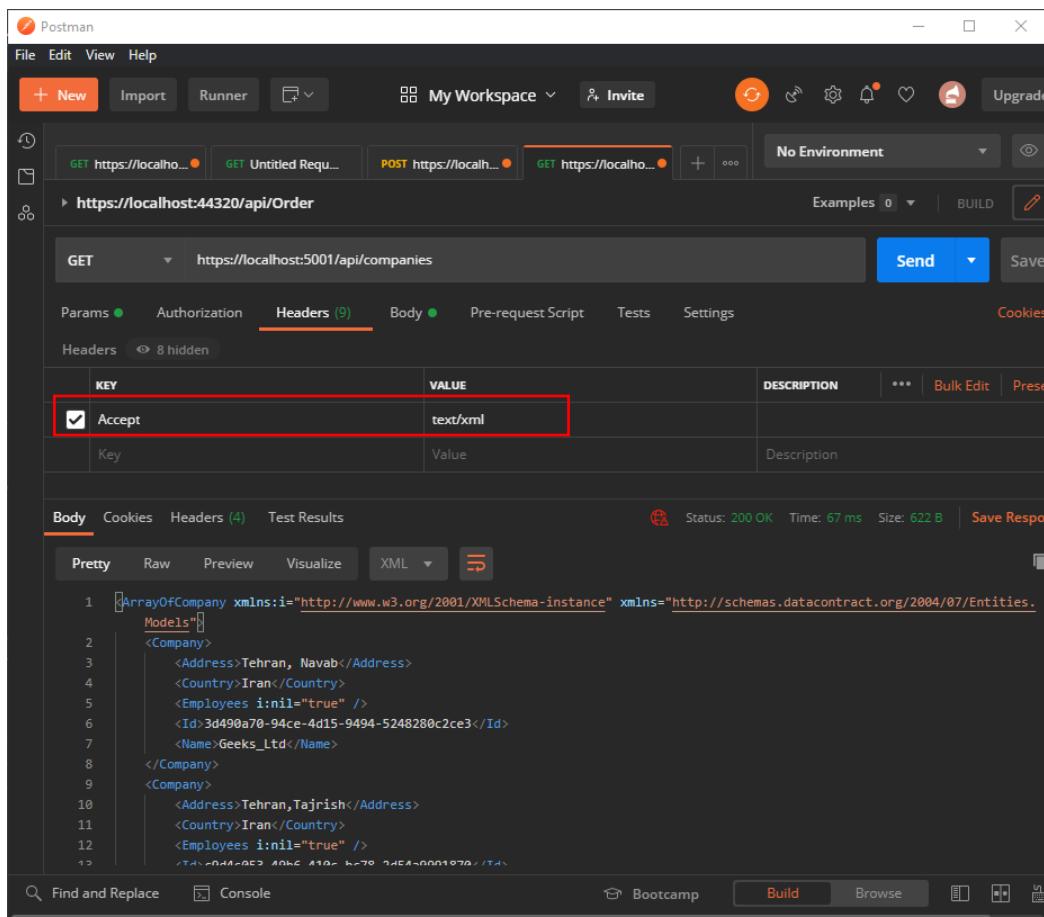
```
services.AddControllers(config => {
  config.RespectBrowserAcceptHeader = true;
}).AddXmlDataContractSerializerFormatters();
```

حالا که سرور خود را پیکربندی کردیم، اجازه دهید یکبار دیگر Content Negotiation را تست کنیم.

تست Content Negotiation

بیایید ببینیم اگر درخواست پایین را از طریق Postman انجام دهیم چه اتفاقی می‌افتد.

<https://localhost:5001/api/companies>



The screenshot shows the Postman interface with a GET request to `https://localhost:5001/api/companies`. The 'Headers' tab is active, displaying the following configuration:

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Accept	text/xml	
Key	Value	Description

The response body is shown in XML format:

```
1 <ArrayOfCompany xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/Entities.
2   <Model>
3     <Company>
4       <Address>Tehran, Navab</Address>
5       <Country>Iran</Country>
6       <Employees i:nil="true" />
7       <Id>3d490a70-94ce-4d15-9494-5248280c2ce3</Id>
8       <Name>Geeks_Ltd</Name>
9     </Company>
10    <Company>
11      <Address>Tehran,Tajrish</Address>
12      <Country>Iran</Country>
13      <Employees i:nil="true" />
14    </Company>
```

همانطور که می‌بینید Response ما XML است.

حالا با تغییر هدر Accept از `text/xml` به `text/json` می‌توانیم ریسپانسی با فرمت متفاوت بگیریم.

خب به نظر شما اگر کلاینت نوعی را درخواست کند که سرور نتواند آن فرمت را نمایش دهد چه اتفاقی می‌افتد؟

پاسخ این سوال را Media Type می‌دهد. با ما همراه باشید تا در این باره بیشتر بدانیم.

Media Type محدود کردن

در حال حاضر، سرور به طور پیشفرض Response را به نوع JSON تبدیل می‌کند. اما ما می‌توانیم با اضافه کردن یک خط به تنظیمات، این رفتار را محدود کنیم.

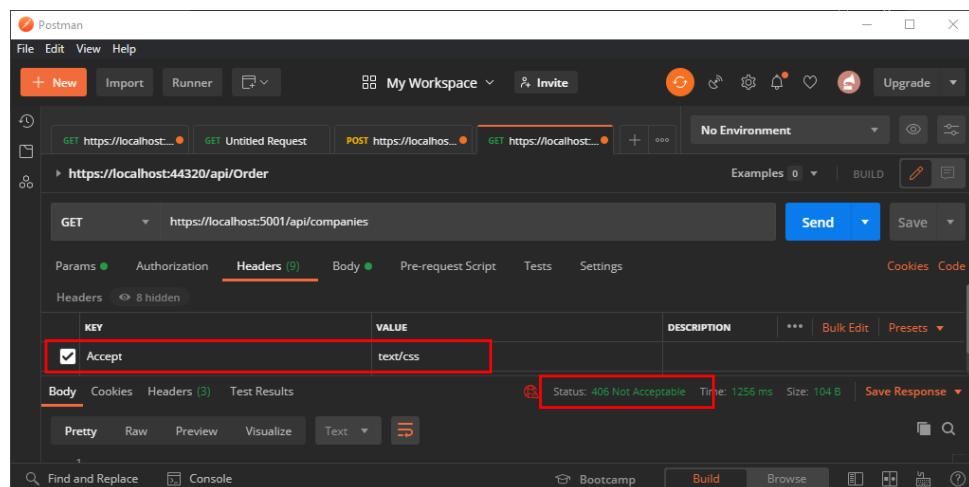
```
services.AddControllers(config => {
    config.RespectBrowserAcceptHeader = true;
    config.ReturnHttpNotAcceptable = true;
}).AddXmlDataContractSerializerFormatters();
```

بررسی کد :

- ما گزینه AddControllers را به تنظیمات ReturnHttpNotAcceptable = true اضافه کردیم تا به سرور بگوییم که اگر کلاینت، فرمتی که سرور پشتیبانی نمی‌کند را خواست باید 406 Not Acceptable را برگردانی.
- این باعث می‌شود برنامه ما محدودتر شود و کلاینت فقط انواعی که سرور پشتیبانی می‌کند را درخواست نماید. کد 406 برای این منظور ایجاد شده است.

حالا برای تست، هدر Accept را بر روی text/css تنظیم کنیم و API پایین را صدا بزنیم.

<https://localhost:5001/api/companies>



The screenshot shows the Postman application interface. A GET request is being made to `https://localhost:5001/api/companies`. In the 'Headers' tab, there is a row for the 'Accept' header with the value 'text/css'. The status bar at the bottom right of the interface displays 'Status: 406 Not Acceptable'.

همانطور که انتظار می‌رفت، هیچ Response Body وجود ندارد و تنها چیزی که می‌گیریم 406 Not Acceptable است.

ایجاد فرمت سفارشی

اگر بخواهیم API، فرمتی که در کادر نیست را هم پشتیبانی کند چه باید کرد؟

خوب خبختانه ASP.NET Core از ایجاد فرمتهای سفارشی پشتیبانی می‌کند. برای اینکه بتوانیم فرمت سفارشی داشته باشیم می‌توانیم از متدهای پایین استفاده کنیم.

- برای ایجاد Input Formatter باید یک کلاس ایجاد کنید که از کلاس TextInputFormatter ارث بری کند. و برای ایجاد Output Formatter باید یک کلاس ایجاد کنید که از کلاس TextOutputFormatter ارث بری کند.
- سپس به همان روشی که برای فرمت XML انجام دادیم باید این کلاس‌ها را به مجموعه OutputFormatters و InputFormatters اضافه کنید.
حالا باید یک قالب CSV سفارشی برای مثالمان آماده کنیم.

خب می‌خواهیم یک فرمت Response، برای برگرداندن لیست شرکت‌ها در قالب CSV ایجاد کنیم بنابراین برای پیاده‌سازی، به یک کلاس نیاز داریم که از TextOutputFormatter ارث بری کند.

باید یک کلاس CsvOutputFormatter در فolder Infrastructure پروژه اصلی اضافه کنیم.

```
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.Net.Http.Headers;
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Infrastructure
{
    public class CsvOutputFormatter : TextOutputFormatter
    {
        public CsvOutputFormatter()
        {
```

```

SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/csv"));
    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
protected override bool CanWriteType(Type type)
{
    if (typeof(CompanyDto).IsAssignableFrom(type) ||
        typeof(IEnumerable<CompanyDto>).IsAssignableFrom(type))
    {
        return base.CanWriteType(type);
    }

    return false;
}

public override async Task
WriteResponseBodyAsync(OutputFormatterWriteContext
context, Encoding selectedEncoding)
{
    var response = context.HttpContext.Response;
    var buffer = new StringBuilder();

    if (context.Object is IEnumerable<CompanyDto>)
    {
        foreach (var company in
        (IEnumerable<CompanyDto>)context.Object)
        {
            FormatCsv(buffer, company);
        }
    }
    else
    {
        FormatCsv(buffer, (CompanyDto)context.Object);
    }

    await response.WriteAsync(buffer.ToString());
}
private static void FormatCsv(StringBuilder buffer, CompanyDto
company)
{
    buffer.AppendLine($"{company.Id},{company.Name},{company.Ful
lAddress}");
}

```

```
    }  
}  
}
```

بررسی کد :

- در Constructor این کلاس، مشخص کردیم که این فرمت باید کدام MediaType را Encode و Parse کند.
- متد CanWriteType، متدهیست که مشخص می‌کند آیا نوع CompanyDto را می‌توان با Serializer نوشت یا نه؟
- متد WriteResponseBodyAsync ریسپانس را ایجاد می‌کند.
- و در پایان، متدهی AddFormatCsv می‌بینیم که فرمتهای Response را درست می‌کند.

حالا فقط باید فرمت جدید را به لیست OutputFormatters اضافه کنیم. پس اکستنشن متد پایین را در کلاس ServicesExtensions اضافه نمایید

```
public static IMvcBuilder AddCustomCSVFormatter(this IMvcBuilder builder)  
=>  
builder.AddMvcOptions(config => config.OutputFormatters.Add(new  
CsvOutputFormatter));
```

حالا این متد را در AddControllers صدا بزنید.

```
services.AddControllers(config => {  
config.RespectBrowserAcceptHeader = true;  
config.ReturnHttpNotAcceptable = true;  
}).AddXmlDataContractSerializerFormatters()  
.AddCustomCSVFormatter();
```

برنامه را اجرا و text/csv را به عنوان مقدار هدر Accept قرار دهید.

<https://localhost:5001/api/companies>

The screenshot shows the Postman application interface. A GET request is made to `https://localhost:5001/api/companies`. In the Headers section, there is a row for `Accept` with the value `text/csv`. The response body contains the following CSV data:

1	3d490a70-94ce-4d15-9494-5248280c2ce3,"Geeks_Ltd,"Tehran, Navab Iran"
2	c9d4c053-49b6-410c-bc78-2d54a9991870,"Raveshmand_Ltd,"Tehran,Tajrish Iran"

خیلی عالی شد، همه چیز خیلی خوب کار می‌کند.

در این فصل ریکوئست‌های GET را اضافه کردیم. در فصل بعد ریکوئست‌های POST، PUT و DELETE را بررسی می‌کنیم.

فصل ششم : DELETE و POST , PUT

آنچه خواهید آموخت:

➤ ایجاد، حذف و آپدیت Resource

➤ چیست؟ Model Binding

مدیریت Post Request

Post یکی از HTTP Methods است که برای ایجاد یا آپدیت بخشی از Resource استفاده می‌شود. اگر Resource دارای Id باشد، این فعل یک Resource را آپدیت می‌کند و گرنه یک Resource جدید را ایجاد خواهد کرد.

گام اول : اتریبوت اکشن متد GetCompany در CompaniesController را مانند کد پایین اصلاح کنید، چون در ادامه به این کد نیاز داریم.

```
[HttpGet("{id}", Name = "CompanyById")]
```

گام دوم : یک کلاس با نام CompanyForCreationDto ایجاد DataTransferObjects در فolder CompanyForCreationDto کنید.

```
namespace Entities.DataTransferObjects
{
    public class CompanyForCreationDto
    {
        public string Name { get; set; }
        public string Address { get; set; }
        public string Country { get; set; }
    }
}
```

همانطور که می‌بینید این کلاس تقریباً مشابه کلاس Company است با این تفاوت که Id ندارد. در برخی پروژه‌ها، کلاس DTO ورودی و خروجی یکی است؛ اما توصیه می‌کنیم این دو را از هم جدا کنید تا نگهداری و ریفکتور کد، ساده‌تر شود. علاوه بر این، وقتی صحبت از اعتبارسنجی را شروع کنیم، نیازی به اعتبارسنجی آبحکت خروجی نیست اما آبحکت ورودی باید اعتبارسنجی شود.

خب حالا باید متد CreateCompany را در اینترفیس ICompanyRepository اضافه کنیم.

```
using Entities.Models;
using System.Collections.Generic;
using System;

namespace Contracts.IServices
```

```

{
    public interface ICompanyRepository
    {
        IEnumerable<Company> GetAllCompanies(bool trackChanges);
        Company GetCompany(Guid companyId, bool trackChanges);
        void CreateCompany(Company company);
    }
}

```

بعد از تغییر اینترفیس، حالا ما این اینترفیس را در کلاس CompanyRepository پیاده‌سازی کنیم.

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext)
            : base(companyEmployeeDbContext)
        { }

        public IEnumerable<Company> GetAllCompanies(bool
            trackChanges) =>
            FindAll(trackChanges)
                .OrderBy(c => c.Name)
                .ToList();

        public Company GetCompany(Guid companyId, bool trackChanges) =>
            FindByCondition(c => c.Id.Equals(companyId), trackChanges)
                .SingleOrDefault();
    }
}

```

```

    public void CreateCompany(Company company) => Create(company);

}

}

```

نکته!!

ما `Id` را برای شرکت خود ایجاد نمی کنیم این کار توسط EF Core انجام می شود. تمام کاری که انجام می دهیم این است `State` شرکت را بر روی `Added` قرار دهیم.

قبل از اینکه یک اکشن متده جدید در `CompaniesController` اضافه کنیم باید یک Rule دیگر برای مپ کردن آبجکت `CompanyForCreationDto` و `Company` بنویسیم. بنابراین کد پایین را در کلاس `MappingProfile` اضافه نمایید.

```
CreateMap<CompanyForCreationDto, Company>();
```

خب حالا باید اکشن متده `CreateCompany` ایجاد شود.

```

[HttpPost]
public IActionResult CreateCompany([FromBody] CompanyForCreationDto
    company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from client
            is null.");
        return BadRequest("CompanyForCreationDto object is null");
    }

    var companyEntity = _mapper.Map<Company>(company);
    _repository.Company.CreateCompany(companyEntity);
    _repository.Save();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id
        },
        companyToReturn);
}

```

کدهای CompaniesController

```
using AutoMapper;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Microsoft.AspNetCore.Mvc;
using Entities.Models;
using System;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public CompaniesController(IRepositoryManager repository,
        ILoggerManager logger,
        IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }

        [HttpGet]
        public IActionResult GetCompanies()
        {
            var companies =
                _repository.Company.GetAllCompanies(trackChanges: false);

            var companiesDto =
                _mapper.Map<IEnumerable<CompanyDto>>(companies);

            return Ok(companiesDto);
        }

        [HttpGet("{id}", Name = "CompanyById")]
    }
}
```

```

public IActionResult GetCompany(Guid id)
{
    var company = _repository.Company.GetCompany(id, trackChanges:
false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {id} doesn't exist in
the database.");
    }

    return NotFound();
}

else
{
    var companyDto = _mapper.Map<CompanyDto>(company);

    return Ok(companyDto);
}
}

[HttpPost]
public IActionResult CreateCompany([FromBody]
    CompanyForCreationDto company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from
client is null.");

        return BadRequest("CompanyForCreationDto object is null");
    }

    var companyEntity = _mapper.Map<Company>(company);
    _repository.Company.CreateCompany(companyEntity);
    _repository.Save();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id =
companyToReturn.Id },
    companyToReturn);
}

```

```

        }
    }
}

```

قبل از توضیح بدنه کد باید یکبار این اکشن مت را با Postman تست کنیم.

<https://localhost:5001/api/companies>

Body:

The screenshot shows the Postman interface with the following details:

- Request Method:** POST
- Request URL:** https://localhost:5001/api/companies
- Body Content:**

```

{
    "name": "Fara_Ltd",
    "address": "Tehran, Vanak Iran",
    "country": "Iran"
}

```
- Response Status:** 201 Created
- Response Body:**

```

{
    "id": "1d86352f-8674-4096-b62f-08d891db2f40",
    "name": "Fara_Ltd",
    "fullAddress": "Tehran, Vanak Iran"
}

```

بررسی کد:

بگذارید در مورد کدهای این اکشن مت و این ریکوئست کمی بیشتر صحبت کنیم.

- اکشن مت CreateCompany یک اtribut [HttpPost] دارد که باعث می شود، تنها با ریکوئست POST بتوانیم آن را صدا بزنیم.
- پارامتر company از سمت کلاینت می آید پس نیاز به اtribut [FromBody] داریم.

- چون پارامتر ورودی از URI گرفته نمی‌شود پس باید پارامتر را از Body ریکوئست بگیریم.
 - از آنجاییکه پارامتر company از سمت کلاینت می‌اید پس ممکن است مقادیر معتبری نداشته باشد و نتوان آن را Deserialized کرد. در نتیجه باید این پارامتر را اعتبارسنجی کنیم تا مقدار آن null نباشد.
 - بعد از اعتبار سنجی این پارامتر را برای ایجاد یک Company مپ کنیم و متدهای CreateCompany را صدا بزنیم.
 - برای ذخیره Entity در دیتابیس، متدهای Save را استفاده می‌کنیم.
 - پس از ذخیره Company در دیتابیس، باید آبجکت company را به CompanyDto مپ کنیم و به کلاینت برگردانیم.
 - آخرین موردی که باید به آن اشاره کنم متدهای CreatedAtRoute است.
 - یک 201 Status Code را بر می‌گرداند.

این متدها Response Body را با یک آبجکت Company پر می‌کند و سپس در Headers مربوط به Response را بر می‌گردانند. اگر Location هدر، مکان Get را پرداختی می‌کند، این آدرس را بررسی کنید، لینکی، پرای واکنشی اطلاعات شرکت ایجاد شده است.

Body	Cookies	Headers (5)	Test Results	 Status: 201 Created	Time: 3.25 s	Size: 336 B	Save Response ▾
KEY				VALUE			
Date	ⓘ			Fri, 27 Nov 2020 08:52:32 GMT			
Content-Type	ⓘ			application/json; charset=utf-8			
Server	ⓘ			Kestrel			
Transfer-Encoding	ⓘ			chunked			
Location	ⓘ			https://localhost:5001/api/companies/a980efb4-b338-4f18-8100-08d892b1c...			

همانطور که قبلاً گفتم Idempotent - Post نیست بنابراین وقتی ریکوئست POST را ارسال کنیم، می‌بینیم که یک Resource جدید در دیتابیس ایجاد شده است. حالا اگر این ریکوئست را چندین بار تکرار کنید، مطمئناً برای هر ریکوئست یک آجکت جدید دریافت خواهد کرد که هر کدام Id متفاوت دارد.

خوب حالا بساید Child Resource ایجاد کنیم.

ایجاد Child Resource

در اکشن متدهم CreateCompany، برای ایجاد یک شرکت به یک DTO نیاز داشتیم حالا برای ایجاد یک کارمند هم ما باید همین کار را انجام دهیم.

در فolder DataTransferObjects ایجاد کنید. EmployeeForCreationDto یک کلاس با نام

```
namespace Entities.DataTransferObjects
{
    public class EmployeeForCreationDto
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public string Position { get; set; }
    }
}
```

ما پرپرتی Id و CompanyId نداریم چون میخواهیم Id را در سمت سرور ایجاد و را از طریق مسیر بفرستیم.

```
[Route("api/companies/{companyId}/employees")]
```

مرحله بعدی ایجاد تغییراتی در اینترفیس IEmployeeRepository است.

```
using Entities.Models;
using System;
using System.Collections.Generic;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);

        Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);

        void CreateEmployeeForCompany(Guid companyId, Employee employee);
    }
}
```

خب حالا باید این اینترفیس را پیادهسازی کنیم.

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
        IEmployeeRepository
    {
        public EmployeeRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext) : base(companyEmployeeDbContext)
        {
        }

        public IEnumerable<Employee> GetEmployees(Guid companyId, bool
            trackChanges) => FindByCondition(e =>
            e.CompanyId.Equals(companyId), trackChanges).OrderBy(e => e.Name);

        public Employee GetEmployee(Guid companyId, Guid id, bool
            trackChanges) => FindByCondition(e =>
            e.CompanyId.Equals(companyId) && e.Id.Equals(id),
            trackChanges).SingleOrDefault();

        public void CreateEmployeeForCompany(Guid companyId, Employee
            employee)
        {
            employee.CompanyId = companyId;
            Create(employee);
        }
    }
}

```

چون می خواهیم آبجکت EmployeeDTO را در اکشن بگیریم پس در کلاس MappingProfile به یک Mapping Rule نیاز داریم.

```
CreateMap<EmployeeForCreationDto, Employee>();
```

حالا نوبت به ایجاد یک اکشن متده جدید در EmployeesController است.

ابتدا Namespace پایین را در این کنترلر اضافه کنید.

```
using Entities.Models;

[HttpPost]
public IActionResult CreateEmployeeForCompany(Guid companyId, [FromBody]
EmployeeForCreationDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForCreationDto object sent from client is
null.");

        return BadRequest("EmployeeForCreationDto object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");

        return NotFound();
    }

    var employeeEntity = _mapper.Map<Employee>(employee);
    _repository.Employee.CreateEmployeeForCompany(companyId,
employeeEntity);
    _repository.Save();

    var employeeToReturn = _mapper.Map<EmployeeDto>(employeeEntity);

    return
CreatedAtRoute("GetEmployeeForCompany",
new
{
```

```

        companyId,
        id = employeeToReturn.Id
    },
    employeeToReturn);
}

```

بررسی کد :

این اکشن متد با **CreateCompany** تفاوت‌هایی دارد :

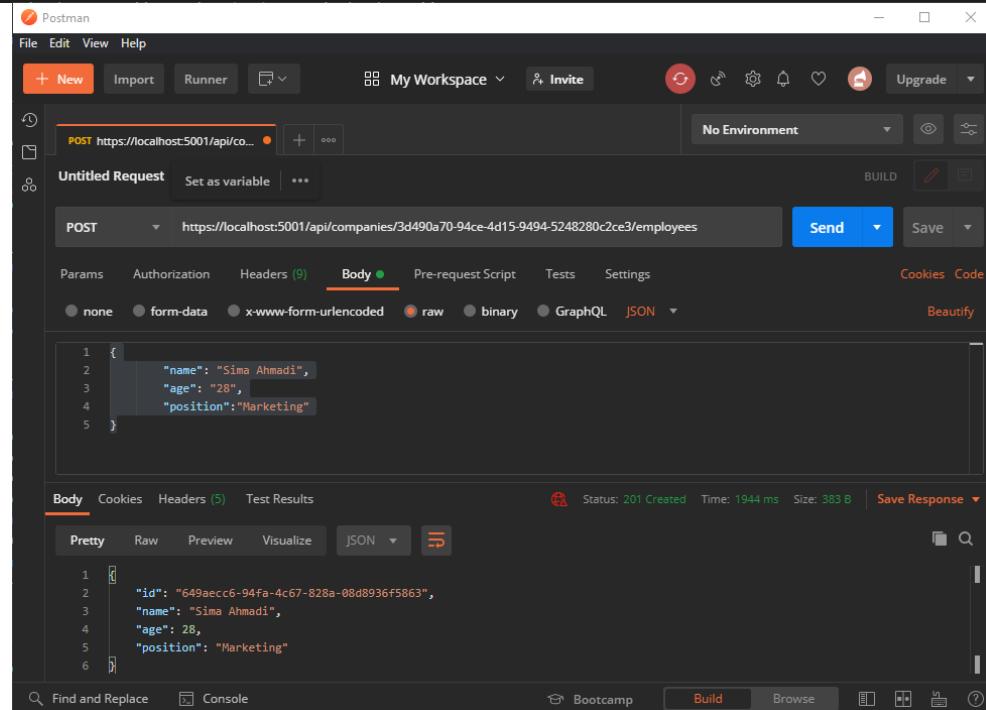
- اولین اینکه باید بررسی کنیم آیا شرکت موردنظر در دیتابیس وجود دارد یا خیر.
- تفاوت دوم عبارت `return` است که الان باید دو پارامتر برگرداند.

قبل از تست باید اtribut بالای اکشن متد `GetEmployeeForCompany` را از `[HttpGet]` به `[HttpGet(Name = "GetEmployeeForCompany")]` تغییر دهید.
`[HttpGet(Name = "GetEmployeeForCompany")]`
حالا باید این مورد را با هم تست کنیم.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3/employees>

body:

```
{
    "name": "Sima Ahmadi",
    "age": "28",
    "position": "Marketing"
}
```



ایجاد Parent همراه با یک Child Resource

در اپلیکیشن‌ها موقعیت‌هایی هست که بخواهیم یک Parent را همراه با Child‌ها ایش کنیم. به طور مثال :

به جای اینکه از ریکوئست‌های متعدد برای درج تک تک Child‌ها استفاده کنیم، می‌توانیم در همان ریکوئست درج Child‌ها، Parent را هم درج کنیم.

پس اولین کاری که باید انجام دهیم این است که در کلاس CompanyForCreationDto تعییراتی انجام شود.

```
using System.Collections.Generic;
namespace Entities.DataTransferObjects
{
    public class CompanyForCreationDto
    {
        public string Name { get; set; }
        public string Address { get; set; }
        public string Country { get; set; }
        public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
    }
}
```

نیازی نیست که منطق اکشن متدهای داخل کنترلر و منطق ریپازیتوری را تغییر دهیم پس باید با هم این را هم تست کنیم.

<https://localhost:5001/api/companies>

body:

```
{
    "name": "Pardazesh",
    "address": "Tehran, Tohid",
    "country": "Iran",
    "employees": [
        {
            "name": "Sana Sadeghi",
            "age": 28,
            "position": "IT"
        },
        {
            "name": "Reza Zargham",
            "age": 20,
        }
    ]
}
```

```

        "position": "IT"
    }
}

```

Postman

File Edit View Help

POST https://localhost:5001/api/co... + ...

No Environment

Untitled Request

POST https://localhost:5001/api/companies Send Save

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies C...

Body (JSON)

```

1  {
2      "name": "Pardazesh",
3      "address": "Tehran, Tohid",
4      "country": "Iran",
5      "employees": [
6          {
7              "name": "Sana Sadeghi",
8              "age": 28,
9              "position": "IT"
10         },
11         {
12             "name": "Reza Zargham",
13             "age": 20,
14             "position": "IT"
15         }
16     ]
17 }
18

```

Body Cookies Headers (5) Test Results

Status: 201 Created Time: 1301 ms Size: 338 B Save Response

Pretty Raw Preview Visualize JSON

```

1  [
2      "id": "20afe396-278f-4133-2f26-08d8b7c1a9fe",
3      "name": "Pardazesh",
4      "fullAddress": "Tehran, Tohid Iran"
5 ]

```

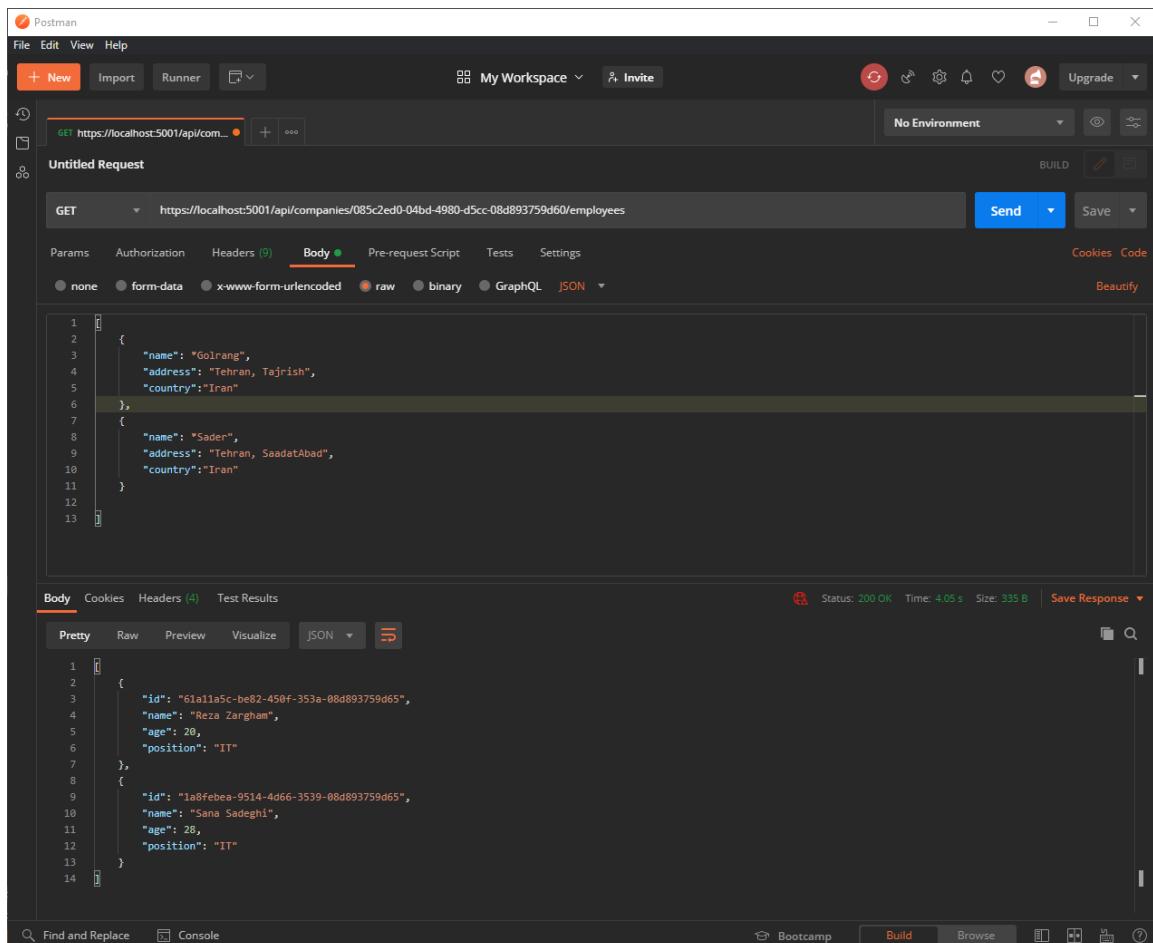
Find and Replace Console Bootcamp Build Browse

همانطور که می‌بینید این شرکت با موفقیت ایجاد شد.

حالا می‌توانیم لینک Location را از برگه Headers کپی کرده و سپس قسمت /employees را به این لینک اضافه و در تب دیگر Postman پیست کنیم.

Body	Cookies	Headers (5)	Test Results
Date ⓘ			Sat, 28 Nov 2020 08:14:48 GMT
Content-Type ⓘ			application/json; charset=utf-8
Server ⓘ			Kestrel
Transfer-Encoding ⓘ			Set as variable ...
Location ⓘ			https://localhost:5001/api/companies/085c2ed0-04bd-4980-d5cc-08d893759d60/employees

<https://localhost:5001/api/companies/085c2ed0-04bd-4980-d5cc-08d893759d60/employees>



ایجاد مجموعه‌ای از Resource‌ها

تا حالا توانستیم یک Resource ایجاد کنیم. لما برای ایجاد مجموعه از Resource‌ها چه باید کرد؟ در این قسمت می‌خواهیم این موضوع را پیاده‌سازی کنیم.

اگر به اکشن متدهای CreateCompany در CompaniesController نگاه کنید، می‌بینید که به مسیر CompanyById اشاره دارد (اکشن متدهای GetCompany).

```

[HttpPost]
public IActionResult CreateCompany([FromBody] CompanyForCreationDto
    company)
{
    //...
    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id
    },
        companyToReturn);
}

```

این یعنی، بعد از ایجاد Company می‌توانیم اطلاعات درج شده در دیتابیس را با استفاده از متدهای GetCompanyCollection برگردانیم. بنابراین قبل از اینکه کدهای GetCompanyCollection را بنویسیم، بهتر است اکشن متدهای برای واکشی اطلاعات شرکت‌ها داشته باشیم.

اول از همه باید متدهای GetByIds را به اینترفیس ICompanyRepository اضافه کنیم.

```
using Entities.Models;
using System.Collections.Generic;
using System;

namespace Contracts.IServices
{
    public interface ICompanyRepository
    {
        IEnumerable<Company> GetAllCompanies(bool trackChanges);
        Company GetCompany(Guid companyId, bool trackChanges);
        void CreateCompany(Company company);
        IEnumerable<Company> GetByIds(IEnumerable<Guid> ids, bool
trackChanges);
    }
}
```

حالا نوبت به پیاده‌سازی این اینترفیس است.

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
companyEmployeeDbContext): base(companyEmployeeDbContext)
        {

```

```

    }

    public IEnumerable<Company> GetAllCompanies(bool
        trackChanges) =>
        FindAll(trackChanges)
            .OrderBy(c => c.Name)
            .ToList();

    public Company GetCompany(Guid companyId, bool trackChanges) =>
        FindByCondition(c => c.Id.Equals(companyId), trackChanges)
            .SingleOrDefault();

    public void CreateCompany(Company company) => Create(company);

    public IEnumerable<Company> GetByIds(IEnumerable<Guid> ids,
        bool trackChanges) =>
        FindByCondition(x => ids.Contains(x.Id), trackChanges)
            .ToList();
}

}

```

خب یک اکشن متدهای پایین را به CompaniesController اضافه نمایید.

پایین را قبل از نوشتن اکشن متدهای کنترلر اضافه کنید.

```

using System.Linq;

    : GetCompanyCollection اکشن متدهای

[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public IActionResult GetCompanyCollection(IEnumerable<Guid> ids)
{
    if (ids == null)
    {
        _logger.LogError("Parameter ids is null");
        return BadRequest("Parameter ids is null");
    }
}

```

```

var companyEntities = _repository.Company.GetByIds(ids, trackChanges:
false);

if (ids.Count() != companyEntities.Count())
{
    _logger.LogError("Some ids are not valid in a collection");

    return NotFound();
}

var companiesToReturn =
_mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

return Ok(companiesToReturn);
}

```

این اکشن متدهای ساده است. حالا بیایید اکشن CreateCompanyCollection را ایجاد کنیم.

```

[HttpPost("collection")]
public IActionResult CreateCompanyCollection([FromBody]
IEnumerable<CompanyForCreationDto> companyCollection)
{
    if (companyCollection == null)
    {
        _logger.LogError("Company collection sent from client is null.");

        return BadRequest("Company collection is null");
    }

    var companyEntities =
_mapper.Map<IEnumerable<Company>>(companyCollection);

    foreach (var company in companyEntities)
    {
        _repository.Company.CreateCompany(company);
    }

    _repository.Save();

    var companyCollectionToReturn =
_mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

```

```

var ids = string.Join(",",
    companyCollectionToReturn.Select(c => c.Id));

return CreatedAtRoute("CompanyCollection", new { ids },
companyCollectionToReturn);
}

```

بررسی کد :

- در این اکشن متده بررسی می‌کنیم که آیا CompanyCollection برابر Null است یا خیر؟
 - اگر Null باشد یک BadRequest بر می‌گردانیم.
 - اگر اینگونه نباشد ما این CompanyCollection را مپ و همه المنتهای آن را در دیتابیس ذخیره می‌کنیم.
 - در پایان برای واکشی شرکت‌هایی که ایجاد کردیم تمام Idها را در یک String (که با کاما جدا شده را) می‌ریزیم و به اکشن متده Get می‌فرستیم.
- حالا می‌توانیم این اکشن متده را تست کنیم.

<https://localhost:5001/api/companies/collection>

body:

```
[
  {
    "name": "Golrang",
    "address": "Tehran, Tajrish",
    "country": "Iran"
  },
  {
    "name": "Sader",
    "address": "Tehran, SaadatAbad",
    "country": "Iran"
  }
]
```

The screenshot shows the Postman interface with a successful POST request to `https://localhost:5001/api/companies/collection`. The request body contains two company objects:

```

1 [
2   {
3     "name": "Golrang",
4     "address": "Tehran, Tajrish",
5     "country": "Iran"
6   },
7   {
8     "name": "Sader",
9     "address": "Tehran, SaadatAbad",
10    "country": "Iran"
11 }
12 ]

```

The response shows two new company documents created with IDs and full addresses:

```

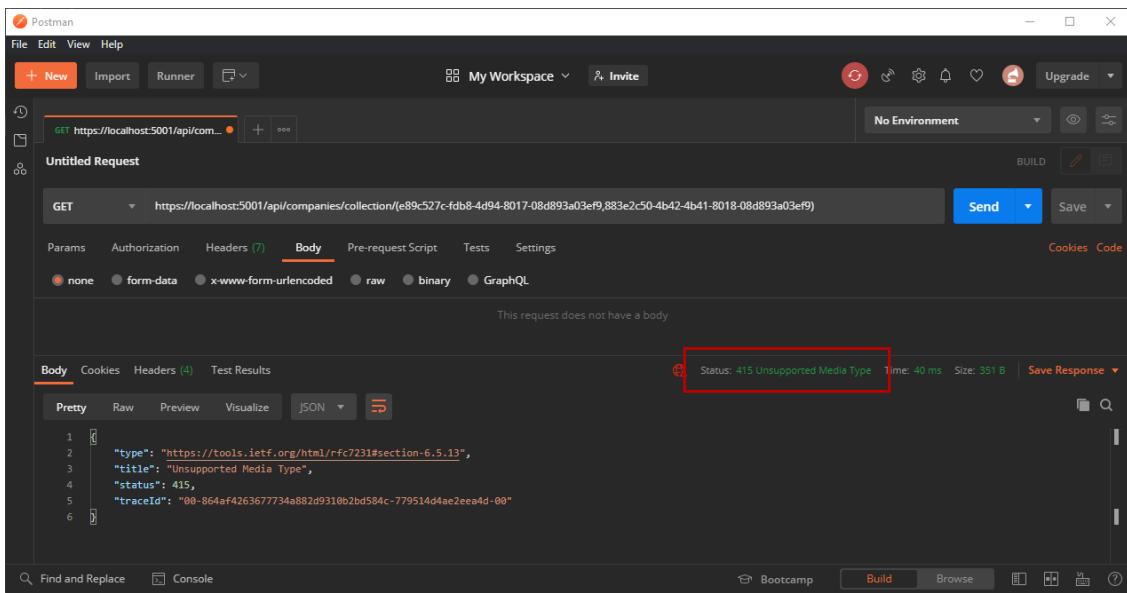
1 [
2   {
3     "id": "0544faed-ba78-4861-d50b-08d8b7cc2a00",
4     "name": "Golrang",
5     "fullAddress": "Tehran, Tajrish Iran"
6   },
7   {
8     "id": "5d81b465-203e-42a2-d50c-08d8b7cc2a00",
9     "name": "Sader",
10    "fullAddress": "Tehran, SaadatAbad Iran"
11 }
12 ]

```

خیلی عالی شد اجازه دهید تب Headers را با هم چک کنیم.

KEY	VALUE
Date ⓘ	Sat, 28 Nov 2020 13:19:32 GMT
Content-Type ⓘ	application/json; charset=utf-8
Server ⓘ	Kestrel
Transfer-Encoding ⓘ	Set as variable ...
Location ⓘ	https://localhost:5001/api/companies/collection/(43bae00f-3ae3-49a7-8015-08d893a03ef9,e935296e-f99c-4205-8016-08d893a03ef9)

همانطور که می‌بینید اینجا یک لینک داریم که با آن می‌توان شرکت‌هایی که جدیداً اضافه شده را ببینیم.



اما همانطور که می‌بینید نتیجه‌ی این اکشن متده است. چون API نمی‌تواند نوع رشته را به آرگومان `IEnumerable<Guid>` مپ کند.

خب در قسمت بعدی ما این مشکل را با Model Binding سفارشی حل می‌کنیم.

چیست؟ Model Binding

داده‌ها را از ریکوئست می‌گیرد و به پارامترهای اکشن متده انتقال می‌دهد. همیشه قبل از اکشن متده اجرا و بررسی می‌کند : آیا مقادیر اتصالی معتبر است یا خیر؟

برای ایجاد Model Binding سفارشی، ابتدا در فolder Infrastructure یک فolder Model Binders ایجاد و در آن یک کلاس جدید با نام `ArrayModelBinder` اضافه نمایید.

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
using System;
using System.ComponentModel;
using System.Linq;
using System.Reflection;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Infrastructure.ModelBinders
{
    public class ArrayModelBinder : IModelBinder
```

```

{
    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (!bindingContext.ModelMetadata.IsEnumerableType)
        {
            bindingContext.Result = ModelBindingResult.Failed();

            return Task.CompletedTask;
        }

        var providedValue = bindingContext.ValueProvider
            .GetValue(bindingContext.ModelName)
            .ToString();

        if (string.IsNullOrEmpty(providedValue))
        {
            bindingContext.Result = ModelBindingResult.Success(null);

            return Task.CompletedTask;
        }

        var genericType =
            bindingContext.ModelType.GetTypeInfo().GenericTypeArguments[0];

        var converter = TypeDescriptor.GetConverter(genericType);
        var objectArray = providedValue.Split(new[] { "," },
            StringSplitOptions.RemoveEmptyEntries)
            .Select(x => converter.ConvertFromString(x.Trim()))
            .ToArray();

        var guidArray = Array.CreateInstance(genericType,
            objectArray.Length);
        objectArray.CopyTo(guidArray, 0);
        bindingContext.Model = guidArray;

        bindingContext.Result =
            ModelBindingResult.Success(bindingContext.Model);

        return Task.CompletedTask;
    }
}

```

```

        }
    }
}

```

بررسی کد :

- چون برای نوع `IEnumerable` داریم یک Model Binder ایجاد می‌کنیم، پس باید بررسی کنیم که پارامتر ما از نوع `IEnumerable` است یا خیر؟
- در مرحله بعد با استفاده از متده `GetValue`، مقدار رشته GUID را واکشی می‌کنیم.
- حالا در صورتیکه مقدار بدست آمده Null یا `Empty` باشد، باید Null را برگردانیم در غیر اینصورت، ادامه می‌دهیم.
- خب حالا با کمک `Reflection`، نوعی که `IEnumerable` از آن تشکیل شده را، در متغیر `genericType` سپس بررسی می‌کنیم که نوع متغیر `genericType` چیست و برای آن نوع `Converter` را ایجاد می‌کنیم.
- پس از آن باید آرایه‌ای از نوع `objectArray` که شامل مقادیر GUID (که به API رسیده است) را ایجاد کنیم.
- حالا آرایه‌ای از نوع `GuidArray` ایجاد و سپس تمام مقادیر را از `objectArray` به `GuidArray` کپی می‌کنیم.
- در پایان باید این مقدار را به `bindingContext` اختصاص دهیم.

حالا باید یک تغییر جزئی در اکشن `GetCompanyCollection` ایجاد کنیم تا `ArrayModelBinder` قبل از اجرای اکشن متده شروع به کار کند.

ابتدا Namespace `CompaniesController` را در `CompaniesController` اضافه کنید.

```
using CompanyEmployee.API.Infrastructure.ModelBinders;
```

سپس اکشن متده `GetCompanyCollection` را تغییر دهید.

```
[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public IActionResult GetCompanyCollection([ModelBinder(BinderType =
typeof(ArrayModelBinder))] IEnumerable<Guid> ids)
{
```

```

if (ids == null)
{
    _logger.LogError("Parameter ids is null");

    return BadRequest("Parameter ids is null");
}

var companyEntities = _repository.Company.GetByIds(ids, trackChanges:
    false);

if (ids.Count() != companyEntities.Count())
{
    _logger.LogError("Some ids are not valid in a collection");

    return NotFound();
}

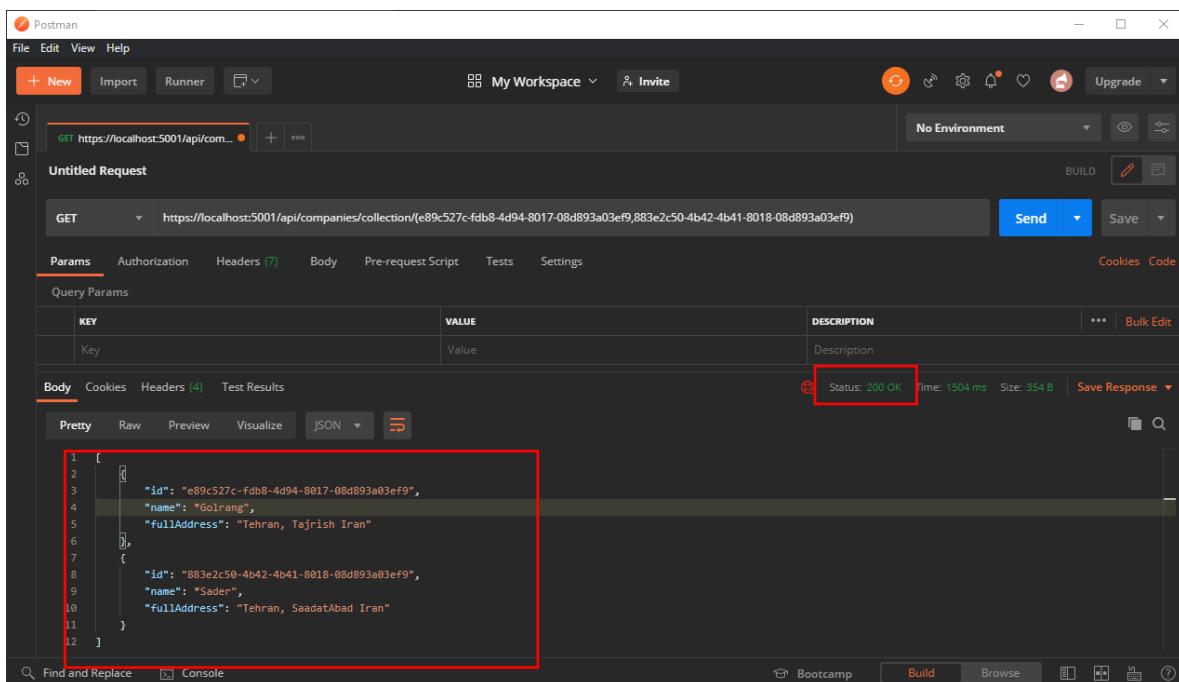
var companiesToReturn =
    _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

return Ok(companiesToReturn);
}

```

این رشته ارسال شده به API را به نوع `IEnumerable<Guid>` تبدیل و سپس این اکشن متده را اجرا می‌شود.

یکبار دیگر ریکوئست قبلی را با Postman اجرا کنید.



ایجاد اکشن متدهای Delete

- بایایید با حذف یک اکشن Delete Child Resource را پیاده‌سازی کنیم.
- متدهای DeleteEmployee را به اینترفیس IEmployeeRepository اضافه کنید.

```
using Entities.Models;
using System;
using System.Collections.Generic;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        IEnumerable<Employee> GetEmployees(Guid companyId, bool trackChanges);

        Employee GetEmployee(Guid companyId, Guid id, bool trackChanges);

        void CreateEmployeeForCompany(Guid companyId, Employee employee);

        void DeleteEmployee(Employee employee);
    }
}
```

- گام بعدی پیاده‌سازی این اینترفیس در کلاس EmployeeRepository است.

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>, IEmployeeRepository
    {
        public EmployeeRepository(CompanyEmployeeDbContext companyEmployeeDbContext) : base(companyEmployeeDbContext)
        {
        }
    }
}
```

```

    public IEnumerable<Employee> GetEmployees(Guid companyId, bool
trackChanges) => FindByCondition(e =>
e.CompanyId.Equals(companyId), trackChanges).OrderBy(e => e.Name);

    public Employee GetEmployee(Guid companyId, Guid id, bool
trackChanges) => FindByCondition(e =>
e.CompanyId.Equals(companyId) && e.Id.Equals(id),
trackChanges).SingleOrDefault();

    public void CreateEmployeeForCompany(Guid companyId, Employee
employee)
{
    employee.CompanyId = companyId;
    Create(employee);
}

    public void DeleteEmployee(Employee employee)
{
    Delete(employee);
}

}

```

- در مرحله پايانى باید اکشن متده Delete را به EmployeesController اضافه کنيد.

```

[HttpDelete("{id}")]
public IActionResult DeleteEmployeeForCompany(Guid companyId, Guid id)
{
    var company = _repository.Company.GetCompany(companyId,
trackChanges: false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't exist in the
database.");
    }

    return NotFound();
}

```

```

var employeeForCompany =
    _repository.Employee.GetEmployee(companyId, id, trackChanges:
        false);

if (employeeForCompany == null)
{
    _logger.LogError($"Employee with id: {id} doesn't exist in the
        database.");
    return NotFound();
}

_repository.Employee.DeleteEmployee(employeeForCompany);
_repository.Save();

return NoContent();
}

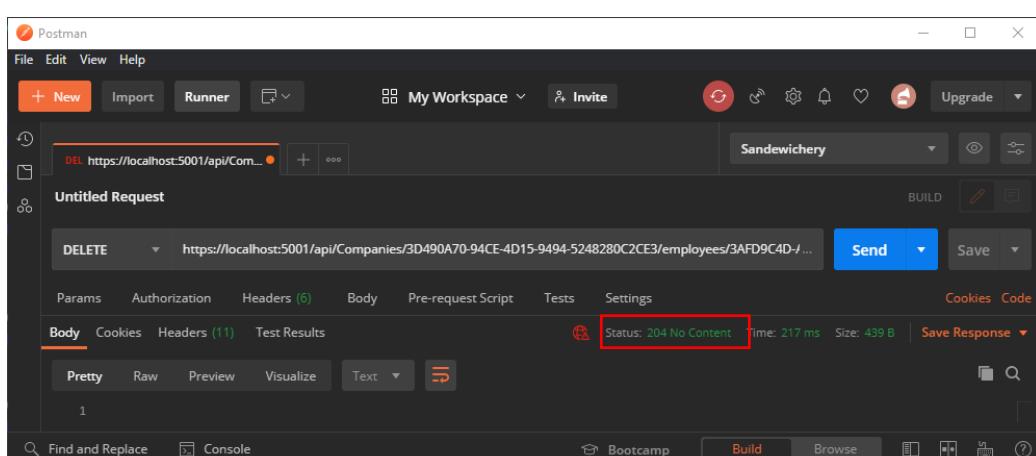
```

بررسی کد :

- چیز جدیدی در این اکشن متده وجود ندارد. ما companyId را از مسیر اصلی و id کارمند را از آرگومان که پاس داده شده می‌گیریم.
- در اینجا باید چک کنیم که شرکت و کارمند مورد نظر وجود داشته باشد.
- در پایان کارمند موردنظر را حذف و متده NoContent را که حاوی Status Code 204 است بر می‌گردانیم.

بیایید این اکشن متده را تست کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees/80abbca8-664d-4b20-b5de-024705497d4a>

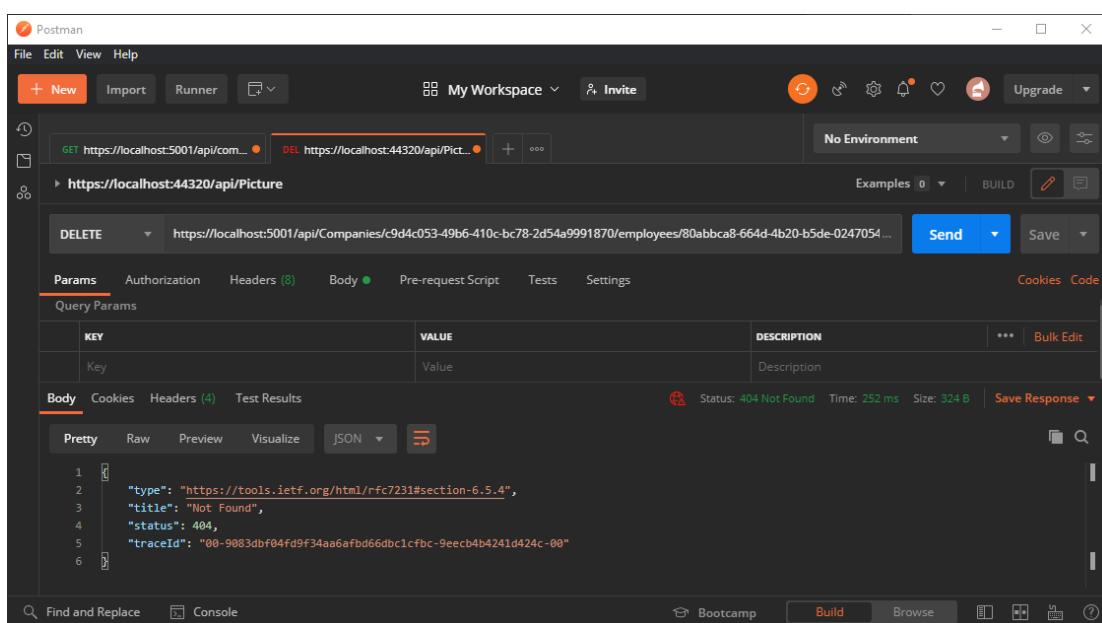


نکته!!

مقدار companyId و id که در این تست می‌بینید با مقادیری که در دیتابیس شما وجود دارد متفاوت است؛ چون این مقادیر GUID هستند و هر بار یک مقدار متفاوت دارند. پس برای تست این گونه مثال‌ها از مقادیر معتبر استفاده کنید.

حالا اگر بخواهید که این کارمند را از دیتابیس بگیرید مطمئناً 404 برگردانده خواهد شد.

<https://localhost:5001/api/companies/0AD5B971-FF51-414D-AF01-34187E407557/employees/DE662003-ACC3-4F9F-9D82-0A74F64594C1>



همانطور که می‌بینید این ریکوئست را هر چند بار دیگر هم ارسال کنید همان 404 را می‌گیرید. در نتیجه Idempotent همیشه DELETE Request است.

حذف یک Child همراه با Parent Resource

برای حذف یک Parent Resource همراه با Child هایش، می‌توانیم از Entity Framework Core کمک بگیریم. این ORM یک تنظیمات پایه به نام Cascade deleting دارد که با آن می‌توان مشخص کنیم که با حذف یک Parent، به صورت اتوماتیک تمام Child‌ها هم حذف شوند.

```
builder.HasOne("Entities.Models.Company", "Company")
    .WithMany("Employees")
    .HasForeignKey("CompanyId")
    .OnDelete(DeleteBehavior.Cascade)
    .IsRequired();
```

برای اعمال این قابلیت، کدهای کلاس EmployeeConfiguration را همانند پایین تغییر دهید.

```
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;

namespace Entities.Configuration
{
    public class EmployeeConfiguration :
        IEntityTypeConfiguration<Employee>
    {
        public void Configure(EntityTypeBuilder<Employee> builder)
        {
            builder.HasOne("Entities.Models.Company", "Company")
                .WithMany("Employees")
                .HasForeignKey("CompanyId")
                .OnDelete(DeleteBehavior.Cascade)
                .IsRequired();

            builder.HasData
            (
                new Employee
                {
                    Id = new Guid("80abbca8-664d-4b20-b5de-024705497d4a"),
                    Name = "Zahra Bayat",
                    Age = 26,
                    Position = "Backend developer",
                    CompanyId = new Guid("c9d4c053-49b6-410c-bc78-
                    2d54a9991870")
                },
                new Employee
                {
                    Id = new Guid("86dba8c0-d178-41e7-938c-ed49778fb52a"),
                    Name = "Ali Bayat",
                }
            );
        }
    }
}
```

```

        Age = 30,
        Position = "Backend developer",
        CompanyId = new Guid("c9d4c053-49b6-410c-bc78-
2d54a9991870")
    },
    new Employee
    {
        Id = new Guid("021ca3c1-0deb-4af8-ae94-2159a8479811"),
        Name = "Sara Bayat",
        Age = 35,
        Position = "Frontend developer",
        CompanyId = new Guid("3d490a70-94ce-4d15-9494-
5248280c2ce3")
    }
);
}

}

```

حالا تنها کاری که باید انجام دهیم ایجاد یک منطق برای حذف Parent Resource است. پس مانند مثال قبل، یک متدهای Delete به اینترفیس ICompanyRepository اضافه کنید.

```

using Entities.Models;
using System.Collections.Generic;
using System;

namespace Contracts.IServices
{
    public interface ICompanyRepository
    {
        IEnumerable<Company> GetAllCompanies(bool trackChanges);
        Company GetCompany(Guid companyId, bool trackChanges);
        void CreateCompany(Company company);
        IEnumerable<Company> GetByIds(IEnumerable<Guid> ids, bool
trackChanges);
        void DeleteCompany(Company company);
    }
}

```

سپس کلاس CompanyRepository را تغییر دهید.

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext)
            : base(companyEmployeeDbContext)
        { }

        public IEnumerable<Company> GetAllCompanies(bool trackChanges)
        =>
            FindAll(trackChanges)
                .OrderBy(c => c.Name)
                .ToList();

        public Company GetCompany(Guid companyId, bool trackChanges) =>
            FindByCondition(c => c.Id.Equals(companyId), trackChanges)
                .SingleOrDefault();

        public void CreateCompany(Company company) => Create(company);

        public IEnumerable<Company> GetByIds(IEnumerable<Guid> ids,
            bool trackChanges) =>
            FindByCondition(x => ids.Contains(x.Id), trackChanges)
                .ToList();

        public void DeleteCompany(Company company)
        {
            Delete(company);
        }
    }
}
```

```
}
```

در پایان اکشن متدهای DeleteCompany را به CompaniesController اضافه نمایید.

```
[HttpDelete("{id}")]
public IActionResult DeleteCompany(Guid id)
{
    var company = _repository.Company.GetCompany(id, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInformation($"Company with id: {id} doesn't exist in the
database.");

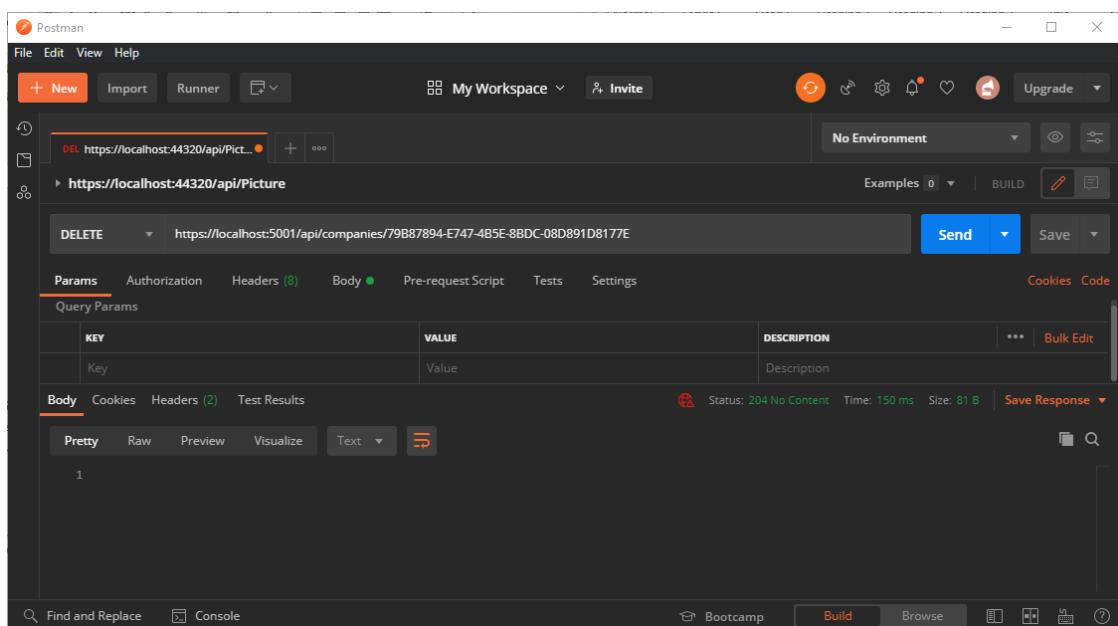
        return NotFound();
    }

    _repository.Company.DeleteCompany(company);
    _repository.Save();

    return NoContent();
}
```

خب حالا بیاید این اکشن متدهای را تست کنیم.

<https://localhost:5001/api/companies/79B87894-E747-4B5E-8BDC-08D891D8177E>



همانطور که می‌بینید این اکشن متدهم به خوبی کار می‌کند. حالا در دیتابیس بررسی کنید که این شرکت همراه با کارمندانش حذف شده است یا خیر؟

```
SQLQuery2.sql - (lo...yEmployee (sa (68))*)  X
*****
 Script for SelectTopNRows command from SSMS *****/
SELECT *
FROM [CompanyEmployee].[dbo].[Employees]
where CompanyId = '79B87894-E747-4B5E-8BDC-08D891D8177E'

100 %  <
Results  Messages
EmployeeId Name Age Position CompanyId
1 John Doe 30 Manager 79B87894-E747-4B5E-8BDC-08D891D8177E
```

خب تست DELETE Request تمام شد و باید PUT request را بررسی کنیم.

آپدیت Employee

در بخش قبل ابتدا اینترفیس و کلاس Repository را تغییر دادیم. سپس متدهای Delete را به کلاس کنترلر اضافه کردیم. اما برای آپدیت این مراحل کمی فرق دارد.

باید مرحله به مرحله این کار را انجام دهیم.

اولین کاری که باید انجام دهیم این است که در فolder DataTransferObjects یک کلاس با نام EmployeeForUpdateDto ایجاد کنیم.

```
namespace Entities.DataTransferObjects
{
    public class EmployeeForUpdateDto
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public string Position { get; set; }
    }
}
```

به پر اپرتی Id نیاز نداریم چون در Update هم مانند DELETE این پر اپرتی باید از طریق URI گرفته شود. همچنین این DTO مشابه EmployeeForCreationDto است. تنها تفاوت این دو کلاس در مفهوم آن هاست یکی از کلاس ها برای ایجاد است و دیگری برای آپدیت کاربرد دارد.

چون ما یک کلاس DTO اضافه کردیم پس باید عمل مپ کردن را هم انجام دهیم بنابراین کد پایین را به کلاس MappingProfile اضافه کنید.

```
CreateMap<EmployeeForUpdateDto, Employee>();
```

خب حالا باید اکشن متدهای UpdateEmployeeForCompany را به EmployeesController اضافه کنید.

```
[HttpPost("{id}")]
public IActionResult UpdateEmployeeForCompany(Guid companyId, Guid id,
[FromBody] EmployeeForUpdateDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForUpdateDto object sent from client is null.");
        return BadRequest("EmployeeForUpdateDto object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges: false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the database.");
        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id, trackChanges: true);

    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(employee, employeeEntity);
```

```

    _repository.Save();

    return NoContent();
}

```

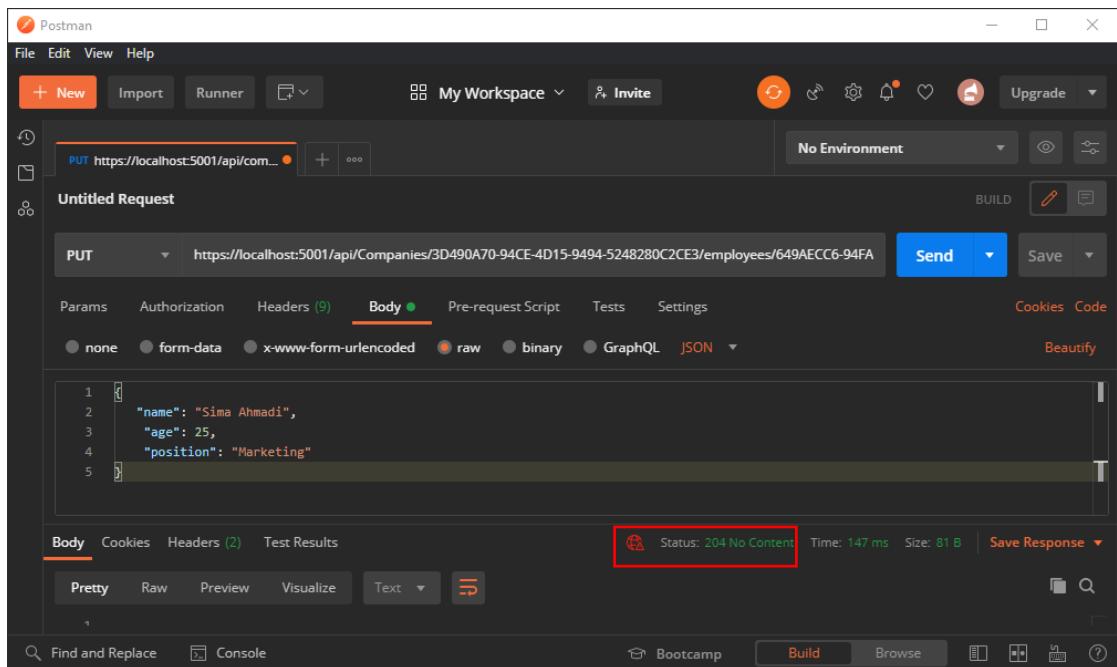
بررسی کد :

- در این اکشن متده از اتریبوت PUT با پارامتر Id استفاده کردیم. این یعنی : مسیر ما برای این اکشن متده برابر {companyId}/employees/{id} است.
 - در کد بالا، سه جا عمل چک کردن داریم که مطمئنا برای همه شما آشنا است.
 - تنها کدی که کمی متفاوت است نحوه واکشی employeeEntity و company میباشد.
 - پارامتر trackChanges برای employeeEntity بر روی true تنظیم شده تا هر پراپرتی این Entity تغییر کرد، وضعیت آن را به Modified تغییر دهد.
 - در پایان این کد میبینید، آبجکت employee را به employeeEntity مپ کردیم بنابراین state آبجکت employeeEntity به Modified تغییر میابد.
 - حالا با صدا زدن Save، Entity در دیتابیس آپدیت میشود.
 - در نهایت NoContent برگردانده میشود.
- حالا میتوانیم این اکشن را تست بزنیم.

<https://localhost:5001/api/Companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
{
  "name": "Sima Ahmadi",
  "age": 25,
  "position": "Marketing"
}
```



در ریکوئست بالا پر اپرتی Age از ۲۸ به ۲۵ تغییر کرد.

همانطور که می بینید این اکشن متدهای کار می کند و ما No Content می گیریم.

اگر همین ریکوئست را با CompanyId یا EmployeeId نامعتبر ارسال کنید باید ریسپانس 404 دریافت کنید.

نکته:

همانطور که دیدید ما می خواستیم فقط پر اپرتی Age را تغییر دهیم اما تمام پر اپرتی ها با مقادیری که در دیتابیس داشتند را، هم ارسال کردیم.

در اینجا نکته ای هست که باید بدانید.

PUT یک ریکوئست برای آپدیت کامل است بنابراین اگر ریکوئست بالا را تنها با پر اپرتی Age (بدون باقی پر اپرتی ها) هم می فرستادیم، باز تمام پر اپرتی ها روی مقادیر پیش فرض شان تنظیم می شدند. بنابراین در ریکوئست PUT نیاز به ارسال پارامترهای اضافه نیست.

انواع روش های Update

احتمالاً این سوال در ذهن شما هم هست که چرا از متدهای Update کلاس RepositoryBase استفاده نمی کنیم؟

قبل از پاسخ به این سوال، بگذارید به دو روش Update اطلاعات اشاره‌ای داشته باشیم :

object: یعنی برای واکشی و آپدیت Entity، از یک **Connected Update** • مشترک استفاده شود.

object: یعنی برای واکشی و آپدیت Entity، از **Disconnected Update** • های مختلف استفاده نمود.

چون شرایط اکشن متدهای UpdateEmployeeForCompany طوری بود که باید به صورت Connected آن را پیاده می‌کردیم، بنابراین از متدهای RepositoryBase کلاس استفاده نکردیم. البته گاهی نیاز است که از Disconnected Update استفاده شود. به طور مثال :

یک آبجکت همراه با Id، از کلاینت دریافت کردیم و نیازی به واکشی اطلاعات از دیتابیس نداریم. در این شرایط، تنها کاری که باید انجام دهیم این است که به EF Core اطلاع دهیم تا تغییرات موجود در Entity را Track و آن را Modified کند. بنابراین می‌توانیم هر دو عمل را با استفاده از متدهای RepositoryBase کلاس Update انجام دهیم.

ایجاد Resource در هنگام آپدیت یک Resource

ما می‌توانیم بدون نوشتگی هیچ گونه کد اضافه‌ای، Child Resource را در هنگام آپدیت ایجاد کنیم. EF Core در این پروسه به ما کمک بزرگی می‌کند.

بیایید ببینیم این کار چطور انجام می‌شود.

اولین کاری که باید انجام دهیم، ایجاد یک کلاس DTO برای آپدیت است. بنابراین در فolder DataTransferObjects یک کلاس با نام CompanyForUpdateDto ایجاد نمایید.

```
using System.Collections.Generic;

namespace Entities.DataTransferObjects
{
    public class CompanyForUpdateDto
    {
        public string Name { get; set; }
        public string Address { get; set; }
        public string Country { get; set; }
        public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
```

```
}
```

```
}
```

بعد از این مرحله یک MappingProfile جدید در کلاس MappingProfile اضافه کنید.

```
CreateMap<CompanyForUpdateDto, Company>();
```

خب حالا باید در CompaniesController اکشن متده UpdateCompany را ایجاد کنید.

```
[HttpPut("{id}")]
public IActionResult UpdateCompany(Guid id, [FromBody] CompanyForUpdateDto company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForUpdateDto object sent from client is null.");
        return BadRequest("CompanyForUpdateDto object is null");
    }

    var companyEntity = _repository.Company.GetCompany(id, trackChanges: true);

    if (companyEntity == null)
    {
        _logger.LogInformation($"Company with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    _repository.Save();

    return NoContent();
}
```

همانطور که می‌بینید این اکشن متده، مشابه UpdateEmployeeForCompany است. پس باید این مورد را هم تست کنیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3>

body:

```
{
```

```

"name": "IDP_Ltd",
"address": "Tehran, Vanak",
"country": "Iran",
"employees": [
    {
        "name": "Sara Mahdavi",
        "age": 25,
        "position": "IT"
    }
]

```

The screenshot shows the Postman interface with a PUT request to the specified URL. The request body is a JSON object containing company information and an employee array. The 'employees' array is highlighted with a red box.

در ریکوئست بالا، نام شرکت را تغییر و یک کارمند را نیز Attach کردیم. در نتیجه همانطور که می‌بینید 204 برگردانده می‌شود؛ یعنی اینکه Entity ما آپدیت شده است.

آیا با ریکوئست بالا کارمند جدید اضافه شد؟ بیایید کوئری خود را بررسی کنیم.

```

SQLQuery1.sql - (Io...yEmployee (sa (52))*)  X
===== Script for SelectTopNRows command from SSMS =====
SELECT *
FROM [CompanyEmployee].[dbo].[Employees]
where
CompanyId='3D490A70-94CE-4D15-9494-5248280C2CE3' and Name='Sara Mahdavi'

```

	EmployeeId	Name	Age	Position	CompanyId
1	189BC18E-0F1C-4183-EF60-08D8969DEF13	Sara Mahdavi	25	IT	3D490A70-94CE-4D15-9494-5248280C2CE3

همانطور که می‌بینید این کارمند در دیتابیس ایجاد شده است چون ما Company Entity را Track کردیم. بنابراین EF Core به محض اینکه Mapping انجام شود، Company → state روی Employees تنظیم و برای State هم Added می‌گذارد. حالا بعد از اینکه متدهای Save صدازده شود، علاوه بر آپدیت شرکت، کارمند جدید هم در دیتابیس اضافه می‌شود.

کار ما با PATCH Request ها تمام شد. در قسمت بعدی می‌خواهیم PUT Request را بررسی کنیم.

فصل هفتم : Patch Request و اعتبارسنجی

آنچه خواهید آموخت:

- کار کردن با **Patch Request**
- اعتبارسنجی در زمان ایجاد و حذف **Resource**

کار کردن با Patch Request

در فصل قبل با PUT Request کار کردیم. PUT Request برای آپدیت کامل استفاده می‌شد اما اگر بخواهیم بخشی از Resource را آپدیت کنیم باید از PATCH Request استفاده کنیم.

تفاوت بین PUT و PATCH :

- در PUT همیشه Resource به صورت کامل آپدیت می‌شود در حالیکه در PATCH Request قسمتی از Resource آپدیت خواهد شد.
- PATCH Request Body نیز در هر کدام متفاوت است. برای مثال :
[FromBody]JsonPatchDocument<Company> Company
برای نوشته می‌شود در حالیکه برای PUT باید از [FromBody]Company استفاده کنیم.
- PATCH برای Media Type application/json باید باشد اما در Request باشد از application/json-patch+json استفاده کرد.

نکته!!

در ASP.NET Core برای PATCH Request application/json هم پذیرفته می‌شود اما استانداردهای REST توصیه می‌کنند از application/json-patch+json استفاده کنید.

بیایید ببینیم بدن PATCH Request چه شکلی است.

```
[  
{  
  "op": "replace",  
  "path": "/name",  
  "value": "new name"  
},  
{  
  "op": "remove",  
  "path": "/name"  
}  

```

• علامت برآکت، مجموعه‌ای از عملیات را نشان می‌دهد. هر عمل باید بین آکولاد قرار گیرد. بنابراین در این مثال دو عمل Remove و Replace که با پراپرتی op مشخص شده‌اند.

• path نمایانگر پراپرتی است که می‌خواهیم آن را تغییر دهیم.

در این مثال برای عمل Replace ما مقدار پراپرتی name را با یک مقدار جدید جایگزین می‌کنیم و در دومی ما پراپرتی name را حذف و مقدار آن را به صورت پیش فرض تنظیم خواهیم کرد.

شش ریکوئست مختلف برای PATCH وجود دارد :

OPERATION	REQUEST BODY	توضیحات
Add	{ "op": "add", "path": "/name", "value": "new value" }	به پراپرتی مقدار جدیدی داده می‌شود.
Remove	{ "op": "remove", "path": "/name" }	برای پراپرتی مقدار پیش‌فرض تنظیم می‌شود.
Replace	{ "op": "replace", "path": "/name", "value": "new value" }	مقدار یک پراپرتی را با یک مقدار جدید جایگزین می‌کند.
Copy	{ "op": "copy", "from": "/name", "path": "/title" }	مقدار را از یک پراپرتی به یک پراپرتی دیگر کپی می‌کند.
Move	{ "op": "move", "from": "/name", "path": "/title" }	مقدار را از یک پراپرتی به یک پراپرتی دیگر جابجا می‌کند.
Test	{ "op": "test", "path": "/name", "value": "new value" }	تست می‌کند آیا پراپرتی مقدار مشخص شده را دارد؟

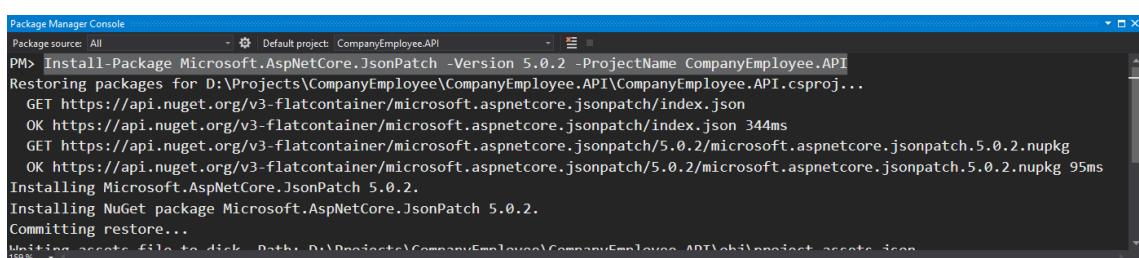
خب حالا بباید وارد کدنویسی شویم.

اضافه کردن Employee Entity به Patch

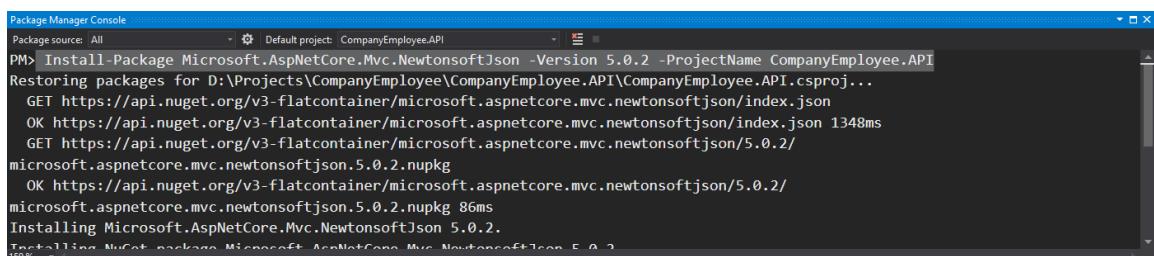
قبل از اینکه کنترلر را تغییر دهیم باید دو پکیج نصب کنیم.

- پکیج JsonPatchDocument برای پشتیبانی از Microsoft.AspNetCore.JsonPatch استفاده می‌شود.
- Request Body Microsoft.AspNetCore.Mvc.NewtonsoftJson و پکیج PatchDocument (پس از ارسال ریکوئست) است.

```
Install-Package Microsoft.AspNetCore.JsonPatch -Version 5.0.2 -  
ProjectName CompanyEmployee.API
```



```
Install-Package Microsoft.AspNetCore.Mvc.NewtonsoftJson -Version 5.0.2 -  
ProjectName CompanyEmployee.API
```



پس از اتمام نصب، باید AddControllers را به NewtonsoftJsonConfiguration اضافه کنیم.

```
services.AddControllers(config => {  
    config.RespectBrowserAcceptHeader = true;  
    config.ReturnHttpNotAcceptable = true;  
}).AddNewtonsoftJson()  
.AddXmlDataContractSerializerFormatters()  
.AddCustomCSVFormatter();
```

توجه داشته باشید که متدهای AddNewtonsoftJson و AddCustomCSVFormatter را قبل از AddNewtonsoftJson اضافه نمایید.

خب حالا نیاز است تا یک EmployeeForUpdateDto به Employee از نوع Mapping داشته باشیم. اگر نگاهی به کلاس MappingProfile بیندازید، می‌بینید که یک Mapping از Rule وجود دارد؛ پس دیگر نیازی به اضافه کردن EmployeeForUpdateDto جدید نداریم، فقط کافیست که از متده استفاده کنیم.

```
CreateMap<EmployeeForUpdateDto, Employee>().ReverseMap();
```

متده استفاده کردن ReverseMap برای معکوس کردن Mapping می‌شود.
حالا می‌توانیم متده PartiallyUpdateEmployeeForCompany را به EmployeesController اضافه کنیم.

ابتدا این EmployeesController را در Namespace اضافه کنید.

```
using Microsoft.AspNetCore.JsonPatch;
```

و سپس متده پایین را بنویسید.

```
[HttpPatch("{id}")]
public IActionResult PartiallyUpdateEmployeeForCompany(Guid companyId,
Guid id, [FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is null.");
        return BadRequest("patchDoc object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInformation($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }
```

```

var employeeEntity = _repository.Employee.GetEmployee(companyId, id,
trackChanges: true);

if (employeeEntity == null)
{
    _logger.LogInfo($"Employee with id: {id} doesn't exist in the
database.");
}

return NotFound();
}

var employeeToPatch =
_mapper.Map<EmployeeForUpdateDto>(employeeEntity);
patchDoc.ApplyTo(employeeToPatch);
_mapper.Map(employeeToPatch, employeeEntity);
_repository.Save();

return NoContent();
}

```

: بررسی کد

- همانطور که می‌بینید Signature این اکشن متدهای PUT و PATCH متفاوت است. ما Request Body را از JsonPatchDocument می‌گیریم.
 - پس از آن بررسی می‌کنیم که اگر patchDoc برابر null است یک BadRequest ایجاد شود.
 - در خطهای بعدی باید مطمئن شویم که employee و company در دیتابیس وجود دارد یا خیر.
 - چون متغیر patchDoc فقط می‌تواند به نوع EmployeeForUpdateDto اعمال شود پس نوع EmployeeForUpdateDto را به Employee مپ می‌کنیم.
 - در پایان دوباره employeeEntity را به employeeToPatch مپ می‌کنیم تا بتوانیم تغییرات را در دیتابیس ذخیره کنیم.
- حالا برای تست، دو ریکوئست پایین را بفرستید.
- ابتدا عملیات Replace را ارسال کنید.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[  
  {  
    "op": "replace",  
    "path": "/age",  
    "value": "28"  
  }  
]
```

The screenshot shows the Postman interface with a PATCH request sent to the specified URL. The request body is a JSON array containing one object with 'op': 'replace', 'path': '/age', and 'value': '28'. The response status is 204 No Content, indicating the operation was successful but no new content was returned.

همانطور که می‌بینید ما پیام 204 No Content را گرفتیم. حالا باید Employee تغییر کرده را چک کنیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

The screenshot shows the Postman interface with a successful GET request. The response body is highlighted with a red box, showing a JSON object with fields id, name, age, and position.

```

1 [
2   {
3     "id": "649aecc6-94fa-4c67-828a-08d8936f5863",
4     "name": "Sima Ahmadi",
5     "age": 28,
6     "position": "Marketing"
7   ]

```

همانطور که می‌بینید پرایپری Age تغییر کرده است. بیایید یک عمل حذف را در یک ریکوئست بفرستیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[
  {
    "op": "remove",
    "path": "/age"
  }
]
```

The screenshot shows the Postman interface with a successful PATCH request. The request body is highlighted with a red box, containing the same JSON object as the previous screenshot.

```

1 [
2   {
3     "op": "remove",
4     "path": "/age"
5   }
6 ]

```

این عمل هم به درستی کار کرد. حال اگر کارمند خود را Get کنیم، پر اپرتی Age در عدد صفر تنظیم می شود (مقدار پیش فرض برای نوع int):

The screenshot shows the Postman interface with a GET request to the specified URL. The response body is displayed in a JSONpretty format, showing an array with one element. The element contains an employee object with properties: id, name, age (which is highlighted with a red box), and position.

```

[{"id": "649aecc6-94fa-4c67-828a-08d8936f5863", "name": "Sara Sadeghi", "age": 0, "position": "..."}]
  
```

در پایان بباید برای پر اپرتی Age مقدار ۲۸ را برگردانیم.

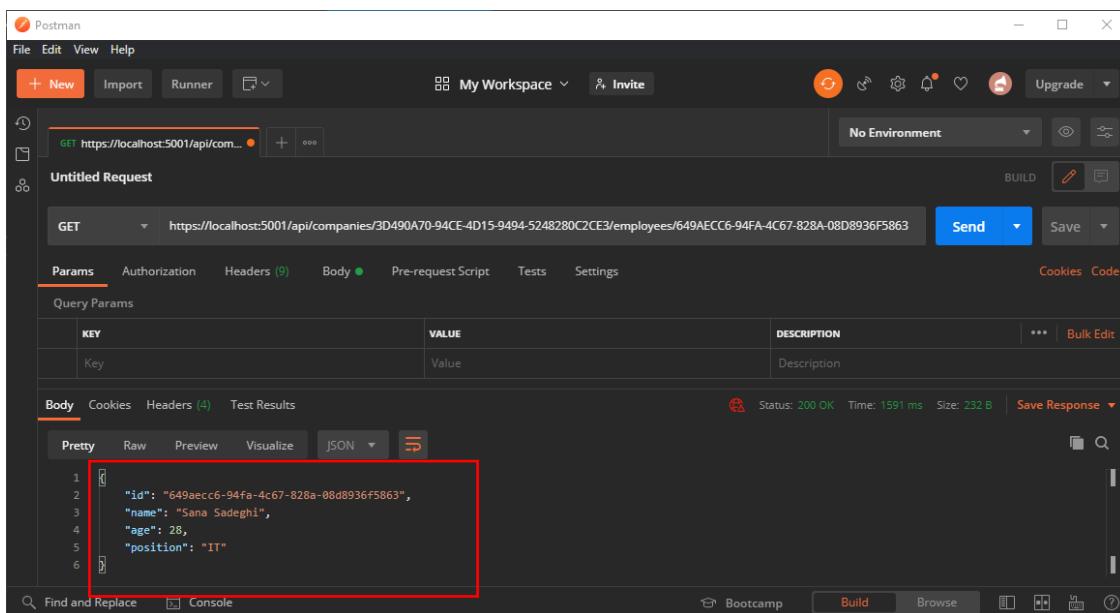
<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[{"op": "add", "path": "/age", "value": "28"}]
```

The screenshot shows the Postman interface with a PATCH request to the same URL. The request body is a JSON array with one element, identical to the one shown in the previous step. The response status is 204 No Content.

باید با هم Employee را چک کنیم.



The screenshot shows the Postman interface with an 'Untitled Request' for a GET request to the specified URL. The response status is 200 OK, and the response body is a JSON object:

```
1 "id": "649aecc6-94fa-4c67-828a-08d8936f5863",
2 "name": "Sana Sadeghi",
3 "age": 28,
4 "position": "IT"
```

عالی شد همه چیز خیلی خوب کار کرد.

اعتبارسنجی چیست؟

داده‌ها قبل از ارسال باید اعتبارسنجی شوند و این موضوع خیلی مهمی در ذخیره‌سازی اطلاعات است. برای حل این مسئله، استفاده از DataAnnotation‌ها مطرح شده است. DataAnnotation‌ها به شما این امکان را می‌دهند تا، قوانینی مشخص کنید و پرپرتبه‌ها در مطابق با این قوانین عمل نمایند. Model

شیوه‌ی کار بدین صورت است که، DataAnnotation‌های برای توصیف داده آماده می‌کنند و داده‌ها باید از قوانین این DataAnnotation‌ها پیروی نمایند.

به عنوان مثال:

به جای اینکه null بودن داده‌ها به صورت دستی چک شود، اtribut [Required] را بالای پرپرتبه‌های خود بگذارید و دیگر خیالتان راحت باشد که نمی‌توان داده‌ی Null وارد نمود.

: DataAnnotation‌ها برخی

• : برای اعتبارسنجی فرمت ایمیل.

- [MinLength(min)] : بررسی حداقل کاراکتر وارد شده.
- [Phone] : برای اعتبارسنجی فرمت تلفن.
- [Required] : ورود مقدار برای پر اپرتی اجباریست.
- [Range(min, max)] : بررسی مقدار بین حداقل و حدکثر.
- [Display(Name = "")] : نام سفارشی پر اپرتی برای نمایش در View.

```

20 references
public class Company
{
    [Column("CompanyId")]
    4 references
    public Guid Id { get; set; }
    [Required(ErrorMessage = "Company name is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Name is 60 characters.")]
    3 references
    public string Name { get; set; }
    [Required(ErrorMessage = "Company address is a required field.")]
    [MaxLength(60, ErrorMessage = "Maximum length for the Address is 60 characters.")]
    3 references
    public string Address { get; set; }
    3 references
    public string Country { get; set; }
    0 references
    public ICollection<Employee> Employees { get; set; }
}

```

برای اعتبارسنجی در زمان ایجاد یا آپدیت Resource از اtribووت‌ها استفاده می‌شود؛ یعنی این اعتبارسنجی‌ها در ریکوئست‌های POST، PUT و PATCH کاربرد دارند.

اعتبارسنجی قبل از اجرای اکشن‌متد، رخ می‌دهد اما توجه داشته باشید که اکشن‌متد‌ها در هر صورت (چه اعتبارسنجی موفق باشد چه نباشد) اجرا خواهد شد بنابراین مدیریت داده‌های معتبر بر عهده‌ی اکشن‌متد است.

برای بررسی اعتبارسنجی، می‌توان از پر اپرتی ModelState موجود در کلاس ControllerBase استفاده کنیم. ModelState یک دیکشنری است که حاوی لیستی از تمام خطاهای اعتبارسنجی می‌باشد.

هنگامی که ریکوئست خود را ارسال می‌کنید، قوانین تعریف شده با Data Annotation بررسی می‌شوند. اگر یکی از قوانین اعمال نشده باشد، پیام خطای مناسب بازگشت داده خواهد شد.

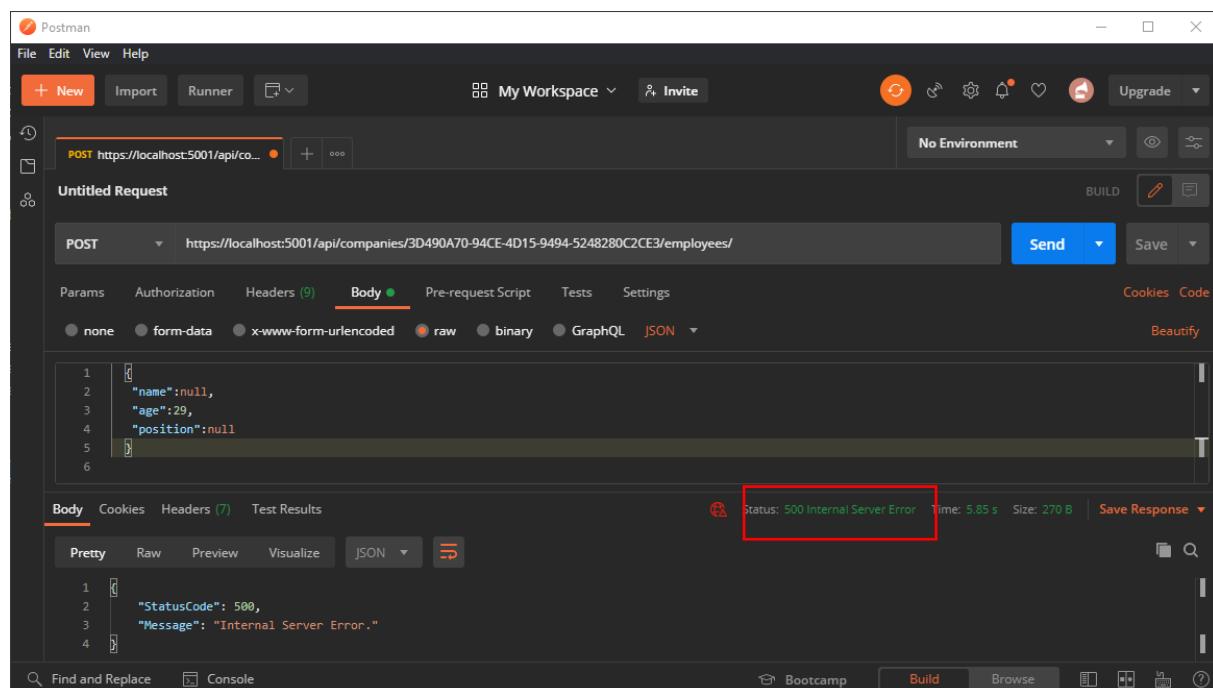
ولیدیشن در زمان ایجاد Resource

قبل از بررسی ولیدیشن، باید ریکوئستی با Body نامعتبر، به اکشن متده CreateEmployee ارسال کنیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{  
    "name":null,  
    "age":29,  
    "position":null  
}
```



همانطور که می‌بینید Response این ریکوئست، خطای Internal Server 500 است. این خطا زمانی داده می‌شود که چیزی غیرقابل کنترل، در کد اتفاق افتاده و سرور خطایی ایجاد کرده باشد.

همه شما می‌دانید که تنها دلیل این خطا، ارسال یک مدل اشتباه به API است بنابراین پیام خطا باید متفاوت باشد.

برای رفع این مشکل، باید کلاس EmployeeForCreationDto را تغییر دهیم؛ زیرا این همان آبجکتی است که ما از بدنه ریکوئست Deserialize می‌کنیم.

```

using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public class EmployeeForCreationDto
    {
        [Required(ErrorMessage = "Employee name is a required field.")]
        [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30
characters.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Age is a required field.")]
        public int Age { get; set; }

        [Required(ErrorMessage = "Position is a required field.")]
        [MaxLength(20, ErrorMessage = "Maximum length for the Position is
20 characters.")]
        public string Position { get; set; }
    }
}

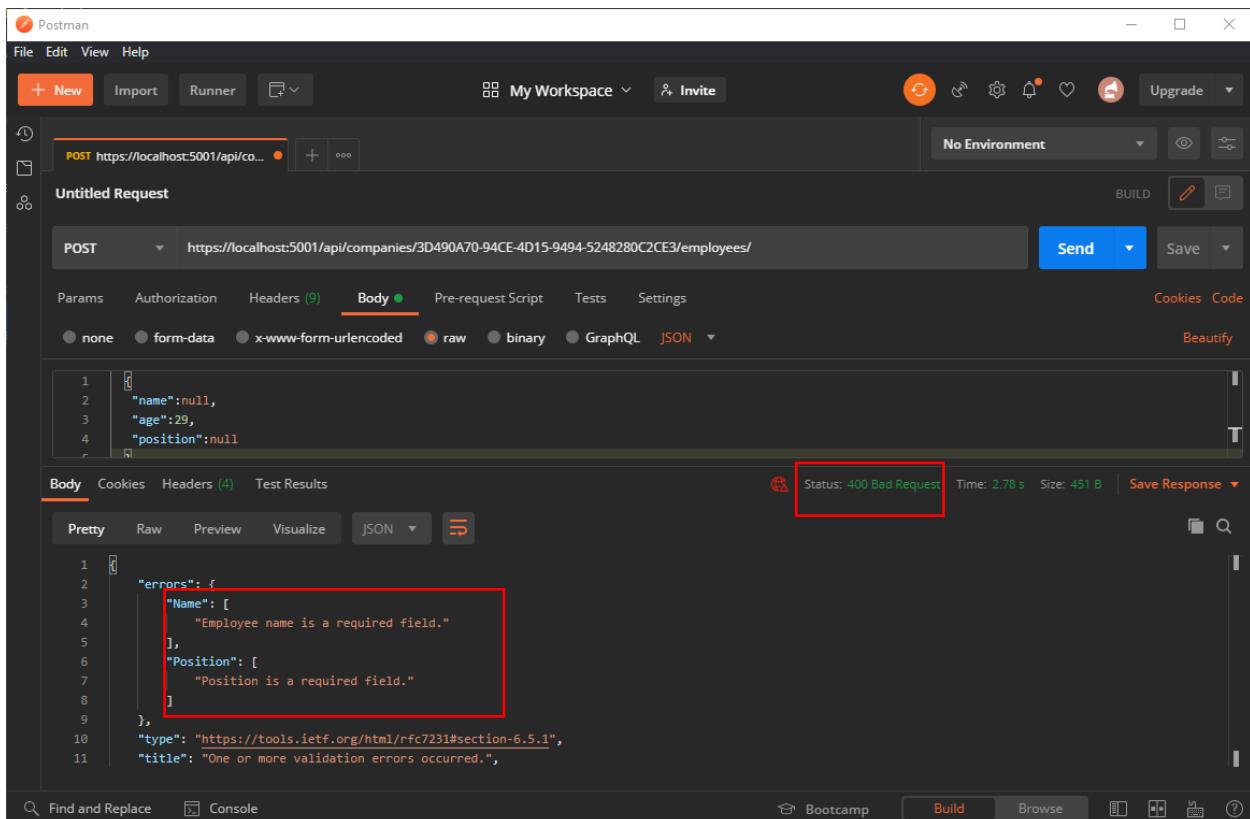
```

پس از اعمال قوانین بالا، می‌توانیم دوباره ریکوئست پایین را ارسال کنیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{
    "name":null,
    "age":29,
    "position":null
}
```



Bad – Status Code به مضمون گرفتن ریکوئست، مدل را اعتبارسنجی و 400 Request را فرستد.

این نتیجه قابل قبول است، اما همانطور که گفتیم بهتر است Status Code هم، با این شرایط متناسب باشد. بهترین Status Code برای این شرایط، 422 Unprocessable Entity است. بنابراین اولین کاری که باید انجام دهیم، این است که در صورت نا معتبر بودن ModelState مانع خطای BadRequest شویم.

گام اول اضافه کردن کد پایین به متدهای ConfigureServices است.

```

services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressModelStateInvalidFilter = true;
});

```

Startup پایین است، پس این را هم در کلاس Namespace ApiBehaviorOptions کلاس اضافه کنید.

```
using Microsoft.AspNetCore.Mvc;
```

حالا باید اکشن متدهیم را تغییر دهیم.

```
[HttpPost]
public IActionResult CreateEmployeeForCompany(Guid companyId, [FromBody]
EmployeeForCreationDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForCreationDto object sent from client is
null.");
        return BadRequest("EmployeeForCreationDto object is null");
    }
    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the
EmployeeForCreationDto object");
        return UnprocessableEntity(ModelState);
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");
        return NotFound();
    }

    var employeeEntity = _mapper.Map<Employee>(employee);
    _repository.Employee.CreateEmployeeForCompany(companyId,
employeeEntity);
    _repository.Save();

    var employeeToReturn = _mapper.Map<EmployeeDto>(employeeEntity);

    return
CreatedAtRoute("GetEmployeeForCompany",
new
```

```
{
    companyId,
    id = employeeToReturn.Id
},
employeeToReturn);
}
```

باید یک بار دیگر ریکوئست را ارسال کنیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{
    "name":null,
    "age":29,
    "position":null
}
```

The screenshot shows the Postman interface with a failed API call. The request URL is <https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>. The request method is POST. The 'Body' tab is selected, showing an empty JSON object. The response status is 422 Unprocessable Entity. The error message in the response body is: "Name": ["Employee name is a required field."], "Position": ["Position is a required field."]".

برای تست Max Length هم یک ریکوئست نیاز داریم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{
    "name": "Saman",
```

```

    "age":29,
    "position":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}

```

The screenshot shows the Postman interface with a request to `https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/`. The request body is set to `JSON` and contains:

```

1 {
2   "name": "Saman",
3   "age": 29,
4   "position": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
5

```

The response status is `422 Unprocessable Entity`, and the error message is:

```

1 [
2   "Position": [
3     "Maximum length for the Position is 20 characters."
4   ]
5

```

عالی شد همانطور که انتظار داشتیم این مورد هم کار می‌کند.

حالا برای اکشن متدهای `CreateCompany` هم باید همین مراحل را انجام دهیم.

• اضافه کنید `CompanyForCreationDto` را به کلاس `Data Annotation`

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public class CompanyForCreationDto
    {
        [Required(ErrorMessage = "Company name is a required field.")]
        [MaxLength(30, ErrorMessage = "Maximum length for the name is 30
        characters.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Company address is a required field.")]
        [MaxLength(100, ErrorMessage = "Maximum length for the address is
        100 characters.")]
        public string Address { get; set; }
    }
}

```

```

        [Required(ErrorMessage = "Company country is a required field.")]
        [MaxLength(50, ErrorMessage = "Maximum length for the country is
50 characters.")]
    public string Country { get; set; }

    public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
}

}

```

اکشن متدهید CreateCompany را تغییر دهد.

```

public IActionResult CreateCompany([FromBody] CompanyForCreationDto
    company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from client is
null.");

        return BadRequest("CompanyForCreationDto object is null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the
EmployeeForCreationDto object");

        return UnprocessableEntity(ModelState);
    }

    var companyEntity = _mapper.Map<Company>(company);
    _repository.Company.CreateCompany(companyEntity);
    _repository.Save();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyId", new { id = companyToReturn.Id },
companyToReturn);
}

```

اعتبارسنجی نوع int

یک Request Body که پر اپرتی Age ندارد را ارسال کنید.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{  
  "name": "Saman",  
  "position": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"  
}
```

The screenshot shows the Postman interface with a POST request to the specified URL. The request body is set to 'JSON' and contains the following JSON payload:

```
POST https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/  
{"name": "Saman", "position": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"}
```

The response status is 422 Unprocessable Entity, with a message: "Maximum length for the Position is 20 characters."

همانطور که می بینید، پر اپرتی Age ارسال نشده اما در Response Body هم هیچ پیام خطای نیست.

این مشکل به این دلیل است که سن از نوع int است و اگر آن را ارسال نکنیم، مقدار پیش فرض صفر برای آن تنظیم و اعتبارسنجی برای پر اپرتی Age نادیده گرفته می شود.

برای رفع این مشکل باید Data Annotation پر اپرتی Age را اصلاح کنیم.

```
using System.ComponentModel.DataAnnotations;  
  
namespace Entities.DataTransferObjects  
{  
    public class EmployeeForCreationDto  
    {  
        [Required(ErrorMessage = "Employee name is a required field.")]
```

```

[MaxLength(30, ErrorMessage = "Maximum length for the Name is 30
characters.")]

public string Name { get; set; }

[Required(ErrorMessage = "Age is a required field.")]
[Range(18, int.MaxValue, ErrorMessage = "Age is required and it
can't be lower than 18")]
public int Age { get; set; }

[Required(ErrorMessage = "Position is a required field.")]
[MaxLength(20, ErrorMessage = "Maximum length for the Position is
20 characters.")]
public string Position { get; set; }

}

```

اکنون بیایید یک بار دیگر ریکوئست پایین را بفرستیم.

<https://localhost:5001/api/companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/>

body:

```
{
  "name": "Saman",
  "position": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}
```

The screenshot shows the Postman interface with a POST request to the specified URL. The request body is a JSON object with 'name' and 'position' fields. The response status is 422 Unprocessable Entity, with an error message indicating validation errors for both the 'Age' and 'Position' fields.

حالا در Response پیام خطای Age را داریم.

اعتبارسنجی برای PUT Request

اعتبارسنجی POST Request هم شبیه PUT Request است باید مرحله به مرحله پیش برویم.

- ابتدا EmployeeForUpdateDto را به کلاس Data Annotation اضافه کنید.

```
using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public class EmployeeForUpdateDto
    {
        [Required(ErrorMessage = "Employee name is a required field.")]
        [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30
        characters.")]
        public string Name { get; set; }

        [Range(18, int.MaxValue, ErrorMessage = "Age is required and it
        can't be lower than 18")]
        public int Age { get; set; }

        [Required(ErrorMessage = "Position is a required field.")]
        [MaxLength(20, ErrorMessage = "Maximum length for the Position is
        20 characters.")]
        public string Position { get; set; }
    }
}
```

خب اگر این کلاس را با کلاس EmployeeForCreationDto مقایسه کنیم می‌بینیم که کد هر دو یکسان است و اینجاست که ما تکرار کد داریم.

پس باید بعضی تغییرات را اضافه کنیم.

گام اول: در فolder DataTransferObjects یک کلاس جدید با نام EmployeeForManipulationDto اضافه کنید.

```

using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public abstract class EmployeeForManipulationDto
    {
        [Required(ErrorMessage = "Employee name is a required field.")]
        [MaxLength(30, ErrorMessage = "Maximum length for the Name is 30
characters.")]
        public string Name { get; set; }

        [Range(18, int.MaxValue, ErrorMessage = "Age is required and it
can't be lower than 18")]
        public int Age { get; set; }

        [Required(ErrorMessage = "Position is a required field.")]
        [MaxLength(20, ErrorMessage = "Maximum length for the Position is
20 characters.")]
        public string Position { get; set; }
    }
}

```

ما این کلاس را به عنوان یک کلاس Abstract ایجاد می‌کنیم تا کلاس EmployeeForCreationDto و EmployeeForUpdateDto از آن ارث بری کند.

```

namespace Entities.DataTransferObjects
{
    public class EmployeeForUpdateDto: EmployeeForManipulationDto
    {
    }

    public class EmployeeForCreationDto: EmployeeForManipulationDto
    {
    }
}

```

اکنون می‌توانیم اکشن متدهای UpdateEmployeeForCompany را تغییر دهیم.

```

[HttpPut("{id}")]
public IActionResult UpdateEmployeeForCompany(Guid companyId, Guid id,
[FromBody] EmployeeForUpdateDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForUpdateDto object sent from client is null.");
        return BadRequest("EmployeeForUpdateDto object is null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the EmployeeForUpdateDto object");
        return UnprocessableEntity(ModelState);
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges: false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the database.");
        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id, trackChanges: true);

    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the database.");
        return NotFound();
    }

    _mapper.Map(employee, employeeEntity);
    _repository.Save();
}

```

```
        return NoContent();
    }
```

حالا نوبت تست این اکشن متده است.

<https://localhost:5001/api/Companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
{  
    "name":null,  
  
    "age":29,  
    "position":null  
}
```

Postman

File Edit View Help

PUT https://localhost:5001/api/com... Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {  
2     "name":null,  
3     "age":29,  
4     "position":null  
5 }  
6
```

Status: 422 Unprocessable Entity Time: 2.44 s Size: 250 B Save Response

Pretty Raw Preview Visualize JSON

Find and Replace Console Bootcamp Build Browse

اعتبارسنجی برای PATCH Request

اعتبارسنجی PATCH Request با ریکوئستهای قبلی کمی فرق دارد. در این اعتبارسنجی باید پر اپرتی ModelState را در متده ApplyTo قرار دهیم.

```
patchDoc.ApplyTo(employeeToPatch, ModelState);
```

خب حالا مانند کد زیر میتوانیم منطق اعتبارسنجی خود را به متده PartiallyUpdateEmployeeForCompany اضافه کنیم.

```
[HttpPatch("{id}")]
```

```

public IActionResult PartiallyUpdateEmployeeForCompany(Guid companyId,
Guid id, [FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is null.");

        return BadRequest("patchDoc object is null");
    }

    var company = _repository.Company.GetCompany(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist in the
database.");

        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id,
trackChanges: true);

    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in the
database.");

        return NotFound();
    }

    var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);
    patchDoc.ApplyTo(employeeToPatch, ModelState);

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the patch document");

        return UnprocessableEntity(ModelState);
    }

    _mapper.Map(employeeToPatch, employeeEntity);
    _repository.Save();

    return NoContent();
}

```

حالا باید این را با هم تست کنیم.

<https://localhost:5001/api/Companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[  
 {  
   "op": "remove",  
   "path": "/ageeeee"  
 }  
]  


The screenshot shows the Postman interface with a PATCH request to the specified URL. The request body is a JSON array with one object. The response status is 422 Unprocessable Entity, and the error message is 'The target location specified by path segment 'ageeeee' was not found.'

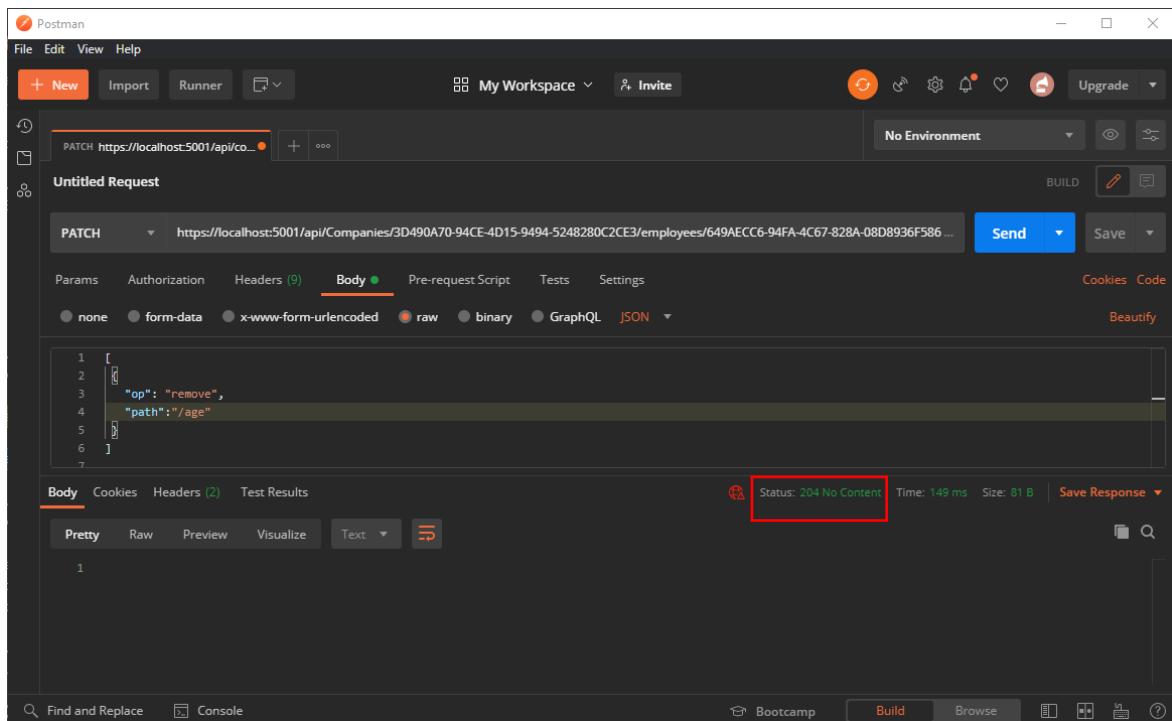

```

همان چیزی که انتظار داشتیم اتفاق افتاد، اما باز هم یک مشکل کوچک وجود دارد. سعی کنید یک عملیات حذف برای ریکوئست پایین بفرستید.

<https://localhost:5001/api/Companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[  
 {  
   "op": "remove",  
   "path": "/age"  
 }  
]
```



این ریکوئست با موفقیت انجام شد اما اگر یادتان باشد گفتیم که عملیات حذف، مقدار پراپرتی را به مقدار پیشفرض تنظیم می‌کند یعنی در اینجا باید به مقدار صفر تنظیم کند.

همانطور که می‌دانید، ما در کلاس `EmployeeForUpdateDto` یک اتریبوت `Range` داشتیم که اجازه نمی‌دهد مقدار پراپرتی، زیر ۱۸ باشد در حالی که الان مقدار پراپرتی زیر ۱۸ تنظیم شد.

مشکل کجاست؟

بیایید این مسئله را با هم بررسی کنیم.

```

var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

patchDoc.ApplyTo(employeeToPatch, ModelState);

if (!ModelState.IsValid)
{
    logger.LogError("Invalid model state for the patch document");
    return UnprocessableEntity(ModelState);
}

_mapper.Map(employeeToPatch, employeeEntity);

_repository.Save();

```

همانطور که در این کد می‌بینید، ما patchDoc را اعتبارسنجی می‌کنیم اما employeeEntity را بدون اعتبارسنجی، در دیتابیس ذخیره می‌کنیم.

بنابراین برای جلوگیری از ذخیره یک کارمند نامعتبر در دیتابیس، نیاز به اعتبارسنجی دیگری هم داریم.

```
[HttpPatch("{id}")]
public IActionResult PartiallyUpdateEmployeeForCompany(Guid companyId,
Guid id, [FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is null.");

        return BadRequest("patchDoc object is null");
    }
    var company = _repository.Company.GetCompany(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogWarning($"Company with id: {companyId} doesn't exist in the
database.");

        return NotFound();
    }

    var employeeEntity = _repository.Employee.GetEmployee(companyId, id,
trackChanges: true);

    if (employeeEntity == null)
    {
        _logger.LogWarning($"Employee with id: {id} doesn't exist in the
database.");

        return NotFound();
    }

    var employeeToPatch = _mapper.Map<EmployeeForUpdateDto>(employeeEntity);
    patchDoc.ApplyTo(employeeToPatch, ModelState);
    TryValidateModel(employeeToPatch);
```

```

if (!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the patch document");

    return UnprocessableEntity(ModelState);
}
_mapper.Map(employeeToPatch, employeeEntity);
_repository.Save();

return NoContent();
}

```

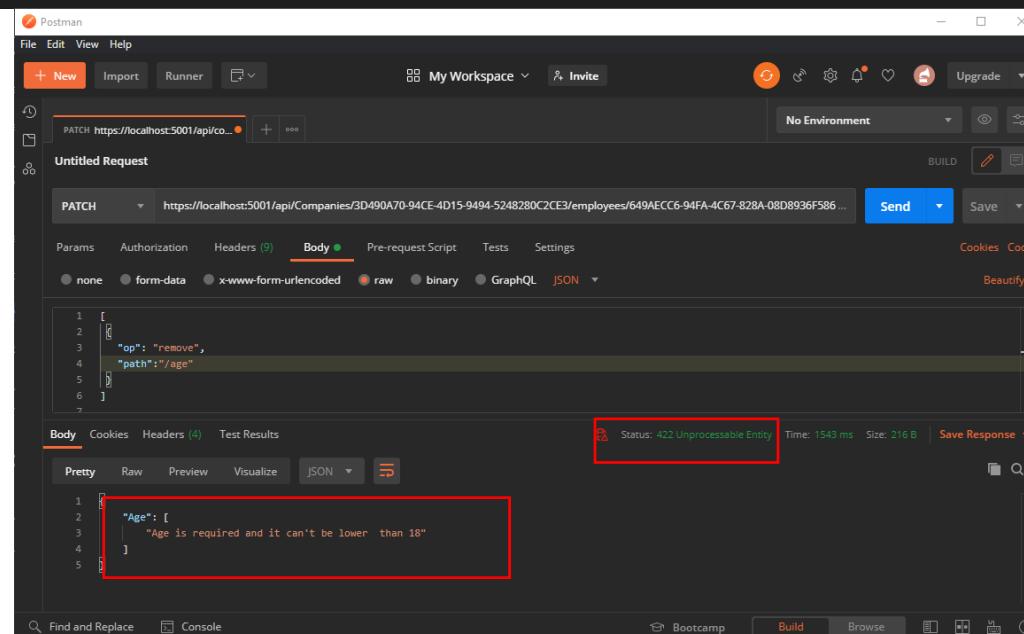
در کد بالا از متدهای TryValidateModel و ValidateModel استفاده نکردیم.
حالا هر خطایی، باعث شدن ModelState invalid می‌شود.

خب می‌توانیم دوباره این تست را انجام دهیم.

<https://localhost:5001/api/Companies/3D490A70-94CE-4D15-9494-5248280C2CE3/employees/649AECC6-94FA-4C67-828A-08D8936F5863>

body:

```
[
{
    "op": "remove",
    "path": "/age"
}]
```



همانطور که می‌بینید Status Code مورد انتظار را دریافت کردیم.

فصل هشتم : برنامه نویسی Action Filter و Async ها

آنچه خواهید آموخت:

- برنامه نویسی Async چیست؟
- کلمه های کلیدی await و async
- Action Filter چیست؟
- پیاده سازی Action Filter
- اعتبار سنجی با Action Filter در Dependency Injection

برنامه‌نویسی Async چیست؟

برنامه‌نویسی Async، تکنیکی است که به شما کمک می‌کند تا کار خود را از Thread اصلی اپلیکیشن جدا و در یک Thread دیگر انجام دهید؛ سپس زمانیکه کار تمام شد ترد نتیجه موفق یا ناموفق کار را به Thread اصلی اطلاع دهد.

با استفاده از برنامه‌نویسی Async می‌توان Scalability اپلیکیشن را افزایش و تا حد زیادی از کاهش پرفورمنس جلوگیری کنید.

اما برنامه‌نویسی Async چطور این اتفاقات خوب را به همرا دارد؟

در حالت Async وقتی ریکوئستی را به Thread اصلی ارسال می‌کنیم، این Thread کار را به یک Background Thread واگذار می‌کند؛ بنابراین برای یک ریکوئست دیگر آزاد است.

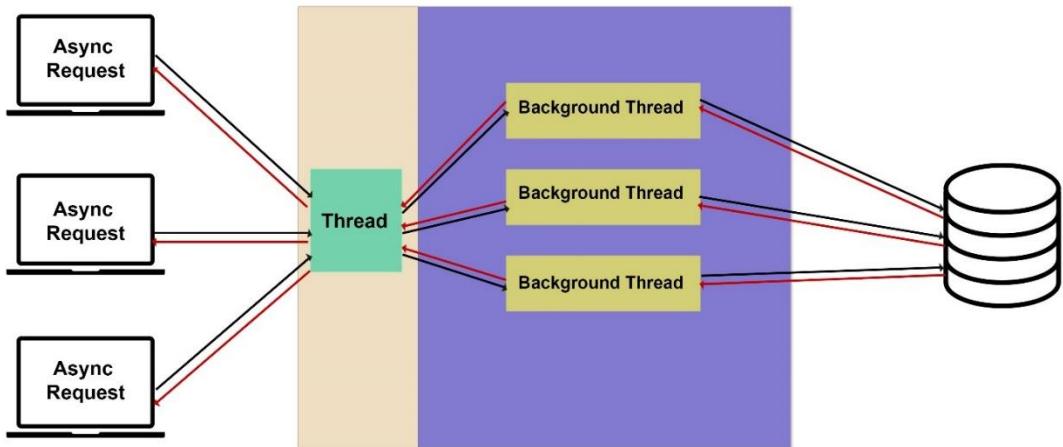
زمانیکه کار Background Thread تمام شد باید نتیجه را به Thread اصلی دهد. در آخر Thread اصلی نتیجه را به درخواست کننده برمی‌گرداند.

پس در نتیجه، چون ریکوئست ما مستقیماً با Thread اصلی کار نمی‌کند پس منظر هیچ ریسپانسی نمی‌ماند و در حقیقت بلاک نخواهد شد.

یک موضوع مهمی که باید بدانید این است که زمان طول کشیدن ریکوئست در حالت Async و Sync یک اندازه است و ما نمی‌توانیم در حالت Async آنرا سریعتر اجرا کنیم.

تنها مزیت Async نسبت به Sync این است که Thread اصلی در زمان اجرا شدن ریکوئست، بلاک نمی‌شود بنابراین Thread اصلی می‌تواند ریکوئست‌های دیگر را پردازش کند.

در پایین تصویری از کار Async را می‌بینید.



حالا که این موضوع برای شما روشن شد بیایید نحوه پیاده‌سازی کدهای Async در .NET 5.0 را یاد بگیریم.

کلمه‌های کلیدی `async` و `await`

کلمه کلیدی `async` نقش مهمی در برنامه‌نویسی Asynchronous دارد. با استفاده از این کلمه می‌توانیم به راحتی متدهای `async` بنویسیم.

`async Task<IEnumerable<Company>> GetAllCompaniesAsync()`

همانطور که در کد بالا می‌بینید، برای نوشتن متدهای `async` تنها باید کلمه کلیدی `async` را در کنار Return Type متدها اضافه کنید.

وقتی از کلمه کلیدی `async` استفاده می‌کنیم، می‌توان کلمه کلیدی `await` را در بدن متدها اضافه و خروجی متدهم تغییر کند.

`await FindAllAsync();`

در برنامه‌نویسی Async سه نوع Return Type داریم :

- `Task<TResult>` برای `async` متدهایی است که مقداری را برمی‌گردانند. این یعنی : اگر متدهم `int` می‌برگردد، پس متدهم `async` باید `Task<int>` را برگرداند و اگر متدهم `string` می‌برگردد پس متدهم `Task<string>` را برگرداند.

- برای Task : Task
متدهایی که مقداری را بر نمی‌گردانند کاربرد دارد. یعنی اگر متدهای sync را برنگردانند، بنابراین متدهای async باید Task برگردانند و ما می‌توانیم داخل متدهای sync استفاده کنیم، اما نیازی به return نداریم.
- از void هم می‌توانیم برای Event Handler استفاده کنیم. ما فقط برای void : از void هم می‌توانیم برای Asynchronous Event Handler ها که به نوع برگشتی void نیاز دارند، از void استفاده کنیم. به غیر از این مورد ما همیشه باید Task برگردانیم.

نکته!!

گاهی اوقات کد sync کندر از کد async عمل می‌کند به طور مثال: دستورات EF Core sync زمان زیادی برای اجرا شدن صرف می‌کند و این به دلیل کد اضافه‌ای است که برای مدیریت Thread ها داریم. بنابراین همیشه async انتخاب مناسبی نیست. به طور کلی هر کجا که امکان دارد باید از کد sync استفاده کنیم اما هر جا که کد sync ها کندر عمل کرد باید از حالت sync کمک بگیریم.

خب حالا بباید کدهای sync پروژه را ریفکتور کنیم.

Rيفکتور Repository

بباید ریفکتور را از CompanyRepository و ICompanyRepository شروع کنیم پس کدهای اینترفیس ICompanyRepository را با کدهای پایین تغییر دهید.

```
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Contracts.IServices
{
    public interface ICompanyRepository
    {
        Task<IEnumerable<Company>> GetAllCompaniesAsync(bool trackChanges);
        Task<Company> GetCompanyAsync(Guid companyId, bool trackChanges);
```

```

    void CreateCompany(Company company);
    Task<IEnumerable<Company>> GetByIdsAsync(IEnumerable<Guid> ids,
        bool trackChanges);
    void DeleteCompany(Company company);
}
}

```

بررسی کد :

• Create و Delete متدهای تغییری نکرده، چون در این متدها فقط state را به Added و Deleted تغییر می‌دهیم و تغییری در دیتابیس نداریم.
حالا مطابق با تغییرات اینترفیس، باید کلاس CompanyRepository هم تغییر کند.

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Repository.Repositories
{
    public class CompanyRepository : RepositoryBase<Company>,
        ICompanyRepository
    {
        public CompanyRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext): base(companyEmployeeDbContext)
        {
        }

        public async Task<IEnumerable<Company>> GetAllCompaniesAsync(bool
            trackChanges) =>
        await FindAll(trackChanges)
            .OrderBy(c => c.Name)
            .ToListAsync();
    }
}

```

```

public async Task<Company> GetCompanyAsync(Guid companyId, bool
trackChanges) =>
await FindByCondition(c => c.Id.Equals(companyId), trackChanges)
.SingleOrDefaultAsync();

public async Task<IEnumerable<Company>>
GetByIdsAsync(IEnumerable<Guid> ids, bool
trackChanges) =>
await FindByCondition(x => ids.Contains(x.Id), trackChanges)
.ToListAsync();

public void CreateCompany(Company company) => Create(company);

public void DeleteCompany(Company company)
{
    Delete(company);
}

}

```

خب حالا نوبت تغییر IRepositoryManager و IRepositoryManager است.

اگر کلاس RepositoryManager را باز کنید می بینید که یک متدهای Save وجود دارد که متدهای SaveChanges EF Core را صدای زند.

ما می خواهیم این متدهای async را کنیم پس IRepositoryManager و IRepositoryManager را باز و این متدهای را تغییر دهیم.

```

using System.Threading.Tasks;

namespace Contracts.IServices
{
    public interface IRepositoryManager
    {
        ICompanyRepository Company { get; }
        IEmployeeRepository Employee { get; }
        Task SaveAsync();
    }
}

```

}

: RepositoryManager کلاس

```
using Contracts.IServices;
using System.Threading.Tasks;
using Entities;

namespace Repository.Repositories
{
    public class RepositoryManager : IRepositoryManager
    {
        private CompanyEmployeeDbContext _repositoryContext;
        private ICompanyRepository _companyRepository;
        private IEmployeeRepository _employeeRepository;

        public RepositoryManager(CompanyEmployeeDbContext repositoryContext)
        {
            _repositoryContext = repositoryContext;
        }

        public ICompanyRepository Company
        {
            get
            {
                if (_companyRepository == null)
                    _companyRepository = new
                        CompanyRepository(_repositoryContext);

                return _companyRepository;
            }
        }

        public IE Employee
        {
            get
            {
                if (_employeeRepository == null)
                    _employeeRepository = new
                        EmployeeRepository(_repositoryContext);

                return _employeeRepository;
            }
        }
    }
}
```

۴۰۰

```

        }
    }

    public Task SaveAsync() => _repositoryContext.SaveChangesAsync();
}

}

```

بررسی کد :

- چون متدهای awaitable `ToListAsync` و ... متدهای `SaveAsync` هستند پس ما از کلمه کلیدی `await` استفاده می‌کنیم.

البته استفاده از کلمه کلیدی `await`، الزامی نیست اما اگر از آن استفاده نکنیم، متدهای `sync` به صورت `async` اجرا می‌شود و مطمئناً این هدف ما نیست.

Rifektor CompaniesController

در پایان کار باید تمام اکشن‌های `CompaniesController` را `async` کنیم. پس اول از متدهای `GetCompanies` شروع می‌کنیم.

```

[HttpGet]
public async Task<IActionResult> GetCompaniesAsync()
{
    var companies = await
        _repository.Company.GetAllCompaniesAsync(trackChanges: false);

    var companiesDto = _mapper.Map<IEnumerable<CompanyDto>>(companies);

    return Ok(companiesDto);
}

```

پایین را در این کنترلر اضافه کنید.

```
using System.Threading.Tasks;
```

بررسی کد :

- برای اینکه مشخص شود این متدهای `GetCompaniesAsync` است نام این متدهای `async` را کردیم.
- در این متدهای `Signature` Return Type را تغییر و کلمه کلیدی `async` را به متدهای `GetCompanies` اضافه می‌کنیم.

- در بدنه متدهم، متدها `await` کردیم. این همان کاری است که تقریباً در تمام اکشن‌ها باید انجام دهیم.
 - خوب بیایید باقی اکشن‌های را هم اصلاح کنیم.
- کدهای **CompaniesController**

```

using AutoMapper;
using CompanyEmployee.API.Infrastructure.ModelBinders;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Entities.Models;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies")]
    [ApiController]
    public class CompaniesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public CompaniesController(IRepositoryManager repository,
        ILoggerManager logger,
        IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }

        [HttpGet]
        public async Task<IActionResult> GetCompaniesAsync()
        {
    }
}

```

```

    var companies = await
    _repository.Company.GetAllCompaniesAsync(trackChanges:
        false);

    var companiesDto =
    _mapper.Map<IEnumerable<CompanyDto>>(companies);

    return Ok(companiesDto);
}

[HttpGet("{id}", Name = "CompanyById")]
public async Task<IActionResult> GetCompanyAsync(Guid id)
{
    var company = await _repository.Company.GetCompanyAsync(id,
    trackChanges: false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in
            the database.");

        return NotFound();
    }
    else
    {
        var companyDto = _mapper.Map<CompanyDto>(company);

        return Ok(companyDto);
    }
}

[HttpGet("collection/({ids})", Name = "CompanyCollection")]
public async Task<IActionResult>
GetCompanyCollectionAsync([ModelBinder(BinderType =
typeof(ArrayModelBinder))] IEnumerable<Guid> ids)
{
    if (ids == null)
    {
        _logger.LogError("Parameter ids is null");

        return BadRequest("Parameter ids is null");
    }

    var companyEntities = await
    _repository.Company.GetByIdsAsync(ids, trackChanges:

```

```

        false);

        if (ids.Count() != companyEntities.Count())
        {
            _logger.LogError("Some ids are not valid in a
                collection");

            return NotFound();
        }

        var companiesToReturn =
        _mapper.Map<IEnumerable<CompanyDto>>(companyEntities);

        return Ok(companiesToReturn);
    }

    [HttpPost]
    public async Task<IActionResult>
    CreateCompanyAsync([FromBody]CompanyForCreationDto
    company)
    {
        if (company == null)
        {
            _logger.LogError("CompanyForCreationDto object sent from
                client is null.");

            return BadRequest("CompanyForCreationDto object is null");
        }
        if (!ModelState.IsValid)
        {
            _logger.LogError("Invalid model state for the
                CompanyForCreationDto object");

            return UnprocessableEntity(ModelState);
        }

        var companyEntity = _mapper.Map<Company>(company);
        _repository.Company.CreateCompany(companyEntity);
        await _repository.SaveAsync();

        var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

```

```

        return CreatedAtRoute("CompanyById", new { id =
companyToReturn.Id },
companyToReturn);
    }

[HttpPost("collection")]
public async Task<IActionResult>
CreateCompanyCollectionAsync([FromBody]
IEnumerable<CompanyForCreationDto> companyCollection)
{
    if (companyCollection == null)
    {
        _logger.LogError("Company collection sent from client is
null.");

        return BadRequest("Company collection is null");
    }

    var companyEntities =
_mapper.Map<IEnumerable<Company>>(companyCollection);

    foreach (var company in companyEntities)
    {
        _repository.Company.CreateCompany(company);
    }

    await _repository.SaveAsync();
    var companyCollectionToReturn =
_mapper.Map<IEnumerable<CompanyDto>>(companyEntities);
    var ids = string.Join(",",
companyCollectionToReturn.Select(c
=> c.Id));

    return CreatedAtRoute("CompanyCollection", new { ids },
companyCollectionToReturn);
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteCompanyAsync(Guid id)
{
    var company = await _repository.Company.GetCompanyAsync(id,
trackChanges: false);
}

```

```

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in
the database.");

        return NotFound();
    }

    _repository.Company.DeleteCompany(company);
    await _repository.SaveAsync();

    return NoContent();
}

[HttpPut("{id}")]
public async Task<IActionResult> UpdateCompanyAsync(Guid id,
[FromBody] CompanyForUpdateDto
company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForUpdateDto object sent from
client is null.");

        return BadRequest("CompanyForUpdateDto object is null");
    }
    if (!ModelState.IsValid)
    {

        _logger.LogError("Invalid model state for the
CompanyForUpdateDto object");

        return UnprocessableEntity(ModelState);
    }

    var companyEntity = await
_repository.Company.GetCompanyAsync(id, trackChanges:
true);

    if (companyEntity == null)
    {
        _logger.LogInfo($"Company with id: {id} doesn't exist in
the database.");

```

```

        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}

}

```

خب حالا نوبت Employee است. تمام مراحل بالا را برای این Entity تکرار می‌کنیم.

کدهای : IEmployeeRepository

```

using Entities.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId, bool trackChanges);
        Task<Employee> GetEmployeeAsync(Guid companyId, Guid id, bool trackChanges);
        void CreateEmployeeForCompany(Guid companyId, Employee employee);
        void DeleteEmployee(Employee employee);
    }
}

```

کدهای : EmployeeRepository

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using System;

```

```

using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
        IEmployeeRepository
    {

        public EmployeeRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext) : base(companyEmployeeDbContext)
        {
        }

        public async Task<IEnumerable<Employee>> GetEmployeesAsync (Guid
            companyId, bool trackChanges) =>
            await FindAll(trackChanges)
                .OrderBy(c => c.Name)
                .ToListAsync();

        public async Task<Employee> GetEmployeeAsync (Guid companyId,
            Guid id, bool trackChanges) =>
            await FindByCondition(e => e.CompanyId.Equals(companyId) &&
            e.Id.Equals(id), trackChanges).SingleOrDefaultAsync();

        public void CreateEmployeeForCompany(Guid companyId, Employee
            employee)
        {
            employee.CompanyId = companyId;
            Create(employee);
        }

        public void DeleteEmployee(Employee employee)
        {
            Delete(employee);
        }

    }
}

```

: EmployeesController

```
using AutoMapper;
using Contracts.IServices;
using Entities.DataTransferObjects;
using Entities.Models;
using Microsoft.AspNetCore.JsonPatch;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/companies/{companyId}/employees")]
    [ApiController]
    public class EmployeesController : ControllerBase
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;

        public EmployeesController(IRepositoryManager repository,
            ILoggerManager logger,
            IMapper mapper)
        {
            _repository = repository;
            _logger = logger;
            _mapper = mapper;
        }

        [HttpGet(Name = "GetEmployeeForCompany")]
        public async Task<IActionResult> GetEmployeesForCompanyAsync(Guid companyId)
        {
            var company = await
                _repository.Company.GetCompanyAsync(companyId, trackChanges:
                    false);

            if (company == null)
```

```

    {
        _logger.LogInfo($"Company with id: {companyId} doesn't
exist in the database.");

        return NotFound();
    }

    var employeesFromDb =await
_repository.Employee.GetEmployeesAsync(companyId,
trackChanges: false);

    var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDto);
}

[HttpGet("{id}")]
public async Task<IActionResult> GetEmployeeForCompanyAsync(Guid
companyId, Guid id)
{
    var company =await
_repository.Company.GetCompanyAsync(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't
exist in the database.");

        return NotFound();
    }

    var employeeDb =await
_repository.Employee.GetEmployeeAsync(companyId, id,
trackChanges: false);

    if (employeeDb == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in
the database.");
    }
}

```

```

        return NotFound();
    }

    var employee = _mapper.Map<EmployeeDto>(employeeDb);

    return Ok(employee);
}

[HttpPost]
public async Task<IActionResult> CreateEmployeeForCompanyAsync(Guid
companyId, [FromBody]
EmployeeForCreationDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForCreationDto object sent from
client is null.");

        return BadRequest("EmployeeForCreationDto object is
null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the
EmployeeForCreationDto object");

        return UnprocessableEntity(ModelState);
    }

    var company = await
_repository.Company.GetCompanyAsync(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't
exist in the database.");

        return NotFound();
    }
}

```

```

    var employeeEntity = _mapper.Map<Employee>(employee);

    _repository.Employee.CreateEmployeeForCompany(companyId,
        employeeEntity);
    await _repository.SaveAsync();

    var employeeToReturn =
        _mapper.Map<EmployeeDto>(employeeEntity);

    return
        CreatedAtRoute("GetEmployeeForCompany",
            new
            {
                companyId,
                id = employeeToReturn.Id
            },
            employeeToReturn);
}

[HttpDelete("{id}")]
public async Task<IActionResult>
DeleteEmployeeForCompanyAsync(Guid companyId, Guid id)
{
    var company = await
        _repository.Company.GetCompanyAsync(companyId, trackChanges:
    false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't
            exist in the database.");

        return NotFound();
    }

    var employeeForCompany =await
        _repository.Employee.GetEmployeeAsync(companyId, id,
        trackChanges: false);

    if (employeeForCompany == null)
    {

```

```

        _logger.LogError($"Employee with id: {id} doesn't exist in
the database.");

        return NotFound();
    }

    _repository.Employee.DeleteEmployee(employeeForCompany);
    await _repository.SaveAsync();

    return NoContent();
}

[HttpPut("{id}")]
public async Task<IActionResult>
UpdateEmployeeForCompanyAsync(Guid companyId, Guid id, [FromBody]
EmployeeForUpdateDto employee)
{
    if (employee == null)
    {
        _logger.LogError("EmployeeForUpdateDto object sent from
client is null.");

        return BadRequest("EmployeeForUpdateDto object is null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the
EmployeeForUpdateDto object");

        return UnprocessableEntity(ModelState);
    }

    var company =await
    _repository.Company.GetCompanyAsync(companyId, trackChanges:
false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't
exist in the database.");
    }
}

```

```

        return NotFound();
    }
    var employeeEntity =await
        _repository.Employee.GetAsync(companyId, id,
        trackChanges:
        true);

    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in
            the database.");
        return NotFound();
    }

    _mapper.Map(employee, employeeEntity);
    await _repository.SaveAsync();

    return NoContent();
}

[HttpPatch("{id}")]
public async Task<IActionResult>
PartiallyUpdateEmployeeForCompanyAsync(Guid companyId, Guid id,
[FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is
            null.");
        return BadRequest("patchDoc object is null");
    }

    var company =await
        _repository.Company.GetAsync(companyId, trackChanges:
        false);

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't
            exist in the database.");
    }
}

```

```

        return NotFound();
    }

    var employeeEntity =await
    _repository.Employee.GetEmployeeAsync(companyId, id,
    trackChanges: true);

    if (employeeEntity == null)
    {
        _logger.LogInfo($"Employee with id: {id} doesn't exist in
        the database.");
        return NotFound();
    }

    var employeeToPatch =
    _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

    patchDoc.ApplyTo(employeeToPatch, ModelState);
    TryValidateModel(employeeToPatch);

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the patch
        document");
        return UnprocessableEntity(ModelState);
    }

    _mapper.Map(employeeToPatch, employeeEntity);

    await _repository.SaveAsync();

    return NoContent();
}

}

```

Action Filter چیست؟

سرور همیشه، ریکوئست را به کنترلر و اکشن مربوطه هدایت می‌کند؛ اما گاهی نیاز است که قبل یا بعد از اجرای اکشن متده، کدی اجرا شود.

فیلترهای موجود در.NET، یک روش عالی برای حل این مسئله هستند. فیلتر یک کلاس است که با آن می‌توانید، کدی را قبل یا بعد از اجرای یک اکشن متده اجرا کنید.

با فیلترها می‌توانیم، کدهای قابل استفاده مجدد بنویسیم و اکشن‌های ما همیشه تمیز و قابل نگهداری باشند.

انواع فیلترها :

• **Authorization Filter**: این فیلتر تعیین می‌کند که آیا کاربر برای ریکوئست جاری

مجاز هست یا خیر.

• **Resource Filter**: این فیلتر بلافاصله بعد از Authorization Filter اجرا می‌شود و برای

Performance و Caching بسیار کاربردی است.

• **Action Filter**: این فیلتر قبل و بعد از اکشن متده اجرا می‌شود.

• **Exception Filter**: این فیلتر برای مدیریت اکسپشن، قبل از پر شدن Response استفاده می‌شود.

• **Result Filter**: این فیلتر قبل و بعد از نتیجه اکشن متده اجرا می‌شود.

در این فصل می‌خواهیم در مورد Action Filter و نحوه استفاده از آن صحبت کنیم.

پیاده‌سازی Action Filter

برای ایجاد Action Filter، باید کلاسی ایجاد کنیم که اینترفیس IActionFilter و یا ActionFilterAttribute از کلاس IAsyncActionFilter ارث بری کند.

نکته!!

کلاس ActionFilterAttribute لینترفیس IActionFilter و چند اینترفیس دیگر را پیاده‌سازی کرده است.

```
public abstract class ActionFilterAttribute : Attribute,  
IActionFilter, IFilterMetadata, IAsyncActionFilter, IResultFilter,  
IAAsyncResultFilter, IOrderedFilter
```

برای نوشتن کدهای Sync و Async، می‌توانید اینترفیس IActionFilter یا IAsyncActionFilter را پیاده‌سازی کنید.

اگر OnActionExecuting را پیاده‌سازی کنید، پس باید کدهای خود را در متدهای OnActionExecuted و بنویسید.

```
using Microsoft.AspNetCore.Mvc.Filters;  
  
namespace CompanyEmployee.API.Infrastructure.Filters  
{  
    namespace ActionFilters.Filters  
    {  
        public class ActionFilterExample : IActionFilter  
        {  
            public void OnActionExecuting(ActionExecutingContext context)  
            {  
                // our code before action executes  
            }  
            public void OnActionExecuted(ActionExecutedContext context)  
            {  
                // our code after action executes  
            }  
        }  
    }  
}
```

اگر IAyncActionFilter را پیاده‌سازی کنید، پس کدهای خود را باید در متدهای OnActionExociationAsync بنویسید.

```
using Microsoft.AspNetCore.Mvc.Filters;  
using System.Threading.Tasks;  
  
namespace CompanyEmployee.API.Infrastructure.Filters  
{  
    namespace ActionFilters.Filters  
    {  
        public class AsyncActionFilterExample : IAsyncActionFilter
```

```

    {
        public async Task
        OnActionExecutionAsync(ActionExecutingContext context,
        ActionExecutionDelegate next)
        {
            // execute any code before the action executes
            var result = await next();
            // execute any code after the action executes
        }
    }
}

```

سطح Action Filter

نیز، مانند سایر فیلترها می‌تواند به سطوح مختلف اضافه شود. به طور مثال :

یا به صورت عمومی یا به اکشن و کنترلر اضافه می‌شود.

اگر بخواهیم از فیلتر در سطح عمومی استفاده کنیم، باید آن را در متدهای AddControllers رجیستر کنیم.

```

services.AddControllers(config =>
{
    config.Filters.Add(new GlobalFilterExample());
});

```

اما اگر بخواهیم Action Filter را، به عنوان یک سرویس در سطح اکشن متدهای کنترلر استفاده کنیم؛ باید آن را به عنوان یک سرویس در متدهای ConfigureServices رجیستر کنیم.

```

services.AddScoped<ActionFilterExample>();
services.AddScoped<ControllerFilterExample>();

```

در پایان برای استفاده از فیلتر باید آن را به عنوان ServiceType، در بالای اکشن متدهای کنترلر قرار دهیم.

```

using CompanyEmployee.API.Infrastructure.Filters.ActionFilters.Filters;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{

```

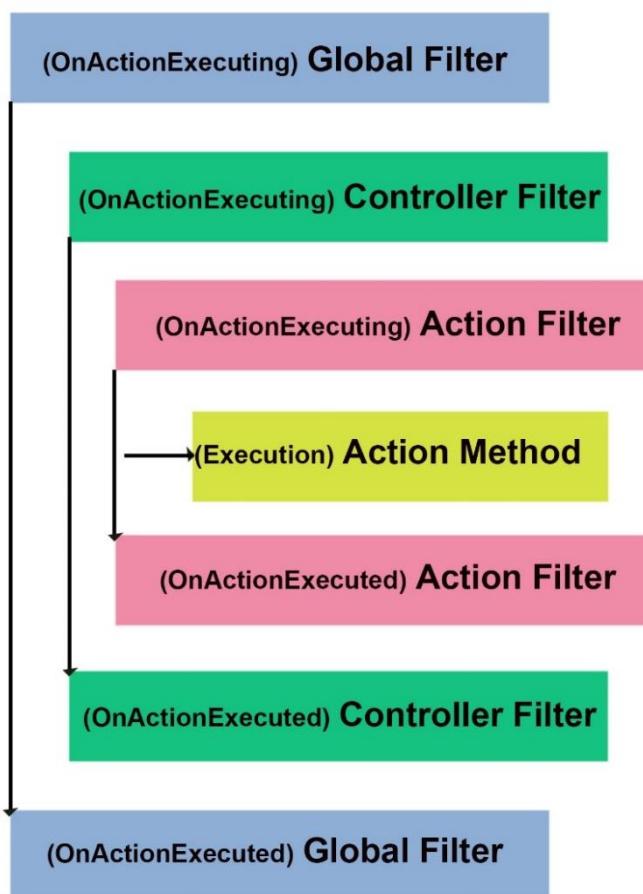
```

[ServiceFilter(typeof(ControllerFilterExample))]
[Route("api/[controller]")]
[ApiController]
public class TestController : ControllerBase
{
    [HttpGet]
    [ServiceFilter(typeof(ActionFilterExample))]
    public IEnumerable<string> Get()
    {
        return new string[] { "example", "data" };
    }
}

```

ترتیب اجرا شدن Filter ها

ترتیب اجرای فیلترها به شرح زیر است :



البته می توانیم با افزودن Order به کد پایین، این ترتیب را تغییر دهیم.

```

using CompanyEmployee.API.Infrastructure.Filters.ActionFilters.Filters;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace CompanyEmployee.API.Controllers
{
    [ServiceFilter(typeof(ControllerFilterExample))]
    [Route("api/[controller]")]
    [ApiController]
    public class TestController : ControllerBase
    {
        [HttpGet]
        [ServiceFilter(typeof(ActionFilterExample), Order = 1)]
        public IEnumerable<string> Get()
        {
            return new string[] { "example", "data" };
        }
    }
}

```

یا چیزی شبیه به این در بالای اکشن متدهای بنویسیم.

```

[ServiceFilter(typeof(ActionFilterExample2), Order = 1)]
[ServiceFilter(typeof(ActionFilterExample), Order = 2)]

```

اعتبارسنجی با Action Filter

اکشن متدهای ما، بدون try-catch و خواناتر شدند چون یک مدیریت اکسپشن عمومی وجود دارد که این موضوع را هندل می‌کند. اما باز هم می‌توان این اکشن‌ها را بهینه‌تر کرد.

ما می‌توانیم کدهای اعتبارسنجی را هم، از اکشن متدها جدا کنیم و با این کار کدهای تمیز و قابل نگهداری داشته باشیم.

اگر به اکشن‌های POST و PUT در CompaniesController نگاه کنید، می‌بینید که اعتبارسنجی Company در هر دو تکرار شده است.

```

[HttpPost]
public async Task<IActionResult>
CreateCompanyAsync([FromBody] CompanyForCreationDto
company)

```

```

{
    if (company == null)
    {
        _logger.LogError("CompanyForCreationDto object sent from client is
null.");

        return BadRequest("CompanyForCreationDto object is null");
    }

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the CompanyForCreationDto
object");

        return UnprocessableEntity(ModelState);
    }

    var companyEntity = _mapper.Map<Company>(company);
    _repository.Company.CreateCompany(companyEntity);
    await _repository.SaveAsync();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyId", new { id = companyToReturn.Id },
companyToReturn);
}

```

```

[HttpPut("{id}")]
public async Task<IActionResult> UpdateCompanyAsync(Guid id, [FromBody]
CompanyForUpdateDto
company)
{
    if (company == null)
    {
        _logger.LogError("CompanyForUpdateDto object sent from client is
null.");

        return BadRequest("CompanyForUpdateDto object is null");
    }
}

```

```

if (!ModelState.IsValid)
{
    _logger.LogError("Invalid model state for the CompanyForUpdateDto
object");

    return UnprocessableEntity(ModelState);
}

var companyEntity = await _repository.Company.GetCompanyAsync(id,
trackChanges:
true);

if (companyEntity == null)
{
    _logger.LogInfo($"Company with id: {id} doesn't exist in the
database.");

    return NotFound();
}

_mapper.Map(company, companyEntity);
await _repository.SaveAsync();

return NoContent();
}

```

خب می توانیم این کدها را از این اکشن متد ها جدا و به یک کلاس Action Filter ببریم. پس در فولدر Infrastructure، یک فولدر جدید بنام ActionFilters ایجاد و سپس یک کلاس بنام ValidationFilterAttribute در آن اضافه نمایید.

```

using Contracts.IServices;
using Microsoft.AspNetCore.Mvc.Filters;

namespace CompanyEmployee.API.Infrastructure.ActionFilters
{
    public class ValidationFilterAttribute : IActionFilter
    {
        private readonly ILoggerManager _logger;
        public ValidationFilterAttribute(ILoggerManager logger)
        {
            _logger = logger;
        }
    }
}

```

```

        }

        public void OnActionExecuting(ActionExecutingContext context)
        { }

        public void OnActionExecuted(ActionExecutedContext context) {
        }

    }
}

```

حالا می توانیم متدهای OnActionExecuting و OnActionExecuted را تغییر دهیم.

```

using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System.Linq;

namespace CompanyEmployee.API.Infrastructure.ActionFilters
{
    public class ValidationFilterAttribute : IActionFilter
    {
        private readonly ILoggerManager _logger;
        public ValidationFilterAttribute(ILoggerManager logger)
        {
            _logger = logger;
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            var action = context.RouteData.Values["action"];
            var controller = context.RouteData.Values["controller"];
            var param = context.ActionArguments
                .SingleOrDefault(x =>
                    x.Value.ToString().Contains("Dto")).Value;

            if (param == null)
            {
                _logger.LogError($"Object sent from client is null.
Controller: {controller}, action: {action} ");
            }
        }
    }
}

```

```

        context.Result = new BadRequestObjectResult($"Object is
null. Controller:{ controller}, action: {action} ");

    return;
}

if (!context.ModelState.IsValid)
{
    _logger.LogError($"Invalid model state for the object.
Controller: {controller}, action: {action} ");

    context.Result = new
UnprocessableEntityObjectResult(context.ModelState);
}
}

public void OnActionExecuted(ActionExecutedContext context) { }
}
}

```

سپس این اکشن فیلتر را در متده `ConfigureServices` رجیستر کنید.

ابتدا `Namespace` پایین را در کلاس `Startup` اضافه کنید.

```
using CompanyEmployee.API.Infrastructure.ActionFilters;
```

و بعد از آن، کد پایین را در متده `ConfigureServices` بنویسید.

```
services.AddScoped<ValidationFilterAttribute>();
```

در پایان، باید کد اعتبارسنجی را از اکشن متدها حذف و این اکشن فیلتر را به عنوان سرویس استفاده کنید.

ابتدا `Namespace` پایین را در `CompaniesController` اضافه کنید.

```
using CompanyEmployee.API.Infrastructure.ActionFilters;
```

سپس متدهای پایین را ریفکتور نمایید.

```
[HttpPost]
```

```
[ServiceFilter(typeof(ValidationFilterAttribute))]
```

```

public async Task<IActionResult>
CreateCompanyAsync([FromBody]CompanyForCreationDto
company)
{
    var companyEntity = _mapper.Map<Company>(company);
    _repository.Company.CreateCompany(companyEntity);
    await _repository.SaveAsync();

    var companyToReturn = _mapper.Map<CompanyDto>(companyEntity);

    return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id },
companyToReturn);
}

[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> UpdateCompanyAsync(Guid id, [FromBody]
CompanyForUpdateDto
company)
{
    var companyEntity = await _repository.Company.GetCompanyAsync(id,
trackChanges:
true);

    if (companyEntity == null)
    {
        _logger.LogError($"Company with id: {id} doesn't exist in the
database.");

        return NotFound();
    }

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}

```

عالی شد.

حالا این اکشن‌متدها (بدون کدهای اعتبارسنجی) بسیار تمیز‌تر و خواناتر شده‌اند.

اگر به عنوان مثال یک ریکوئست POST با مدل نامعتبر بفرستیم، نتیجه موردنظر را می‌گیریم.

<https://localhost:5001/api/companies>

body:

```
{  
    "name": "Zahra",  
    "address":null  
}
```

The screenshot shows the Postman interface with a POST request to `https://localhost:5001/api/Companies`. The request body is a JSON object with `"name": "Zahra"` and `"address":null`. The response status is `422 Unprocessable Entity`, and the error message indicates required fields for `Address` and `Country`.

Action Filter در Dependency Injection

اگر اکشن‌متد `DeleteCompanyAsync` و یا `UpdateCompanyAsync` را بررسی کنید، می‌بینید که در آن `Company` را با `Id` از دیتابیس واکشی و سپس چک می‌کند که آیا اطلاعاتی از سمت دیتابیس بازگشت داده شده یا خیر؟

```
if (company == null)  
{  
    _logger.LogError($"Company with id: {id} doesn't exist in the  
    database.");
```

```
        return NotFound();
    }
```

این تکه کد، همان چیزی است که ما می‌توانیم در کلاس Action Filter داشته باشیم، تا برخی اکشن‌های متدها بتوانند از آن استفاده مجدد کنند.

البته باید در نظر داشته باشیم که در این کلاس، باید ریپازیتوری را با استفاده از Dependency Injection تزریق کنیم.

خب با گفتن این موضوع، اجازه دهید یک کلاس دیگر با نام ValidateCompanyExistsAttribute در فولدر ActionFilters ایجاد کنیم.

```
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Infrastructure.ActionFilters
{
    public class ValidateCompanyExistsAttribute : IAsyncActionFilter
    {
        private readonly IRepositoryManager _repository;
        private readonly ILoggerManager _logger;

        public ValidateCompanyExistsAttribute(IRepositoryManager repository,
                                              ILoggerManager logger)
        {
            _repository = repository;
            _logger = logger;
        }

        public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            var trackChanges =
                context.HttpContext.Request.Method.Equals("PUT");
            var id = (Guid)context.ActionArguments["id"];
        }
    }
}
```

```

var company = await _repository.Company.GetCompanyAsync(id,
trackChanges);

if (company == null)
{
    _logger.LogError($"Company with id: {id} doesn't exist in
the database.");
    context.Result = new NotFoundResult();
}

else
{
    context.HttpContext.Items.Add("company", company);
    await next();
}
}

}

```

: بررسی کد :

- ما در این کد، Entity خود را به صورت Async واکشی می‌کنیم پس باید نسخه async اکشن فیلتر را استفاده کنیم. در اینجا دو نکته وجود دارد که باید به آن توجه کنید.
- اولین نکته این است که، اگر ریکوئست برابر PUT باشد می‌توانیم trackChanges را با تنظیم true کنیم.
- نکته دوم این که، اگر Entity در دیتابیس باشد آن را در HttpContext ذخیره می‌کنیم؛ چون ما در اکشن متدها به Entity احتیاج داریم و نمی‌خواهیم دو بار از دیتابیس کوئری بگیریم.

خب بیایید این ActionFilter را در متدها Register کنیم.

```
services.AddScoped<ValidateCompanyExistsAttribute>();
```

حالا نوبت تغییر اکشن متدها است.

```

[HttpDelete("{id}")]
[ServiceFilter(typeof(ValidateCompanyExistsAttribute))]
public async Task<IActionResult> DeleteCompanyAsync(Guid id)
{
    var company = HttpContext.Items["company"] as Company;

```

```

    _repository.Company.DeleteCompany(company);
    await _repository.SaveAsync();

    return NoContent();
}

[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
[ServiceFilter(typeof(ValidateCompanyExistsAttribute))]
public async Task<IActionResult> UpdateCompanyAsync(Guid id, [FromBody]
CompanyForUpdateDto
company)
{
    var company = HttpContext.Items["company"] as Company;

    _mapper.Map(company, companyEntity);
    await _repository.SaveAsync();

    return NoContent();
}

```

همانطور که می‌بینید، اکشن‌متد‌ها بدون کدهای تکراری بسیار تمیز‌تر شده‌اند.

می‌توانید در Postman این اکشن‌متد‌ها را با ریکوئست‌های Delete و Put تست کنید.

تمام مراحلی که در بالا انجام دادیم برای EmployeesController هم باید تکرار شود (به جز برخی اختلافات در اجرای فیلتر).

باید ببینیم چطور این کار را انجام دهیم.

- در فolder ActionFilters یک کلاس با نام

ValidateEmployeeForCompanyExistsAttribute ایجاد کنید.

```

using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Infrastructure.ActionFilters
{

```

```

public class ValidateEmployeeForCompanyExistsAttribute :  

IAsyncActionFilter  

{  

    private readonly IRepositoryManager _repository;  

    private readonly ILoggerManager _logger;  

    public ValidateEmployeeForCompanyExistsAttribute(IRepositoryManager  

repository, ILoggerManager logger)  

{  

    _repository = repository;  

    _logger = logger;  

}  

    public async Task OnActionExecutionAsync(ActionExecutingContext  

context, ActionExecutionDelegate next)  

{  

    var method = context.HttpContext.Request.Method;  

var trackChanges = (method.Equals("PUT") ||  

method.Equals("PATCH")) ? true : false;  

var companyId = (Guid)context.ActionArguments["companyId"];  

var company = await  

_repository.Company.GetCompanyAsync(companyId, false);  

if (company == null)  

{  

    _logger.LogError($"Company with id: {companyId} doesn't  

exist in the database.");  

    context.Result = new NotFoundResult();  

    return;
}  

var id = (Guid)context.ActionArguments["id"];  

var employee = await  

_repository.Employee.GetEmployeeAsync(companyId, id,  

trackChanges);  

if (employee == null)
{
    _logger.LogError($"Employee with id: {id} doesn't exist in  

the database.");
}
}

```

```

        context.Result = new NotFoundResult();
    }
    else
    {
        context.HttpContext.Items.Add("employee", employee);
        await next();
    }
}

}

```

- مرحله دوم رجيستر کردن اين کلاس در متده Configuration است.

```
services.AddScoped<ValidateEmployeeForCompanyExistsAttribute>();
```

در پايان باید کدهای اکشن متدهای PUT، Delete و Patch را در EmployeesController تغییر دهید. اما قبل از انجام این تغییرات حتما Namespace پایین را اضافه کنید.

```
using CompanyEmployee.API.Infrastructure.ActionFilters;
```

اکشن متدهای PUT و Delete و Patch :

```
[HttpDelete("{id}")]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult> DeleteEmployeeForCompanyAsync(Guid companyId, Guid id)
{
    var employeeForCompany = HttpContext.Items["employee"] as Employee;
    _repository.Employee.DeleteEmployee(employeeForCompany);

    await _repository.SaveAsync();

    return NoContent();
}

[HttpPut("{id}")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult> UpdateEmployeeForCompanyAsync(Guid companyId, Guid id, [FromBody] EmployeeForUpdateDto employee)
{
    var employeeEntity = HttpContext.Items["employee"] as Employee;
```

```

    _mapper.Map(employee, employeeEntity);

    await _repository.SaveAsync();

    return NoContent();
}

[HttpPatch("{id}")]
[ServiceFilter(typeof(ValidateEmployeeForCompanyExistsAttribute))]
public async Task<IActionResult>
PartiallyUpdateEmployeeForCompanyAsync(Guid companyId, Guid id, [FromBody]
JsonPatchDocument<EmployeeForUpdateDto> patchDoc)
{
    if (patchDoc == null)
    {
        _logger.LogError("patchDoc object sent from client is null.");

        return BadRequest("patchDoc object is null");
    }

    var employeeEntity = HttpContext.Items["employee"] as Employee;
    var employeeToPatch =
        _mapper.Map<EmployeeForUpdateDto>(employeeEntity);

    patchDoc.ApplyTo(employeeToPatch, ModelState);
    TryValidateModel(employeeToPatch);

    if (!ModelState.IsValid)
    {
        _logger.LogError("Invalid model state for the patch document");

        return UnprocessableEntity(ModelState);
    }

    _mapper.Map(employeeToPatch, employeeEntity);
    await _repository.SaveAsync();

    return NoContent();
}

```

این تغییرات را می‌توان با ریکوئست‌های Postman تست کنید.

Paging, Filtering, Searching : فصل نهم

آنچه خواهید آموخت:

► آشنایی و پیاده‌سازی Paging, Filtering, Searching

Paging چیست؟

تا حالا ویژگی‌های زیادی به پروژه اضافه کردیم اما هنوز کارهایی وجود دارد که می‌تواند Web API را بهتر کند. یکی از این کارها، اعمال Paging به API است.

Paging یعنی، فقط قسمتی از نتیجه را از API بگیریم. تصور کنید میلیون‌ها رکورد در دیتابیس وجود دارد و با یک ریکوئست بخواهید همه‌ی این اطلاعات را به یکباره واکشی کنید. به نظر شما چه اتفاقی می‌افتد؟

واکشی تمام اطلاعات یک روش بسیار ناکارآمد است و علاوه بر مشکلات پرفورمنسی و کندی API، می‌تواند تأثیرات بدی بر اپلیکیشن یا سخت افزاری که در آن اجرا می‌شود داشته باشد. همچنین منابع حافظه‌ی هر کلاینت، محدود است بنابراین تعداد نتایج هم باید محدود شود.

خب حالا که متوجه صورت مسئله شدیم، بباید با اعمال Paging از این پیامدها جلوگیری و تنها تعداد مشخصی از نتایج را به کلاینت برگردانیم.

پیاده‌سازی Paging

اگر اکشن‌متدهای GetEmployeesForCompanyAsync را بررسی کنید، می‌بینید که تمام اطلاعات کارمندان یک شرکت، به صورت یکجا برگردانده می‌شود. در اینجا می‌خواهیم ویژگی Paging را به این اکشن‌متدهای اضافه کنیم تا از مشکلات پرفورمنسی جلوگیری شود.

توجه داشته باشید که قرار نیست کدهای Base Repository تغییر کند و یا اینکه منطق بیزینسی خاصی را در کنترلر پیاده‌سازی نمایید.

ما تنها کاری که باید انجام دهیم این است که تعداد نتایج و شماره صفحه را از URL بگیریم و متناسب با درخواست کلاینت، تعدادی نتایج را برگردانیم.

<https://localhost:5001/api/companies/companyId/employees?pageNumber=3&pageSize=2>

در URL بالا، کلاینت درخواست می‌کند تا اپلیکیشن در صفحه سوم، اطلاعات دو کارمند را برگرداند.

نکته!!

باید API را طوری محدود کنیم که، حتی در صورتی که ریکوئست پایین صدا زده شد، اطلاعات همه کارمندان برگردانده نشود.

<https://localhost:5001/api/companies/companyId/employees>

قبل از هر گونه تغییر در اکشن متدها، چند کار باید انجام شود.

- برای درخواست شماره صفحه و تعداد کارمندان، باید از پارامترهای کوئری استفاده کنیم؛ پس در اکشن متدها [FromQuery] استفاده نمایید.
- برای نگهداری پارامترهای Employee Entity، به یک کلاس EmployeeParameters نیاز داریم.

پس اولین گام ایجاد یک فolder به نام RequestFeatures در پروژه Entities است.
درون این فolder یک کلاس با نام RequestParameters اضافه نمایید.

```
namespace Entities.RequestFeatures
{
    public abstract class RequestParameters
    {
        const int maxPageSize = 50;
        public int PageNumber { get; set; } = 1;
        private int _pageSize = 10;
        public int PageSize
        {
            get
            {
                return _pageSize;
            }
            set
            {
                _pageSize = (value > maxPageSize) ? maxPageSize : value;
            }
        }
    }
}
```

```

public class EmployeeParameters : RequestParameters
{
}

}

```

بررسی کد:

- کلاس RequestFeatures یک کلاس Abstract است که شامل پردازشی‌های مشترک تمام Entity‌ها می‌باشد.
- کلاس EmployeeParameters هم، پارامترهای خاص Entity را در خود نگه می‌دارد. البته الان پارامتری ندارد و این پارامترها در طول کار اضافه خواهد شد.
- در کلاس RequestFeatures یک Constant به نام maxPageSize داریم که API را محدود به نمایش حداکثر ۵۰ رکورد اطلاعات می‌کند.
- ما دو پردازشی public به نام PageSize و PageNumber داریم که اگر توسط کلاینت تنظیم نشده باشد، به ترتیب بر روی ۱ و ۱۰ تنظیم خواهد شد.
- حالا می‌توانیم به EmployeesController برگردیم و using کلاس EmployeeParameters را اضافه کنیم.

```
using Entities.RequestFeatures;
```

- پس از این تغییرات، باید منطق ریپازیتوری را پیاده‌سازی کنیم. پس همانند کد پایین، متدهای IEmployeeRepository را در اینترفیس GetEmployeesAsync و کلاس EmployeeRepository اصلاح کنید.

: IEmployeeRepository کدهای

```

using Entities.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Entities.RequestFeatures;

namespace Contracts.IServices

```

```

{
    public interface IEmployeeRepository
    {
        Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId, bool
trackChanges);
        Task<Employee> GetEmployeeAsync(Guid companyId, Guid id, bool
trackChanges);
        Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId,
EmployeeParameters employeeParameters, bool trackChanges);
        void CreateEmployeeForCompany(Guid companyId, Employee employee);
        void DeleteEmployee(Employee employee);
    }
}

```

: کدهای EmployeeRepository

```

using Contracts.IServices;
using Entities;
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using Entities.RequestFeatures;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
IEmployeeRepository
    {

        public EmployeeRepository(CompanyEmployeeDbContext
companyEmployeeDbContext) : base(companyEmployeeDbContext)
        {
        }
        public async Task<IEnumerable<Employee>> GetEmployeesAsync (Guid
companyId, bool trackChanges) =>
            await FindAll(trackChanges)
                .OrderBy(c => c.Name)
    }
}

```

```

        .ToListAsync();

    public async Task<Employee> GetEmployeeAsync (Guid companyId, Guid
id, bool trackChanges) =>
    await FindByCondition(e => e.CompanyId.Equals(companyId) &&
e.Id.Equals(id), trackChanges).SingleOrDefaultAsync();

public async Task<IEnumerable<Employee>> GetEmployeesAsync(Guid
companyId, EmployeeParameters employeeParameters, bool
trackChanges) =>
    await FindByCondition(e => e.CompanyId.Equals(companyId),
trackChanges)
        .OrderBy(e => e.Name)
        .Skip((employeeParameters.PageNumber - 1) *
employeeParameters.PageSize)
        .Take(employeeParameters.PageSize)
        .ToListAsync();
}

public void CreateEmployeeForCompany(Guid companyId, Employee
employee)
{
    employee.CompanyId = companyId;
    Create(employee);
}

public void DeleteEmployee(Employee employee)
{
    Delete(employee);
}
}

}

```

• خب حالا باید بدنه اکشن متده GetEmployeesForCompanyAsync را تغییر دهیم.

```

[HttpGet(Name = "GetEmployeeForCompany")]
public async Task<IActionResult> GetEmployeesForCompanyAsync(Guid
companyId, [FromQuery] EmployeeParameters employeeParameters)

```

```

{
    var company = await _repository.Company.GetCompanyAsync(companyId,
        trackChanges: false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {companyId} doesn't exist
            in the database.");

        return NotFound();
    }

    var employeesFromDb = await
        _repository.Employee.GetEmployeesAsync(companyId,
        employeeParameters, trackChanges: false);

    var employeesDto =
        _mapper.Map<IList<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDto);
}

```

قبل تست اين API، حتما کارمندان بيشتری اضافه کنيد.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?pageNumber=1&pageSize=3>

KEY	VALUE	DESCRIPTION
pageNumber	1	

```

1 [
2   {
3     "id": "86dba8c0-d178-41e7-938c-ed49778fb52a",
4     "name": "Ali Bayat",
5     "age": 30,
6     "position": "Backend developer"
7   },
8   {
9     "id": "021ca3c1-8deb-4af8-ac94-2159a8479811",
10    "name": "Sara Bayat",
11    "age": 35,
12    "position": "Frontend developer"
13  },
14  {
15    "id": "548d8c97-738a-4a7b-7fbd-08d893793288",
16    "name": "Zahra Bayat",
17    "age": 28,
18    "position": "IT"
19  }
20 ]

```

همانطور که می‌بینید در صفحه اول، ۳ کارمند را درخواست کردیم و نتیجه بالا بازگشت داده شد.

اگر این همان چیزی است که شما هم بدست آورده اید پس در مسیر صحیح هستید. بیایید نتیجه را در دیتابیس هم بررسی کنیم.

The screenshot shows the SQL Query window with the following T-SQL code:

```
SQLQuery1.sql - (lo...yEmployee (sa (52))*
exec sp_executesql N'SELECT [e].[EmployeeId], [e].[Age], [e].[CompanyId], [e].[Name], [e].[Position]
FROM [Employees] AS [e]
WHERE [e].[CompanyId] = @_companyId_0
ORDER BY [e].[Name]
--OFFSET @_p_1 ROWS FETCH NEXT @_p_1 ROWS ONLY',N'@_companyId_0 uniqueidentifier,@_p_1 int',
 @_companyId_0='C9D4C053-49B6-410C-BC78-2D54A9991870', @_p_1=0
```

The Results window displays the following data:

EmployeeId	Age	CompanyId	Name	Position
86DBA8C0-D178-41E7-938C-ED49778FB52A	30	C9D4C053-49B6-410C-BC78-2D54A9991870	Ali Bayat	Backend developer
021CA3C1-0DEB-4AFD-AE94-2159A8479811	35	C9D4C053-49B6-410C-BC78-2D54A9991870	Sara Bayat	Frontend developer
548D8C97-738A-4A7B-7FBD-08D893793288	28	C9D4C053-49B6-410C-BC78-2D54A9991870	Zahra Bayat	IT

ارتقای Paging

تا اینجا، فقط یک List به کلاینت برمی‌گردندیم؛ اما ممکن است بخواهید به جای یک ساده، یک PagedList به کلاینت برگردانید.

Klassی است که، از کلاس List ارث بری می‌کند و ما می‌توانیم منطق Skip/Take را هم، در آن داشته باشیم.

بنابراین بیایید در مسیر Entities/RequestFeatures یک کلاس با نام MetaData ایجاد کنیم.

```
namespace Entities.RequestFeatures
{
    public class MetaData
    {
        public int CurrentPage { get; set; }
        public int TotalPages { get; set; }
        public int PageSize { get; set; }
        public int TotalCount { get; set; }
        public bool HasPrevious => CurrentPage > 1;
        public bool HasNext => CurrentPage < TotalPages;
    }
}
```

```
}
```

```
}
```

حالا در همین فolder، کلاس PagedList را اضافه کنید.

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

namespace Entities.RequestFeatures
{
    public class PagedList<T> : List<T>
    {
        public MetaData MetaData { get; set; }
        public PagedList(List<T> items, int count, int pageNumber, int pageSize)
        {
            MetaData = new MetaData
            {
                TotalCount = count,
                PageSize = pageSize,
                CurrentPage = pageNumber,
                TotalPages = (int)Math.Ceiling(count / (double)pageSize)
            };

            AddRange(items);
        }

        public static PagedList<T> ToPagedList(IEnumerable<T> source, int pageNumber, int pageSize)
        {
            var count = source.Count();

            var items = source
                .Skip((pageNumber - 1) * pageSize)
                .Take(pageSize).ToList();

            return new PagedList<T>(items, count, pageNumber, pageSize);
        }
    }
}
```

بررسی کد:

- همانطور که می‌بینید منطق Skip/Take را در ToPagedList اضافه کردیم.
- پرایپریتی‌های کلاس MetaData را هم، با مقادیر ورودی Constructor پر کردیم، تا برای Metadata ریسپانس استفاده شود.
- اگر به کلاس MetaData نگاه کنید می‌بینید که، اگر CurrentPage بزرگتر از ۱ باشد، پرایپریتی HasPrevious برابر true و در صورتی که CurrentPage کوچکتر از تعداد کل صفحات باشد، پرایپریتی HasNext محاسبه خواهد شد.
- پرایپریتی TotalPages هم، با تقسیم تعداد آیتم‌ها به عدد Page Size بدست می‌آید و سپس این عدد به عدد بزرگتر گرد خواهد شد.

خب حالا بباید به همین ترتیب EmployeeRepository و EmployeesController را هم تغییر دهیم.

کدهای : IEmployeeRepository

```
using Entities.Models;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Entities.RequestFeatures;

namespace Contracts.IServices
{
    public interface IEmployeeRepository
    {
        Task<IEnumerable<Employee>> GetEmployeesAsync(Guid companyId, bool trackChanges);
        Task<Employee> GetEmployeeAsync(Guid companyId, Guid id, bool trackChanges);
        Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId,
            EmployeeParameters employeeParameters, bool trackChanges);
        void CreateEmployeeForCompany(Guid companyId, Employee employee);
        void DeleteEmployee(Employee employee);
    }
}
```

}

: EmployeeRepository کدهای

```
using Contracts.IServices;
using Entities;
using Entities.Models;
using Microsoft.EntityFrameworkCore;
using Entities.RequestFeatures;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Repository.Repositories
{
    public class EmployeeRepository : RepositoryBase<Employee>,
        IEmployeeRepository
    {

        public EmployeeRepository(CompanyEmployeeDbContext
            companyEmployeeDbContext) : base(companyEmployeeDbContext)
        {
        }

        public async Task<IEnumerable<Employee>> GetEmployeesAsync (Guid
            companyId, bool trackChanges) =>
            await FindAll(trackChanges)
                .OrderBy(c => c.Name)
                .ToListAsync();

        public async Task<Employee> GetEmployeeAsync (Guid companyId, Guid
            id, bool trackChanges) =>
            await FindByCondition(e => e.CompanyId.Equals(companyId) &&
            e.Id.Equals(id), trackChanges).SingleOrDefaultAsync();

        public async Task<PagedList<Employee>> GetEmployeesAsync(Guid
            companyId, EmployeeParameters employeeParameters, bool
            trackChanges)
        {
        }
    }
}
```

```

        var employees = await FindByCondition(e =>
            e.CompanyId.Equals(companyId),
            trackChanges)
            .OrderBy(e => e.Name)
            .ToListAsync();

        return PagedList<Employee>
            .ToPagedList(employees, employeeParameters.PageNumber,
            employeeParameters.PageSize);
    }

    public void CreateEmployeeForCompany(Guid companyId,
Employee employee)
{
    employee.CompanyId = companyId;
    Create(employee);
}

public void DeleteEmployee(Employee employee)
{
    Delete(employee);
}

}
}

```

خب حالا باید بدنہ اکشن متدا GetEmployeesForCompanyAsync را تغییر دهیم.

ابتدا Namespace پایین را در کنترلر اضافه کنید.

```
using Newtonsoft.Json;
```

سپس بدنہ اکشن متدا تغییر دهید.

```
[HttpGet(Name = "GetEmployeeForCompany")]
```

```

public async Task<IActionResult> GetEmployeesForCompanyAsync(Guid
companyId, [FromQuery] EmployeeParameters employeeParameters)
{
    var company = await _repository.Company.GetCompanyAsync(companyId,
trackChanges: false);

```

```

if (company == null)
{
    _logger.LogInfo($"Company with id: {companyId} doesn't exist
    in the database.");

    return NotFound();
}

var employeesFromDb = await
_repository.Employee.GetEmployeesAsync(companyId,
employeeParameters, trackChanges: false);

Response.Headers.Add("X-Pagination",
JsonConvert.SerializeObject(employeesFromDb.MetaData));

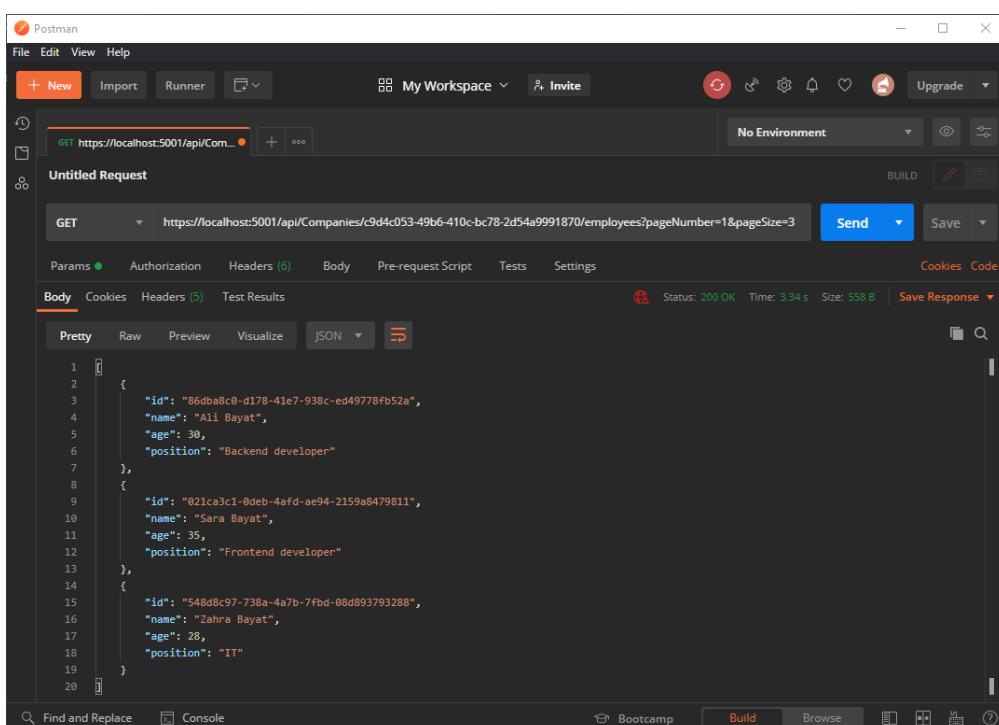
var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

return Ok(employeesDto);
}

```

حال اگر ریکوئست پایین را دوباره ارسال کنیم، می‌بینیم که نتیجه با ریکوئستی که بالاتر انجام دادیم یکی است.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?pageNumber=1&pageSize=3>



تفاوت این ریکوئست با قبلی در این است که، حالا اطلاعات مفیدی در هدر ریسپانس-X-Pagination داریم.

body	Cookies	Headers (5)	Test Results
KEY	VALUE		
Date	Wed, 16 Dec 2020 11:36:02 GMT		
Content-Type	application/json; charset=utf-8		
Server	Kestrel		
Content-Length	305		
X-Pagination	{"CurrentPage":1,"TotalPages":1,"PageSize":3,"TotalCount":3,"HasPrevious":false,"HasNext":false}		

همانطور که می‌بینید تمام پرایپریتی‌های Metadata در X-Pagination وجود دارند. ما می‌توانیم از این اطلاعات برای ایجاد لینک به صفحه قبلی و بعدی استفاده کنیم.

چیست Filtering

mekanizmi است که، نتایج را با توجه به برخی معیارها و اکشی می‌کند و ما می‌توانیم به اطلاعات دقیق‌تر برسیم.

فیلترها را می‌توان با توجه به نوع پرایپریتی، رنج عددی، محدوده تاریخ و یا هر چیز دیگری نوشت. در هنگام اجرای فیلتر، همیشه محدود به مجموعه گزینه‌های از پیش تعریف شده هستید که می‌توانید در ریکوئست خود آن‌ها را تنظیم کنید.

در سمت Frontend به صورت Checkbox یا Radio Button، Dropdown یا پیاده‌سازی Filtering می‌شود. این نوع پیاده‌سازی شما را به گزینه‌هایی محدود می‌کند. به عنوان مثال :

یک وب سایت فروش اتومبیل را در نظر بگیرید. هنگام فیلتر کردن اتومبیل‌های مورد نظر خود، می‌توانید گزینه‌های پایین را انتخاب کنید :

- تولیدکننده ماشین در یک لیست Dropdown
- مدل ماشین در یک لیست Dropdown
- یک Radio Button برای مشخص کردن اینکه ماشین جدید است یا خیر.
- قیمت خودرو در یک TextBox
- ..

بنابراین ریکوئست برای این وب سایت، می‌تواند چیزی شبیه به این است.

https://bestcarswebsite.com/sale?manufacturer=ford&model=expedition&state=used&city=washington&price_from=30000&price_to=50000

یا شاید هم شبیه پایین باشد.

[https://bestcarswebsite.com/sale/filter?data\[manufacturer\]=ford&\[model\]=expedition&\[state\]=used&\[city\]=washington&\[price_from\]=30000&\[price_to\]=50000](https://bestcarswebsite.com/sale/filter?data[manufacturer]=ford&[model]=expedition&[state]=used&[city]=washington&[price_from]=30000&[price_to]=50000)

حالا که مفهوم Filtering را دانستید بباید آن را پیاده‌سازی کنیم.

پیاده‌سازی ASP.NET Core در Filtering

در کلاس Employee، پراپرتی به نام Age وجود دارد که بر روی آن می‌توانیم عمل Filtering را پیاده کنیم. به طور مثال :

کارمندان بین ۲۶ تا ۲۹ سال.

<https://localhost:5001/api/companies/companyId/employees?minAge=26&maxAge=29>

قبل‌اً Paging را در کنترلر پیاده‌سازی کردیم؛ پس زیرساخت‌های لازم برای Filtering را داریم.

ما در کلاس EmployeeParameters، پارامترهایی را برای ریکوئست Paging اضافه کردیم پس باید همین کلاس را برای عمل Filtering هم انجام دهیم.

```
public class EmployeeParameters : RequestParameters
{
    public uint MinAge { get; set; }
    public uint MaxAge { get; set; } = int.MaxValue;
    public bool ValidAgeRange => MaxAge > MinAge;
}
```

بررسی کد :

- در این کلاس دو پراپرتی uint با نام‌های MinAge و MaxAge اضافه کردیم تا عدد سال منفی وارد نشود.
- از آنجا که مقدار پیش فرض uint عدد صفر است، پس نیازی نیست مقدار پراپرتی‌های آن را صفر تعریف کنیم.

- در اینجا یک اعتبارسنجی ساده، با نام ValidAgeRange که هدفش، بررسی این است که آیا MaxAge بیشتر از MinAge است. اگر اینطور نباشد، باید به کلاینت اطلاع دهیم که مقادیرش اشتباه است.

خب حالا که پارامترهای خود را آماده کردیم بیایید یک اعتبارسنجی برای بررسی اکشن متدهای GetEmployeesForCompanyAsync اضافه کنیم.

```
[HttpGet(Name = "GetEmployeeForCompany")]
public async Task<IActionResult> GetEmployeesForCompanyAsync(Guid
companyId, [FromQuery] EmployeeParameters employeeParameters)
{
    var company = await _repository.Company.GetCompanyAsync(companyId,
trackChanges: false);

    if (!employeeParameters.ValidAgeRange)
        return BadRequest("Max age can't be less than min age.");

    if (company == null)
    {
        _logger.LogInfo($"Company with id: {companyId} doesn't exist
in the database.");

        return NotFound();
    }

    var employeesFromDb = await
_repository.Employee.GetEmployeesAsync(companyId,
employeeParameters, trackChanges: false);

    Response.Headers.Add("X-Pagination",
JsonConvert.SerializeObject(employeesFromDb.MetaData));

    var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

    return Ok(employeesDto);
}
```

همانطور که می‌بینید چیز زیادی اضافه نکردیم، فقط اعتبارسنجی انجام دادیم و نتیجه را با یک پیام BadRequest به کلاینت برگرداندیم.

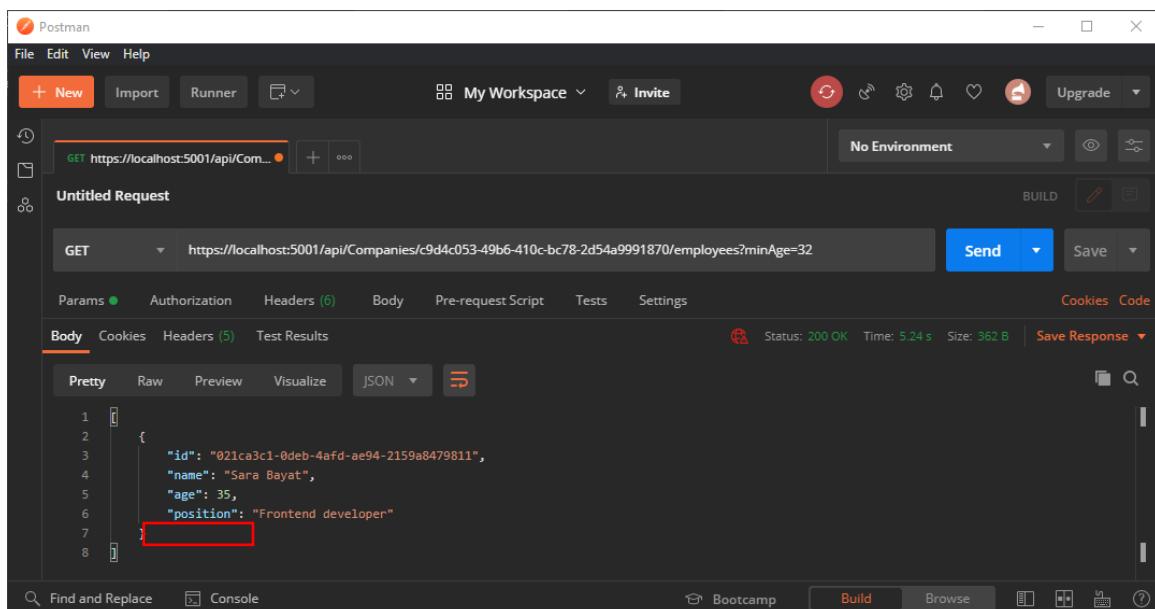
حالا باید یک تغییر کوچک در متدهای GetEmployeesAsync کلاس EmployeeRepository انجام دهیم، تا تمام کارمندان با سن بین MinAge و MaxAge را بدست آوریم.

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid  
companyId, EmployeeParameters employeeParameters, bool trackChanges)  
{  
    var employees = await FindByCondition(e =>  
        e.CompanyId.Equals(companyId) && (e.Age >= employeeParameters.MinAge  
&& e.Age <= employeeParameters.MaxAge),  
        trackChanges)  
        .OrderBy(e => e.Name)  
        .ToListAsync();  
  
    return PagedList<Employee>  
        .ToPagedList(employees, employeeParameters.PageNumber,  
        employeeParameters.PageSize);  
}
```

ارسال و تست Filtering

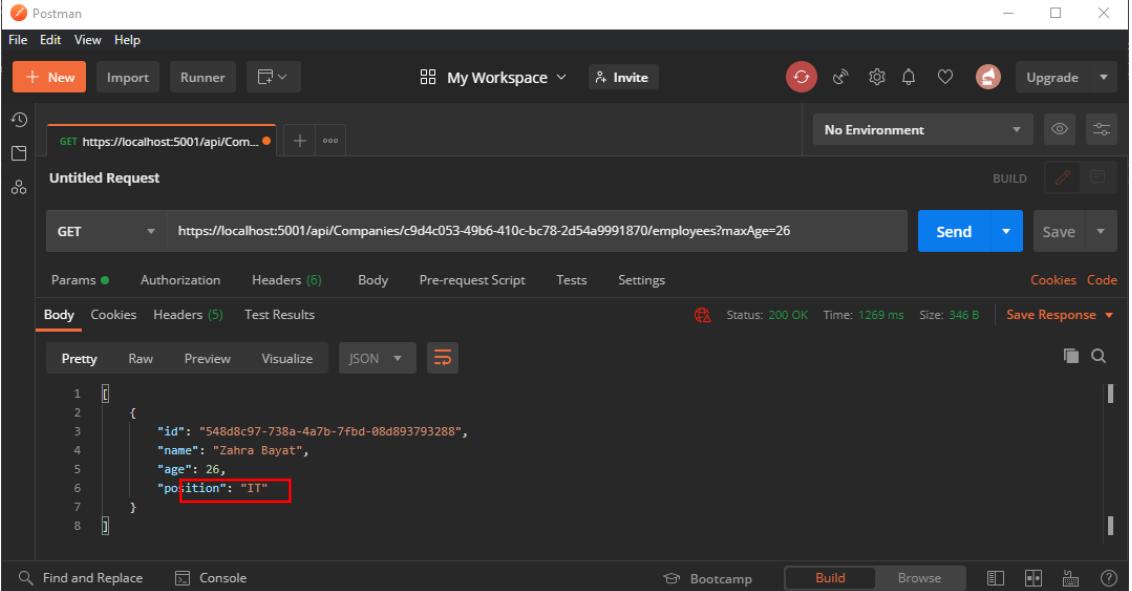
باید اولین ریکوئست را با یک پارامتر MinAge ارسال کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?minAge=32>



حالا باید با پارامتر MaxAge ارسال کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?maxAge=26>

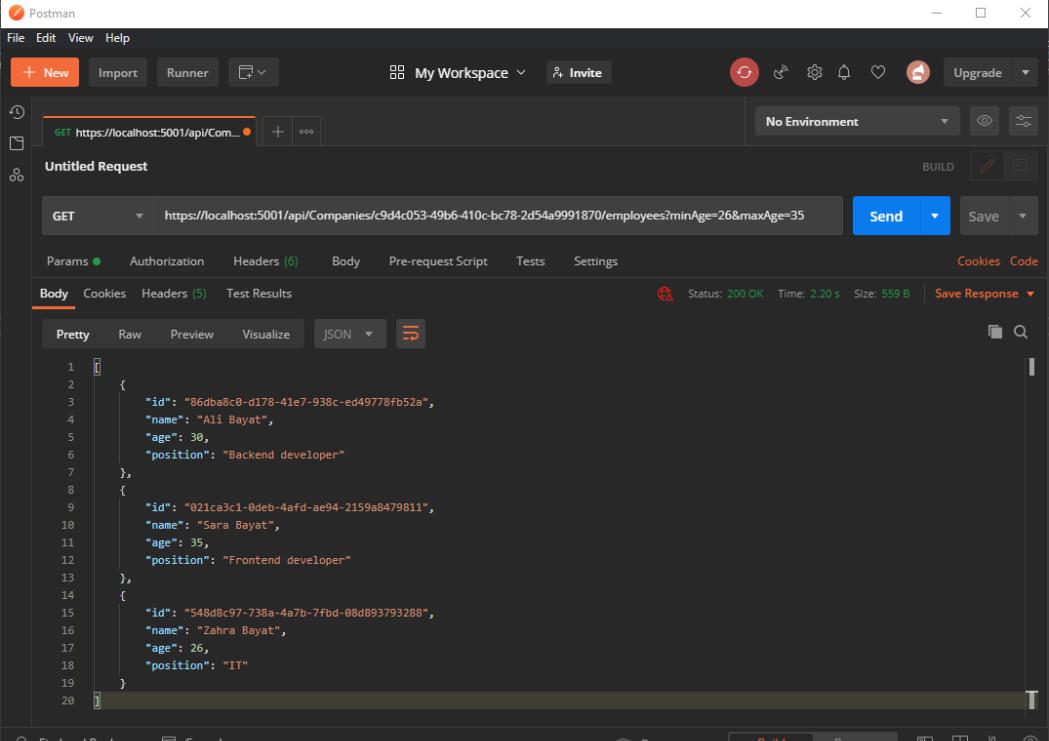


The screenshot shows the Postman application interface. A GET request is made to the specified URL. The response body is displayed in Pretty JSON format, showing one employee record with an age of 26 and a position of "IT".

```
1 [ { 2   "id": "548d8c97-738a-4a7b-7fb0-08d893793288", 3   "name": "Zahra Bayat", 4   "age": 26, 5   "position": "IT" 6 } ]
```

حالا ترکیب هر دو پارامتر را استفاده کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?minAge=26&maxAge=35>

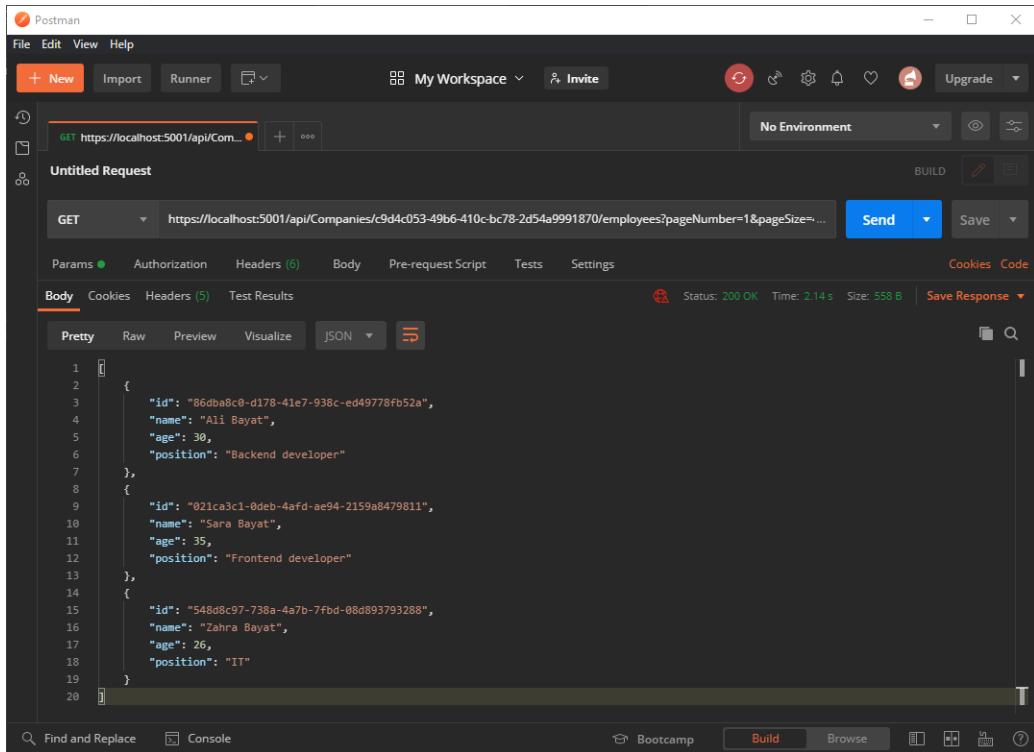


The screenshot shows the Postman application interface. A GET request is made to the specified URL with filtering parameters. The response body is displayed in Pretty JSON format, showing three employee records with ages 26, 30, and 35, and their respective positions.

```
1 [ { 2   "id": "86dba8c0-d178-41e7-938c-ed49778fb52a", 3   "name": "Ali Bayat", 4   "age": 30, 5   "position": "Backend developer" 6 }, 7   { 8     "id": "021ca3c1-0deb-4af0-ae94-2159a8479811", 9     "name": "Sara Bayat", 10    "age": 35, 11    "position": "Frontend developer" 12 }, 13   { 14     "id": "548d8c97-738a-4a7b-7fb0-08d893793288", 15     "name": "Zahra Bayat", 16     "age": 26, 17     "position": "IT" 18   } ]
```

و در پایان می‌توانیم Paging را با Filtering تست کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?pageNumber=1&pageSize=&minAge=26&maxAge=35>



عالی شد.

پیاده‌سازی Filtering تمام شد. حالا می‌توانیم Searching را پیاده‌سازی کنیم.

Searching چیست؟

جستجو تقریباً در هر وب سایتی وجود دارد. جستجو یکی از ویژگی‌هایی است که می‌تواند API را درست یا خراب کند؛ و میزان دشواری آن، بسته به مشخصات اپلیکیشن دارد.

شاید در وب سایتی که با ساختارش آشنا هستید، بتوانید هر چیزی را پیدا کنید؛ اما اگر وارد یک سایت بزرگ شوید مطمئناً نمی‌دانید که چه مطالبی وجود دارد و چه چیزی را باید جستجو کنید. اینجاست که نیاز به جستجو دارید.

در این پروژه می‌خواهیم یک جستجو ساده برای یافتن اطلاعات یک کارمند داشته باشیم، اما قبل از پیاده‌سازی، بیایید کمی در مورد Searching بدانیم.

- در Searching معمولاً یک ورودی داریم که از آن برای جستجوی هر چیزی در وب سایت استفاده می‌شود. بنابراین شما عبارتی را به API ارسال می‌کنید و API مسئول

استفاده از آن عبارت برای یافتن نتیجه‌ای است که با آن مطابقت داشته باشد. به طور مثال:

در وب سایت اتومبیل، ما از قسمت جستجو، مدل اتومبیل را تایپ می‌کنیم و تمام نتایج مربوط به آن اتومبیل به ما بازگشت داده می‌شود.

- ما می‌توانیم پیاده‌سازی Searching را طوری انجام دهیم که اگر کاربر عبارت را بدون کوتیشن جستجو کرد، تمام نتایج مربوط به عبارت برگرداند شود. و اگر عبارت را در کوتیشن نوشت، باید دقیقاً نتایج با عبارت مطابقت داشته باشد.

استفاده از جستجو به معنای این نیست که، ما نمی‌توانیم از Filtering استفاده کنیم. استفاده از Filtering و جستجو در کنار هم، کاملاً منطقی است بنابراین باید هنگام کدنویسی این موضوع را در نظر بگیریم.

تئوری کافی است بیایید کمی کد بزنیم.

پیاده‌سازی جستجو در اپلیکیشن

چون قبل از ساخت مورد نیاز برای Paging و Filtering اضافه کردیم، پس پیاده‌سازی Searching به هیچ وجه پیچیده نیست. تنها کاری که باید انجام دهیم این است که این زیر ساخت را کمی گسترش دهیم.

چیزی که می‌خواهیم به آن بررسیم شبیه به API پایین است.

<https://localhost:5001/api/companies/companyId/employees?searchTerm=Zahra>

در این جستجو می‌خواهیم تمام کسانی که نامشان Zahra است را برگردانیم. در نظر داشته باشید که این جستجو باید همراه با Paging و Filtering کار کند.

اولین کاری که باید انجام دهیم اضافه کردن یک پارامتر جستجو در کلاس EmployeeParameters است.

```
public class EmployeeParameters : RequestParameters
{
    public uint MinAge { get; set; }
```

```

    public uint MaxAge { get; set; } = int.MaxValue;
    public bool ValidAgeRange => MaxAge > MinAge;
    public string SearchTerm { get; set; }
}

```

به همین سادگی.

حالا می‌توانیم کوئری "searchTerm=name" را بنویسیم.

خب حالا برای انجام جستجو، باید متدهای داشته باشیم. پس در پروژه Repository فolderی با نام Extensions ایجاد و سپس کلاس RepositoryEmployeeExtensions را در آن اضافه کنید.

```

using Entities.Models;
using System.Linq;

namespace Repository.Extensions
{
    public static class RepositoryEmployeeExtensions
    {
        public static IQueryable<Employee> FilterEmployees(this
            IQueryable<Employee> employees, uint minAge, uint maxAge) =>
            employees.Where(e => (e.Age >= minAge && e.Age <= maxAge));

        public static IQueryable<Employee> Search(this
            IQueryable<Employee> employees, string searchTerm)
        {
            if (string.IsNullOrWhiteSpace(searchTerm))
                return employees;

            var lowerCaseTerm = searchTerm.Trim().ToLower();

            return employees.Where(e =>
                e.Name.ToLower().Contains(lowerCaseTerm));
        }
    }
}

```

تا اینجا برای آپدیت کوئری‌هایی که در EmployeeRepository بود اکستنشن متدهای نوشته‌یم. حالا تنها کاری که باید انجام دهیم این است که Namespace پایین را به کلاس GetEmployeesAsync و سپس متدهای EmployeeRepository اضافه کنیم.

```
using Repository.Extensions;
```

متدهای GetEmployeesAsync

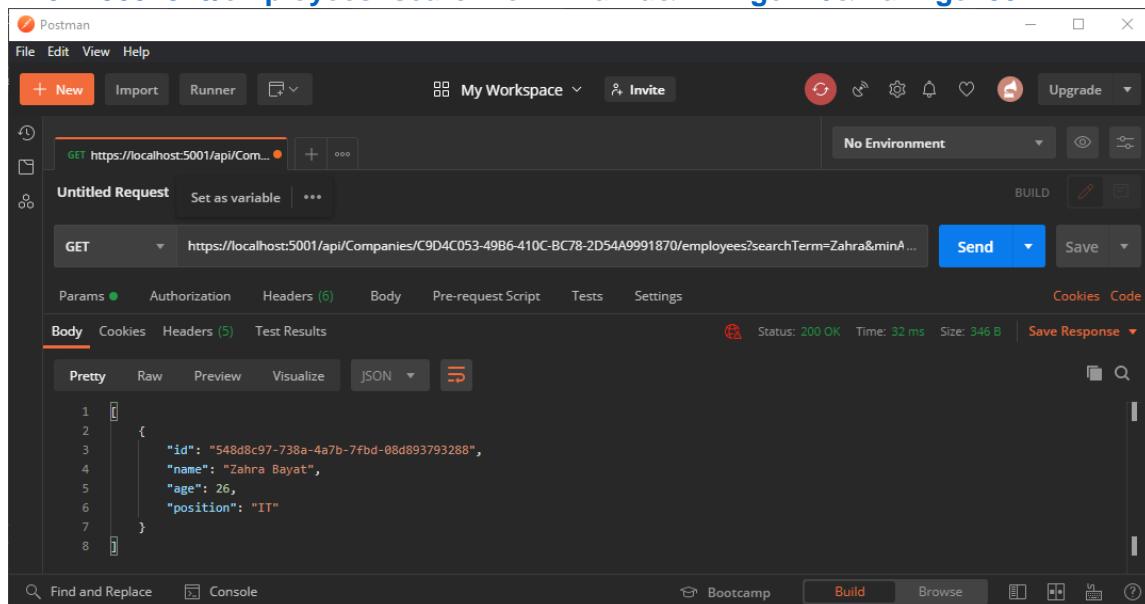
```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId, EmployeeParameters employeeParameters, bool trackChanges)
{
    var employees = await FindByCondition(e =>
        e.CompanyId.Equals(companyId),
        trackChanges)
    .FilterEmployees(employeeParameters.MinAge, employeeParameters.MaxAge)
    .Search(employeeParameters.SearchTerm)
    .OrderBy(e => e.Name)
    .ToListAsync();

    return PagedList<Employee>
    .ToPagedList(employees, employeeParameters.PageNumber,
    employeeParameters.PageSize);
}
```

تست پیادهسازی Searching

باید اولین ریکوئست با مقدار Zahra را برای جستجو ارسال کنیم.

<https://localhost:5001/api/Companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?searchTerm=Zahra&minAge=26&maxAge=35>



فصل دهم : Data Shaping و Sorting

آنچه خواهید آموخت:

➤ آشنایی و پیاده‌سازی Data Shaping و Sorting

چیست؟ Sorting

ترتیب نمایش اطلاعات را Sorting می‌گویند. مکانیسمی است که معمولاً در هر API، باید آن را پیاده‌سازی کرد. البته در این فصل درمورد الگوریتم‌های Sorting صحبت نمی‌کنیم و فقط به نحوه پیاده‌سازی می‌پردازیم.

هدف Sorting این است که ترتیب نمایش نتایج، به شکلی باشد که ما می‌خواهیم. به طور مثال :

می‌خواهیم نام کارمندان را به ترتیب با استفاده از نام و سپس سن به صورت صعودی مرتب کنیم.

برای انجام این کار، ریکوئست ما باید چیزی شبیه به این باشد.

<https://localhost:5001/api/companies/companyId/employees?orderBy=name,age desc>

API باید تمام پارامترها را در نظر بگیرد و نتایج را طبق آن مرتب کند. در ریکوئست بالا باید نتایج را براساس نام مرتب، سپس اگر کارمندانی با نام مشابه وجود داشته باشند باید نتایج را بر اساس پراپرتی سن مرتب کند.

پیاده‌سازی در Sorting

پیاده‌سازی Sorting در ASP.NET Core به دلیل انعطاف پذیری LINQ و ادغام خوب با Core، کار سختی نیست.

برای پیاده‌سازی این قابلیت، باید همانند سایر ویژگی‌هایی که تاکنون انجام دادیم، ابتدا در کلاس RequestParameters یک پراپرتی با نام OrderBy اضافه کنیم.

```
public abstract class RequestParameters
{
    const int maxPageSize = 50;
    public int PageNumber { get; set; } = 1;
    private int _pageSize = 10;
    public int PageSize
    {
        get
    }
```

```

    {
        return _pageSize;
    }
    set
    {
        _pageSize = (value > maxPageSize) ? maxPageSize : value;
    }
}

public string OrderBy { get; set; }
}

```

می خواهیم نتایج خود را بر اساس نام مرتب کنیم، حتی اگر در ریکوئست مشخص نشده باشد. پس بیایید کلاس EmployeeParameters را تغییر دهیم تا در صورتی که حتی نامی هم مشخص نشده بود، شرط مرتب سازی پیش فرض برای Employee فعال باشد.

```

public class EmployeeParameters : RequestParameters
{
    public EmployeeParameters()
    {
        OrderBy = "name";
    }

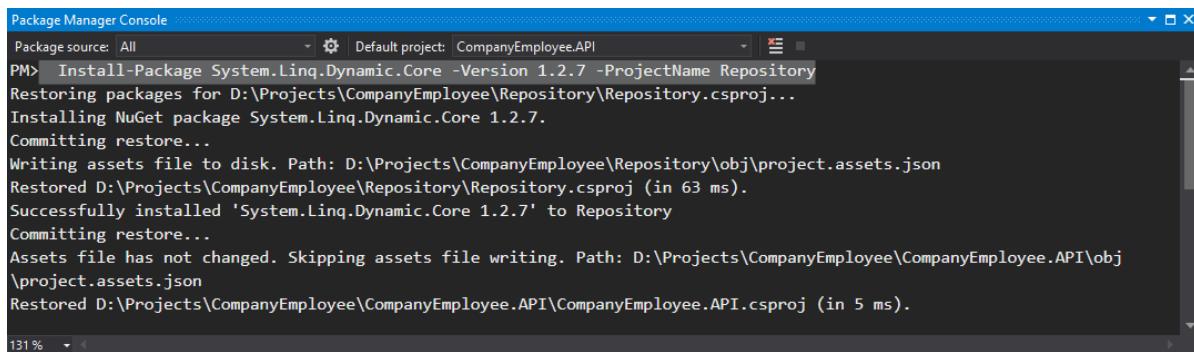
    public uint MinAge { get; set; }
    public uint MaxAge { get; set; } = int.MaxValue;
    public bool ValidAgeRange => MaxAge > MinAge;
    public string SearchTerm { get; set; }
}

```

در مرحله بعد، باید مکانیسم Sorting را پیاده کنیم.

توجه داشته باشید که برای اضافه کردن OrderBy داینامیک در کوئری، نیاز به پکیج داریم پس آن را در پروژه Repository نصب کنید.

```
Install-Package System.Linq.Dynamic.Core -Version 1.2.7 -ProjectName
Repository
```



```
Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Install-Package System.Linq.Dynamic.Core -Version 1.2.7 -ProjectName Repository
Restoring packages for D:\Projects\CompanyEmployee\Repository\Repository.csproj...
Installing NuGet package System.Linq.Dynamic.Core 1.2.7.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\Repository\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\Repository\Repository.csproj (in 63 ms).
Successfully installed 'System.Linq.Dynamic.Core 1.2.7' to Repository
Committing restore...
Assets file has not changed. Skipping assets file writing. Path: D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj (in 5 ms).

131 %
```

حالا `RepositoryEmployeeExtensions` Namespace پایین را در کلاس اضافه کنید.

خب بیاید اکستنشن متدهای `Sort` را در کلاس `RepositoryEmployeeExtensions` بنویسیم.

```
using Entities.Models;
using System;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Linq.Dynamic.Core;

namespace Repository.Extensions
{
    public static class RepositoryEmployeeExtensions
    {
        public static IQueryable<Employee> FilterEmployees(this
            IQueryable<Employee> employees, uint minAge, uint maxAge) =>
            employees.Where(e => (e.Age >= minAge && e.Age <= maxAge));

        public static IQueryable<Employee> Search(this
            IQueryable<Employee> employees, string searchTerm)
        {
            if (string.IsNullOrWhiteSpace(searchTerm))
                return employees;

            var lowerCaseTerm = searchTerm.Trim().ToLower();

            return employees.Where(e =>
                e.Name.ToLower().Contains(lowerCaseTerm));
        }

        public static IQueryable<Employee> Sort(this IQueryable<Employee>
            employees, string orderByQueryString)
```

```

{
    if (string.IsNullOrWhiteSpace(orderByQueryString))
        return employees.OrderBy(e => e.Name);

    var orderParams = orderByQueryString.Trim().Split(',');
    var propertyInfos =
typeof(Employee).GetProperties(BindingFlags.Public |
BindingFlags.Instance);

    var orderQueryBuilder = new StringBuilder();

    foreach (var param in orderParams)
    {
        if (string.IsNullOrWhiteSpace(param))
            continue;

        var propertyFromQueryName = param.Split(" ")[0];
        var objectProperty = propertyInfos.FirstOrDefault(pi =>
pi.Name.Equals(propertyFromQueryName,
StringComparison.InvariantCultureIgnoreCase));

        if (objectProperty == null)
            continue;

        var direction = param.EndsWith(" desc") ? "descending" :
"ascending";

        orderQueryBuilder.Append($"{objectProperty.Name.ToString()}")
            {direction}, ");
    }

    var orderQuery = orderQueryBuilder.ToString().TrimEnd(',', ' ');
    if (string.IsNullOrWhiteSpace(orderQuery))
        return employees.OrderBy(e => e.Name);

    return employees.OrderBy(orderQuery);
}
}

```

این متدهای خیلی کارها را انجام می‌دهد پس بگذارید قدم به قدم آن را بررسی کنیم تا ببینیم دقیقاً چه کاری انجام داده‌ایم.

بررسی کد:

- این متدهای دو آرگومان می‌گیرد، یکی برای لیست کارمندان و دیگری برای ریکوئست Ordering است.

- با چک کردن آرگومان null یا orderQueryString شروع کنیم. اگر این آرگومان null یا خالی باشد، ما فقط مجموعه را مرتب می‌کنیم و برمی‌گردانیم.

```
if (string.IsNullOrWhiteSpace(orderQueryString))
    return employees.OrderBy(e => e.Name);
```

- در غیر این صورت، orderQueryString را split کرده تا فیلدهای موردنظر را بدست آوریم.

```
var propertyFromQueryName = param.Split(" ")[0];
```

- برای اینکه لیستی از پراپرتی‌های کلاس Employee را بدست آوریم از Reflection استفاده کردیم. با این لیست می‌توان مطمئن شد که فیلد دریافت شده از Query String، حتماً در کلاس Employee وجود داشته باشد.

```
var propertyInfos = typeof(Employee).GetProperties(BindingFlags.Public |
    BindingFlags.Instance);
```

- حالا در یک حلقه Foreach تک تک پارامترها را بررسی می‌کنیم تا ببینیم محتوای آن‌ها حاوی پراپرتی هست یا خیر.

```
if (string.IsNullOrWhiteSpace(param))
    continue;

var propertyFromQueryName = param.Split(" ")[0];
var objectProperty = propertyInfos.FirstOrDefault(pi =>
    pi.Name.Equals(propertyFromQueryName,
        StringComparison.InvariantCultureIgnoreCase));
```

- اگر پراپرتی پیدا نکردیم، مرحله رفتن به پارامتر بعدی را بی‌خیال می‌شویم.

```
if (objectProperty == null)
    continue;
```

- اگر پر اپرتی را پیدا کنیم، آن را بر می گردانیم و سپس برای تصمیم گیری در مورد مرتب سازی، چک می کنیم که در انتهای رشته، پارامتر "desc" هست یا خیر.

```
var direction = param.EndsWith(" desc") ? "descending" : "ascending";
orderQueryBuilder.Append($"{objectProperty.Name.ToString()}{direction}, ");
```

- برای ساخت کوئری از StringBuilder استفاده می کنیم.

```
orderQueryBuilder.Append($"{objectProperty.Name.ToString()}{direction}, ");
```

- حالا که همه پر اپرتی ها را با Foreach بررسی کردیم، باید کاماهای اضافی را حذف و ببینیم آیا در ریکوئست واقعاً چیزی هست یا خیر.

```
var orderQuery = orderQueryBuilder.ToString().TrimEnd(' ', ' ', ' ');
```

```
if (string.IsNullOrWhiteSpace(orderQuery))
    return employees.OrderBy(e => e.Name);
```

- در پایان می توانیم کوئری خود را مرتب کنیم.

```
return employees.OrderBy(orderQuery);
```

- در این مرحله متغیر orderQuery باید شامل عبارت "Name ascending, DateOfBirth" باشد. این عبارت نتایج ما را ابتدا بر اساس Name به صورت صعودی و سپس توسط DateOfBirth به صورت نزولی مرتب می کند.

نکته!!

برای زدن کوئری LINQ این عبارت، می توانید به صورت پایین عمل کنید.

```
employees.OrderBy(e => e.Name).ThenByDescending(o => o.Age);
```

این کد ترفندهایی برای مرتب سازی کوئری است و برای زمانی که نمی دانید چطور مرتب سازی کنید پر کاربرد است.

پس از نوشتن این متدها در GetEmployeesAsync را در EmployeeRepository اصلاح کنیم.

```
public async Task<PagedList<Employee>> GetEmployeesAsync(Guid companyId, EmployeeParameters employeeParameters, bool trackChanges)
{
```

```

var employees = await FindByCondition(e =>
    e.CompanyId.Equals(companyId),
    trackChanges)
    .FilterEmployees(employeeParameters.MinAge,
    employeeParameters.MaxAge)
    .Search(employeeParameters.SearchTerm)
    .Sort(employeeParameters.OrderBy)
    .ToListAsync();

return PagedList<Employee>
    .ToPagedList(employees, employeeParameters.PageNumber,
    employeeParameters.PageSize);
}

```

حالا می‌توانیم این ویژگی را تست کنیم.

تست Sorting

ابتدا، کوئری که بالاتر گفتیم را تست می‌کنیم.

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870/employees?orderBy=name,age desc>

```

[{"id": "86dba8c0-d178-41e7-938c-ed49778fb52a", "name": "Ali Bayat", "age": 30, "position": "Backend developer"}, {"id": "021ca3c1-0deb-4af8-ae94-2159a8479811", "name": "Sara Bayat", "age": 35, "position": "Frontend developer"}, {"id": "548d8c97-738a-4a7b-7fbd-08d893793288", "name": "Zahra Bayat", "age": 26, "position": "IT"}]

```

همانطور که می‌بینید لیست به صورت صعودی مرتب شد.

چیست؟ Data Shaping

Data Shaping یک روش عالی برای کاهش ترافیک ارسال شده از API به کلاینت است. Data Shaping به کلاینت این امکان را می‌دهد، تا با انتخاب فیلدها از طریق Query String، داده‌ها را انتخاب و شکل دهد.

ما می‌توانیم برای کلاینت امکانی را فراهم کنیم تا فیلدہای مورد نیازش را به درستی انتخاب و با این کار استرس را در API کاهش دهیم. البته توجه داشته باشید که Data Shaping چیزی نیست که هر API به آن نیاز داشته باشد؛ چون با Reflection همراه است و همانطور که می‌دانید Reflection مشکلاتی دارد و Performance ما را پایین می‌آورد. بنابراین باید با دقت تصمیم بگیرید که آیا نیاز به پیاده‌سازی دارید یا خیر.

در پایان باید مثل همیشه Data Shaping با مفاهیمی Filtering، Paging و Searching را پیاده‌سازی کرد. Sorting به خوبی کار کند.

چطور Data Shaping را پیاده‌سازی کنیم؟

چون می‌خواهیم پرپرتری جدیدی را به Query String اضافه کنیم و این پرپرتری باید برای هر وجود داشته باشد، پس ابتدا باید کلاس RequestParameters را تغییر دهیم.

```
public abstract class RequestParameters
{
    const int maxPageSize = 50;
    public int PageNumber { get; set; } = 1;
    private int _pageSize = 10;
    public int PageSize
    {
        get
        {
            return _pageSize;
        }
        set
        {
```

```

        _pageSize = (value > maxPageSize) ? maxPageSize : value;
    }

}

public string OrderBy { get; set; }
public string Fields { get; set; }
}

```

ما پر اپرتی Fields را اضافه کردیم و حالا می توانیم از این پر اپرتی به عنوان یک پارامتر Query استفاده کنیم.

باید با ایجاد یک اینترفیس جدید با نام Contracts/ IServices، در مسیر IDataShaper این کار را ادامه دهیم.

```

using System.Collections.Generic;
using System.Dynamic;

namespace Contracts.IServices
{
    public interface IDataShaper<T>
    {
        IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities,
                                                string
                                                fieldsString);

        ExpandoObject ShapeData(T entity, string fieldsString);
    }
}

```

بررسی کد :

- این اینترفیس دو متده دارد؛ یکی برای یک Entity و دیگری برای مجموعه Entity ها که باید پیاده سازی شود. هر دوی این متدها ShapeData نامگذاری شده اند اما های مختلفی دارند.
- توجه کنید که از کلاس ExpandoObject به عنوان Return Type استفاده کردیم، تا داده هایمان را به صورتی که می خواهیم شکل دهیم.

برای پیاده‌سازی این اینترفیس، باید در پروژه Repository فolder جدیدی با نام DataShaping ایجاد و در آن کلاسی با نام DataShaper اضافه نمایید.

```
using Contracts.IServices;
using System;
using System.Collections.Generic;
using System.Dynamic;
using System.Linq;
using System.Reflection;

namespace Repository.DataShaping
{
    public class DataShaper<T> : IDataShaper<T> where T : class
    {
        public PropertyInfo[] Properties { get; set; }
        public DataShaper()
        {
            Properties = typeof(T).GetProperties(BindingFlags.Public |
            BindingFlags.Instance);
        }

        public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T>
entities, string fieldsString)
        {
            var requiredProperties = GetRequiredProperties(fieldsString);

            return FetchData(entities, requiredProperties);
        }

        public ExpandoObject ShapeData(T entity, string fieldsString)
        {
            var requiredProperties = GetRequiredProperties(fieldsString);

            return FetchDataForEntity(entity, requiredProperties);
        }

        private IEnumerable<PropertyInfo> GetRequiredProperties(string
fieldsString)
        {
            var requiredProperties = new List<PropertyInfo>();
            if (!string.IsNullOrWhiteSpace(fieldsString))
            {
                var fields = fieldsString.Split(',', StringSplitOptions.RemoveEmptyEntries);
                foreach (var field in fields)
                {
                    var property = typeof(T).GetProperty(field);
                    if (property != null)
                    {
                        requiredProperties.Add(property);
                    }
                }
            }
            return requiredProperties;
        }
    }
}
```

```

        foreach (var field in fields)
        {
            var property = Properties
                .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
                    StringComparison.InvariantCultureIgnoreCase));

            if (property == null)
                continue;

            requiredProperties.Add(property);
        }
    }
    else
    {
        requiredProperties = Properties.ToList();
    }

    return requiredProperties;
}

private IEnumerable<ExpandoObject> FetchData(IEnumerable<T>
entities,
IEnumerable< PropertyInfo > requiredProperties)
{
    var shapedData = new List<ExpandoObject>();

    foreach (var entity in entities)
    {
        var shapedObject = FetchDataForEntity(entity,
            requiredProperties);

        shapedData.Add(shapedObject);
    }

    return shapedData;
}

private ExpandoObject FetchDataForEntity(T entity,
IEnumerable< PropertyInfo > requiredProperties)
{
    var shapedObject = new ExpandoObject();

    foreach (var property in requiredProperties)
    {
        var objectPropertyValue = property.GetValue(entity);

```

```

        shapedObject.TryAdd(property.Name, objectPropertyValue);
    }

    return shapedObject;
}
}
}

```

کد زیادی در اینجا وجود دارد، پس باید آن را بشکنیم و با هم بررسی کنیم.

بررسی کد:

- در این کلاس یک پرایمیتی public به نام Properties وجود دارد. نوع این پرایمیتی آرایه ای از PropertyInfo's است و نوع ورودی از هر نوعی که باشد درون آن ریخته می شود. بنابراین در Constructor باید همه پرایمیتی های کلاس ورودی را بدست آوریم. (در پروژه ما این نوع ورودی Employee یا Company است).

```

public PropertyInfo[] Properties { get; set; }
public DataShaper()
{
    Properties = typeof(T).GetProperties(BindingFlags.Public |
    BindingFlags.Instance);
}

```

- در قسمت بعد دو متده با نام ShapeData داریم. وظیفه هی هر دو متده این است که string ورودی را با استفاده از متده GetRequiredProperties تجزیه و فیلد هایی که در این ورودی هست را واکشی کنند.

```

public IEnumerable<ExpandoObject> ShapeData(IEnumerable<T> entities,
string fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchData(entities, requiredProperties);
}

public ExpandoObject ShapeData(T entity, string fieldsString)
{
    var requiredProperties = GetRequiredProperties(fieldsString);

    return FetchDataForEntity(entity, requiredProperties);
}

```

- متدهای GetRequiredProperties ورودی را تجزیه و فقط پرایپریتی‌های مورد نیاز را به کنترلر برمی‌گرداند.

```
private IEnumerable< PropertyInfo > GetRequiredProperties(string fieldsString)
{
    var requiredProperties = new List< PropertyInfo >();
    if (!string.IsNullOrWhiteSpace(fieldsString))
    {
        var fields = fieldsString.Split(',',
            StringSplitOptions.RemoveEmptyEntries);

        foreach (var field in fields)
        {
            var property = Properties
                .FirstOrDefault(pi => pi.Name.Equals(field.Trim(),
                    StringComparison.InvariantCultureIgnoreCase));

            if (property == null)
                continue;

            requiredProperties.Add(property);
        }
    }
    else
    {
        requiredProperties = Properties.ToList();
    }

    return requiredProperties;
}
```

- همانطور که می‌بینید چیز خاصی در این کد وجود ندارد. اگر fieldsString خالی نباشد، آن را Split و سپس چک می‌کنیم که آیا فیلدها با پرایپریتی‌های Entity مطابقت دارند یا خیر.
- اگر مطابق باشند، آنها را به لیست پرایپریتی‌های مورد نیاز اضافه می‌کنیم.
- اما اگر fieldsString خالی باشد، همه‌ی پرایپریتی‌ها موردنیاز است.
- متدهای private FetchDataForEntity و FetchData هستند که مقادیر را از این پرایپریتی‌ها، جدا و برای ما آماده می‌کنند.

- متد FetchDataForEntity این عملیات را برای یک Entity انجام می‌دهد.

```
private ExpandoObject FetchDataForEntity(T entity,
IEnumerable< PropertyInfo > requiredProperties)
{
    var shapedObject = new ExpandoObject();

    foreach (var property in requiredProperties)
    {
        var objectPropertyValue = property.GetValue(entity);

        shapedObject.TryAdd(property.Name, objectPropertyValue);
    }

    return shapedObject;
}
```

- همانطور که می‌بینید با استفاده از Foreach یک حلقه requiredProperties را اجرا و سپس با استفاده از Reflection مقادیر را استخراج کردیم.
- در پایان، این مقادیر را به آبجکت ExpandoObject اضافه می‌کنیم.
- کلاس ExpandoObject یک IDictionary<string,object> را پیاده‌سازی می‌کند.
- بنابراین می‌توانیم برای اضافه کردن پرآپرتی‌ها، از متد TryAdd استفاده کنیم و به این صورت به طور خودکار پرآپرتی‌های موردنیاز، به آبجکت اضافه شوند.
- متد FetchDataForEntity هم، شبیه FetchData است با این تفاوت که برای مجموعه‌ای از Entity‌ها، عمل واکشی اطلاعات پرآپرتی را انجام می‌دهد.

```
private IEnumerable<ExpandoObject> FetchData(IEnumerable<T> entities,
IEnumerable< PropertyInfo > requiredProperties)
{
    var shapedData = new List<ExpandoObject>();

    foreach (var entity in entities)
    {
        var shapedObject = FetchDataForEntity(entity, requiredProperties);

        shapedData.Add(shapedObject);
    }

    return shapedData;
}
```

برای ادامه، اجازه دهید کلاس ConfigureServices را در متد DataShaper رجیستر کنیم.

```
services.AddScoped <IDataShaper<EmployeeDto>, DataShaper<EmployeeDto>>();
```

فراموش نکنید که حتما Namespace‌هایی پایین را به کلاس Startup اضافه نمایید.

```
using Repository.DataShaping;  
using Entities.DataTransferObjects;
```

در پایان باید EmployeesController Constructor را اصلاح کنیم.

```
private readonly IDataShaper<EmployeeDto> _dataShaper;  
  
public EmployeesController(IRepositoryManager repository, ILoggerManager  
    logger, IMapper mapper, IDataShaper<EmployeeDto> dataShaper)  
{  
    _repository = repository;  
    _logger = logger;  
    _mapper = mapper;  
    _dataShaper = dataShaper;  
}
```

حالا خروجی اکشن GetEmployeesForCompanyAsync را همانند کد پایین تغییر دهید.

```
[HttpGet(Name = "GetEmployeeForCompany")]  
public async Task<IActionResult> GetEmployeesForCompanyAsync(Guid  
    companyId, [FromQuery] EmployeeParameters employeeParameters)  
{  
    var company = await _repository.Company.GetCompanyAsync(companyId,  
        trackChanges: false);  
  
    if (!employeeParameters.ValidAgeRange)  
        return BadRequest("Max age can't be less than min age.");  
  
    if (company == null)  
    {  
        _logger.LogError($"Company with id: {companyId} doesn't exist  
            in the database.");  
  
        return NotFound();  
    }  
  
    var employeesFromDb = await  
        _repository.Employee.GetEmployeesAsync(companyId,  
        employeeParameters, trackChanges: false);
```

```

        Response.Headers.Add("X-Pagination",
JsonConvert.SerializeObject(employeesFromDb.MetaData));

var employeesDto =
_mapper.Map<IEnumerable<EmployeeDto>>(employeesFromDb);

return Ok(_dataShaper.ShapeData(employeesDto,
employeeParameters.Fields));
}

}

```

اکنون میتوانیم این متدها را تست کنیم.

<https://localhost:5001/api/Companies/c9d4c053-49b6-410c-bc78-2d54a9991870/employees?pageNumber=1&pageSize=3&minAge=26&maxAge=35&fields=name,age>

KEY	VALUE	DESCRIPTION
pageNumber	1	
pageSize	3	
minAge	26	
maxAge	35	
fields	name,age	

```

1 [
2   {
3     "Name": "Ali Bayat",
4     "Age": 30
5   },
6   {
7     "Name": "Sara Bayat",
8     "Age": 35
9   },
10  {
11    "Name": "Zahra Bayat",
12    "Age": 26
13  }
14 ]

```

همه چیز عالی کار میکند.

فصل یازدهم : API Versioning

آنچه خواهید آموخت:

- API Versioning چیست؟
- پیاده سازی و تست API Versioning
- منسوخ کردن Version

API Versioning

با رشد اپلیکیشن، نیازمندی‌های پروژه هم تغییر می‌کند و همین باعث می‌شود API‌ها نیز تغییر کنند.

تغییرات API می‌تواند شامل موارد پایین باشد.

- تغییر نام فیلدها، پرایپرتبی‌ها یا Resource URI‌ها باشد.
- تغییرات در ساختار باشد.
- تغییرات HTTP Verb یا Response Code باشد.
- طراحی مجدد API باشد.

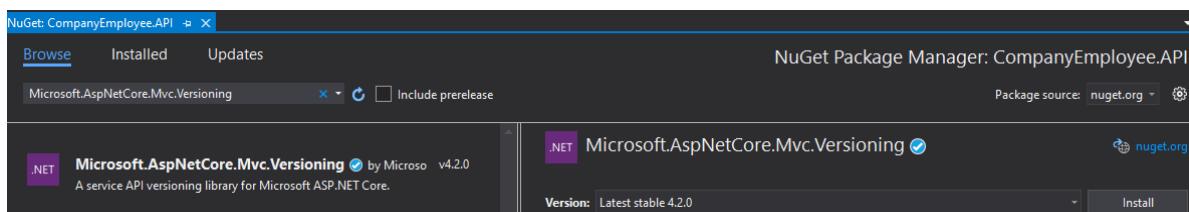
وقتی API تغییر کند، کد سمت کلاینت به مشکل برخورد و در نتیجه API با شکست روبرو می‌شود.

بهترین راه برای جلوگیری از این شکست، استفاده از API Versioning است و روش‌های مختلفی برای پیاده‌سازی آن وجود دارد.

هیچ راهنمایی وجود ندارد که کدام روش بهتر از باقی روش‌های است، بنابراین ما روش‌های مختلف را معرفی می‌کنیم و شما با توجه به نیازتان می‌توانید هر کدام را انتخاب نمایید.

روش اول :

نصب پکیج API در پروژه Microsoft.AspNetCore.Mvc.Versioning



پس از نصب پکیج بالا، باید سرویس Versioning را رجیستر کنیم پس به یک متدهای در کلاس ServiceExtensions نیاز داریم.

ابتدا Namespace را در کلاس ServiceExtensions اضافه کنید.

```
using Microsoft.AspNetCore.Mvc;
```

و سپس متد پایین را بنویسید.

```
public static void ConfigureVersioning(this IServiceCollection services)
{
    services.AddApiVersioning(opt =>
    {
        opt.ReportApiVersions = true;
        opt.AssumeDefaultVersionWhenUnspecified = true;
        opt.DefaultApiVersion = new ApiVersion(1, 0);
    });
}
```

این اکستنشن برای پیکربندی Versioning، از سه پراپرتی استفاده می‌کند.

- ورژن API را به هدر Response اضافه می‌کند.
- اگر کلاینت ورژن را ارسال نکند، این پراپرتی ورژن پیشفرض API را مشخص می‌کند.
- این پراپرتی تعداد ورژن پیشفرض را تنظیم می‌کند.

خب حالا با استفاده از متد API Versioning، باید AddApiVersioning را در متد ConfigureServices اضافه کنید.

```
services.ConfigureVersioning();
```

خب API نصب و پیکربندی شد، حالا می‌توانیم ادامه دهیم.

تست Versioning

برای تست این قابلیت باید کنترلر دیگری با نام CompaniesV2Controller ایجاد کنیم. این کنترلر فقط یک اکشن متد Get دارد.

```
using Contracts.IServices;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Controllers
{
    [ApiVersion("2.0")]
    [Route("api/companies")]
}
```

```

[ApiController]
public class CompaniesV2Controller : ControllerBase
{
    private readonly IRepositoryManager _repository;

    public CompaniesV2Controller(IRepositoryManager repository)
    {
        _repository = repository;
    }

    [HttpGet]
    public async Task<IActionResult> GetCompaniesAsync()
    {
        var companies = await
            _repository.Company.GetAllCompaniesAsync(trackChanges:
            false);

        return Ok(companies);
    }
}

```

بررسی کد :

- اتریبیوت [ApiVersion("2.0")] مشخص می‌کند که این کنترلر ورژن 2.0 است.
- در اکشن متدهای Get، به جای برگرداندن DTO به کلاینت، یک Entity برگرداندیم.

خب حالا قبل از اجرا باید Version را به کنترلر اصلی اضافه نمایید.

```

[ApiVersion("1.0")]
[Route("api/companies")]
[ApiController]
public class CompaniesController : ControllerBase

```

همانطور که در اکسشن متدهای AddApiVersioning، ما Versioning را بر روی 1.0 تنظیم کردیم.

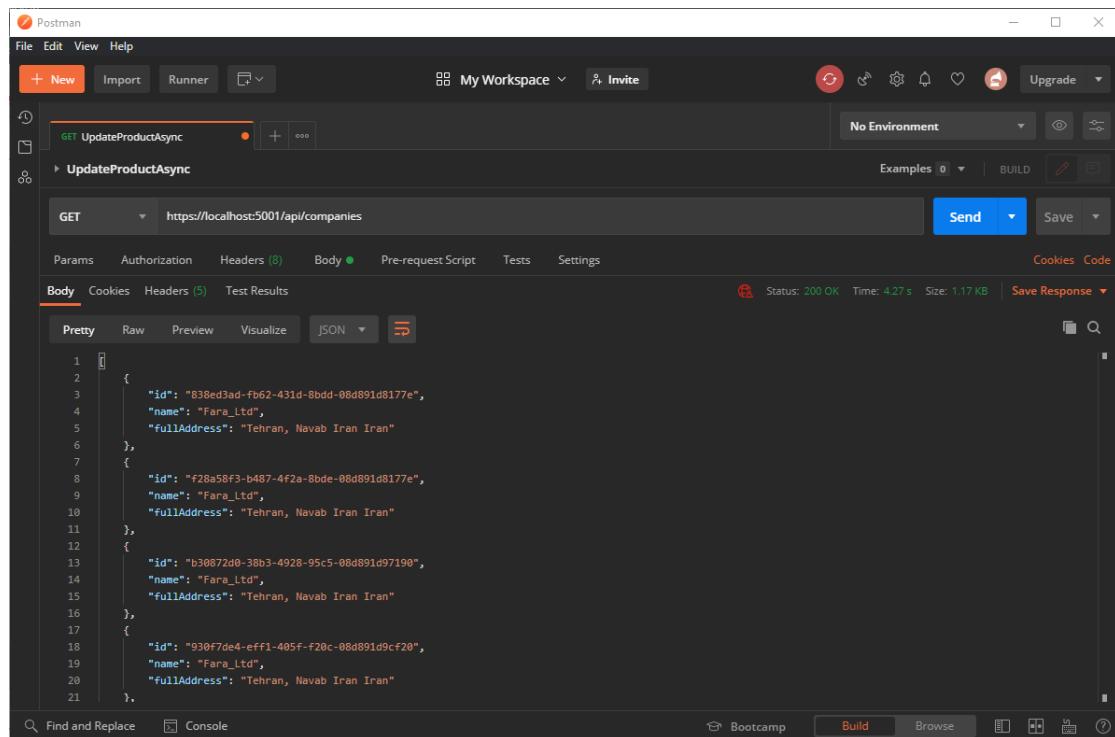
```
opt.DefaultApiVersion = new ApiVersion(1, 0);
```

همچنین با کد پایین مشخص می‌کنیم که اگر Versioning مشخص نشده بود مقدار پیش فرض را در نظر بگیر.

```
opt.AssumeDefaultVersionWhenUnspecified = true;
```

بنابراین اگر کلاینت ورژن را مشخص نکند، API از نسخه 1.0 استفاده خواهد کرد.

<https://localhost:5001/api/companies>

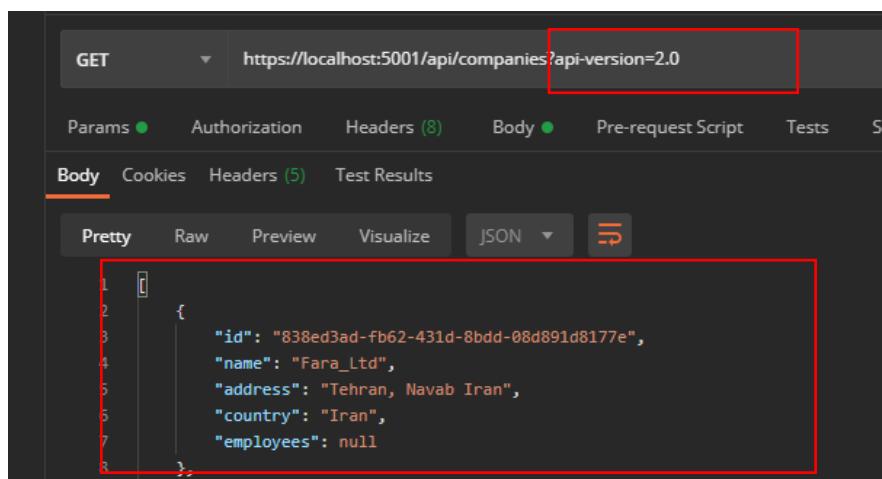


```
1 [
2   {
3     "id": "838ed3ad-fb62-431d-8bdd-08d891d8177e",
4     "name": "Fara_Ltd",
5     "fullAddress": "Tehran, Navab Iran Iran"
6   },
7   {
8     "id": "f28a58f3-b487-4f2a-8bde-08d891d8177e",
9     "name": "Fara_Ltd",
10    "fullAddress": "Tehran, Navab Iran Iran"
11  },
12  {
13    "id": "b30872d0-38b3-4928-95c5-08d891d97190",
14    "name": "Fara_Ltd",
15    "fullAddress": "Tehran, Navab Iran Iran"
16  },
17  {
18    "id": "930f7de4-eff1-405f-f20c-08d891d9cf20",
19    "name": "Fara_Ltd",
20    "fullAddress": "Tehran, Navab Iran Iran"
21 },
```

پراپرتی fullAddress در این تصویر به این معنی است که کنترلر اصلی ما فراخوانی شده است، حتی اگر در خواستی ورژن API را مشخص نکرده باشیم.

خب حالا می‌توانیم با استفاده از یک Query String در URI، نسخه‌ی API را در ریکوئست ارائه کنیم.

<https://localhost:5001/api/companies?api-version=2.0>



```
1 [
2   {
3     "id": "838ed3ad-fb62-431d-8bdd-08d891d8177e",
4     "name": "Fara_Ltd",
5     "address": "Tehran, Navab Iran",
6     "country": "Iran",
7     "employees": null
8   }
]
```

همانطور که می‌بینید، به جای خروجی Company Entity، یک لیستی از نوع Company Dto داریم. بنابراین مطمئن می‌شویم که حتماً ورژن 2.0 صدا زده شده است.

برای مطمئن شدن می‌توانیم هدرهای Response را هم بررسی کنیم.

Body	Cookies	Headers (5)	Test Results
			Status: 200 OK
KEY			VALUE
Date			Tue, 22 Dec 2020 06:22:31 GMT
Content-Type			application/json; charset=utf-8
Server			Kestrel
Content-Length			2039
api-supported-versions			2.0

استفاده از URL Versioning

در تست بالا برای مشخص کردن ورژن، از Query String استفاده کردیم. اما همین کار را می‌توان با استفاده از URL هم مشخص کرد.

برای این کار باید اتریبوت Route بالای کنترلر را به صورت پایین تغییر دهید.

```
[ApiVersion("2.0")]
[Route("api/{v:apiversion}/companies")]
[ApiController]
public class CompaniesV2Controller : ControllerBase
```

حالا می‌توانیم آن را تست کنیم.

<https://localhost:5001/api/2.0/companies>

```
1 [
2   {
3     "id": "838ed3ad-fb62-431d-8bdd-08d891d8177e",
4     "name": "Fara_Ltd",
5     "address": "Tehran, Navab Iran",
6     "country": "Iran",
7     "employees": null
8   },
9 ]
```

به این نکته توجه داشته باشید که برای CompaniesV2Controller نمی‌توان از الگوی Query String استفاده کرد اما برای ورژن 1.0 باید از Query String استفاده کنیم.

HTTP Header Versioning

اگر نخواهیم URI را تغییر دهیم، می‌توان ورژن را در هدر HTTP ارسال کرد. برای فعال کردن این گزینه باید متدهای ConfigureVersioning را تغییر دهیم.

```
public static void ConfigureVersioning(this IServiceCollection services)
{
    services.AddApiVersioning(opt =>
    {
        opt.ReportApiVersions = true;
        opt.AssumeDefaultVersionWhenUnspecified = true;
        opt.DefaultApiVersion = new ApiVersion(1, 0);
        opt.ApiVersionReader = new HeaderApiVersionReader("api-
version");
    });
}
```

فراموش نکنید که حتما Namespace پایین را هم در این کلاس اضافه کنید.

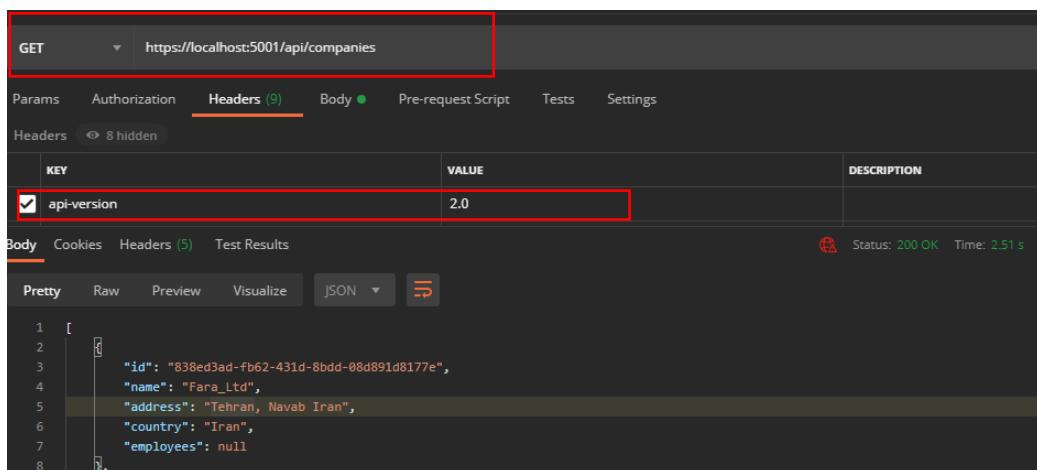
```
using Microsoft.AspNetCore.Mvc.Versioning;
```

خب حالا باید اتریبوت Route را در CompaniesV2Controller را تغییر دهید.

```
[ApiVersion("2.0")]
[Route("api/companies")]
[ApiController]
public class CompaniesV2Controller : ControllerBase
```

بیایید این تغییرات را تست کنیم.

<https://localhost:5001/api/companies>



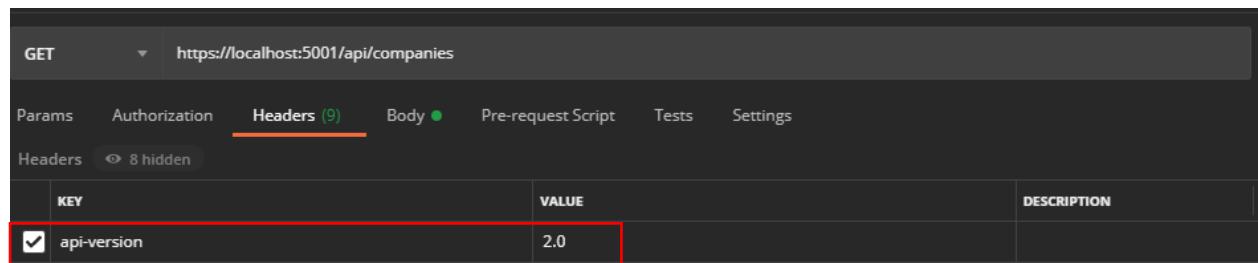
منسخ کردن Version

اگر بخواهیم ورژن 2.0 منسخ شود اما نخواهیم آن را به طور کامل حذف کنیم، باید از پراپرتی استفاده کنیم Deprecated.

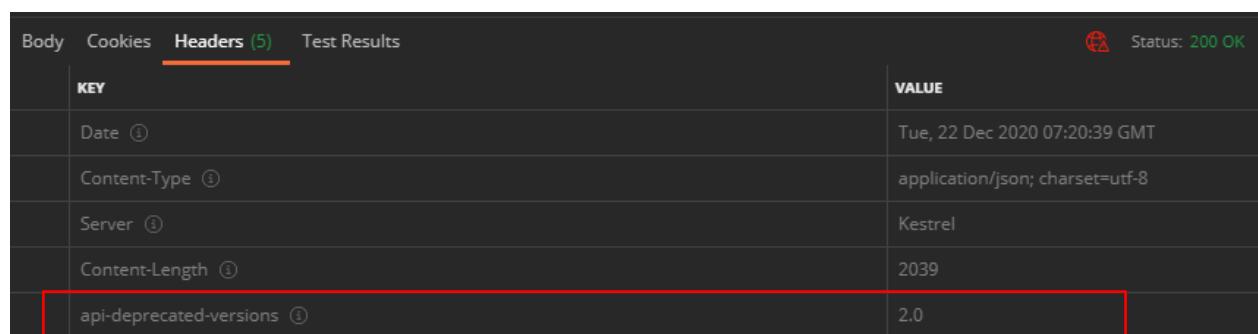
```
[ApiVersion("2.0", Deprecated = true)]  
[Route("api/companies")]  
[ApiController]  
public class CompaniesV2Controller : ControllerBase
```

خب اگر ریکوئست پایین را اجرا کنیم نتیجه Header را اینطور می‌بینیم.

<https://localhost:5001/api/companies>



KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> api-version	2.0	



KEY	VALUE
Date	Tue, 22 Dec 2020 07:20:39 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	2039
api-deprecated-versions	2.0

نکته!!

اگر ورژن‌های زیادی از یک کنترل داشته باشیم و بخواهیم تعدادی از این کنترل‌ها را منسخ کنیم، می‌توان از پیکربندی اختصاصی استفاده کرد.

```
public static void ConfigureVersioning(this IServiceCollection services)  
{  
    services.AddApiVersioning(opt =>  
    {  
        opt.ReportApiVersions = true;  
        opt.AssumeDefaultVersionWhenUnspecified = true;
```

```
    opt.DefaultApiVersion = new ApiVersion(1, 0);

    opt.Conventions.Controller<CompaniesController>().HasApiVe
    rsion(new ApiVersion(1, 0));

    opt.Conventions.Controller<CompaniesV2Controller>().HasDepr
    ectedApiVersion(new ApiVersion(2, 0));
}

}
```

فراموش نکنید که Namespace پایین را اضافه کنید.

```
using CompanyEmployee.API.Controllers;
```

حالا باید اتریبوت [ApiVersion] را از کنترلرها حذف کنیم.

فصل دوازدهم : Cache و Rate Limiting

آنچه خواهید آموخت:

- **Caching** چیست؟
- معرفی انواع **Cache**
- افزودن هدرهای **Cache** چیست؟
- **Expiration Model** چیست؟
- **Validation Model** چیست؟
- پیکربندی هدرهای **Expiration** و **Validation** چیست؟
- **Rate Limiting** چیست؟
- **Rate-Limit** پیادهسازی

Caching چیست؟

به ذخیره داده‌ها در قسمتی از حافظه به نام Cache را، Caching می‌گویند. با سرعت دسترسی به داده‌ها زیادتر از حالت عادی است.

Cache یک کامپوننت جداگانه است که ریکوئست‌های کلاینت و ریسپانس API را می‌پذیرد و در صورتی که قابل کش شدن باشند، آن‌ها را ذخیره می‌کند.

Caching می‌تواند کیفیت و پرفورمنس اپلیکیشن را بالاتر ببرد و هدف اصلی آن این است که در بسیاری از موارد، نیاز به ارسال ریکوئست به سمت API و یا ارسال Response کامل نداشته باشیم.

برای کاهش تعداد ریکوئست‌های ارسالی، Cache از مکانیزم انقضا استفاده می‌کند. این کار باعث می‌شود رفت و برگشت درون شبکه‌ای کاهش یابد.

علاوه بر این، Cache از مکانیزم اعتبارسنجی استفاده می‌کند تا نیاز به ارسال Response کامل را از بین ببرد.

پس از ذخیره ریسپانس، اگر کلاینت دوباره همان ریسپانس را درخواست کند، باید ریسپانس از حافظه Cache ارائه شود.

انواع Cache

سه نوع Cache وجود دارد :

Client Cache : Client Cache • بر روی Client (مرورگر) زندگی می‌کند. پس چون

مربوط به کلاینت است، باید یک Private cache باشد. بنابراین هر کلاینت که API ما را مصرف می‌کند یک Private cache دارد.

Gateway Cache : Gateway Cache • در Server زندگی می‌کند و یک Cache

است. این Cache به اشتراک گذاشته می‌شود چون Resource‌هایی که در آن ذخیره می‌شوند باید بر روی کلاینت‌های مختلف به اشتراک گذاشته شوند.

Proxy Cache : Proxy Cache • Shared Cache یک Shared Cache هم است که نه در سرور و در نه

در کلاینت زندگی می‌کند؛ بلکه محل زندگی او در شبکه است.

تفاوت بین Shared Cache و Private cache

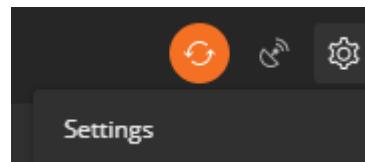
- در Private cache اگر پنج کلاینت برای اولین بار ریسپانس مشابه درخواست کنند، در این صورت هر ریسپانس از سمت API ارائه می‌شود (و نه از Cache) اما اگر آنها دوباره همان ریسپانس را بخواهند، این بار باید ریسپانس از Cache باشد (البته اگر انقضای آن تمام نشده باشد).
- در مورد Shared Cache اینگونه نیست. در این نوع Cache، ابتدا ریسپانس کلاینت اول ذخیره می‌شود و سپس چهار کلاینت دیگر، در صورت درخواست مشابه باید ریسپانس Cache شده را دریافت کنند.

افزودن هدرهای Cache

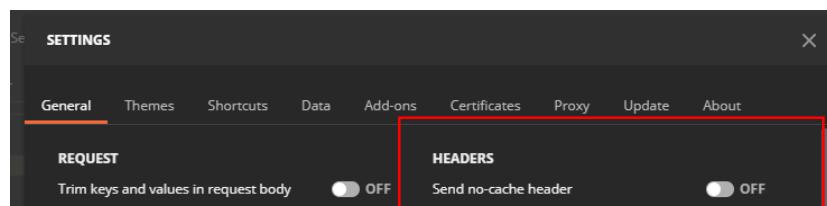
برای Cache کردن برخی Resource‌ها، باید بدانیم که آیا Cacheable هست یا خیر. Response Header در انجام این کار به ما کمک می‌کند. به طور مثال:

Cache-Control: max-age = 180 موردنی است که بیشتر اوقات استفاده می‌شود و بدین معنی است که Response باید به مدت ۱۸۰ ثانیه Cache شود.

قبل از افزودن هدرهای Cache، باید Postman را باز کنیم و تنظیمات پشتیبانی از Caching را تغییر دهیم.



در سربرگ General در بخش Headers، باید سربرگ Send no-cache را OFF کنیم.



بیایید اکشن متدهای GetCompanyAsync را به مدت ۶۰ ثانیه Cache کنیم.

```
[HttpGet("{id}", Name = "CompanyById")]
[ResponseCache(Duration = 60)]
```

```

public async Task<IActionResult> GetCompanyAsync(Guid id)
{
    var company = await _repository.Company.GetCompanyAsync(id,
trackChanges: false);

    if (company == null)
    {
        _logger.LogError($"Company with id: {id} doesn't exist in the
database.");
    }

    return NotFound();
}
else
{
    var companyDto = _mapper.Map<CompanyDto>(company);

    return Ok(companyDto);
}
}

```

خب حالا بیایید نتیجه را تست کنیم.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce2>

KEY	VALUE
Date	Wed, 23 Dec 2020 12:47:23 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	105
Cache-Control	public,max-age=60
api-supported-versions	1.0

می‌بینید که هدر Cache-Control با duration ۶۰ ثانیه ایجاد شده است. اما همانطور که گفتیم، این فقط یک هدر است. برای ذخیره‌سازی ریسپانس نیاز به cache-store نوع داریم.

cache-store اضافه کردن

برای افزودن cache-store، اولین کاری که باید انجام دهیم، اضافه کردن یک اکستنشن متده است در کلاس ServiceExtensions.

```
public static void ConfigureResponseCaching(this IServiceCollection services) => services.AddResponseCaching();
```

ما ریسپانس Caching را در IoC container می‌registر کنیم و حالا باید این اکستنشن متده در ConfigureServices صدا بزنیم.

```
services.ConfigureResponseCaching();
```

علاوه بر این، باید Caching Middleware را بالای UseRouting به اپلیکیشن اضافه کنیم.

```
app.UseResponseCaching();
app.UseRouting();
```

حال برنامه خود را Start کرده و ریکوئست پایین را ارسال کنید. این ریکوئست هدر Cache-Control را ایجاد می‌کند.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce2>

بار اول:

The screenshot shows the Postman application interface. A GET request is made to `https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870`. The Headers tab is selected, displaying the following key-value pairs:

KEY	VALUE
Date	Wed, 23 Dec 2020 13:16:52 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	105
Cache-Control	public,max-age=60
api-supported-versions	1.0

بار دوم:

قبل از اینکه ۶۰ ثانیه بگذرد، دوباره همان ریکوئست را ارسال کنید و سپس هدرها را بررسی نمایید.

The screenshot shows the Postman interface with a successful response from a GET request. The 'Headers' tab is selected, displaying the following data:

KEY	VALUE
Date	Wed, 23 Dec 2020 13:16:52 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	105
Cache-Control	public,max-age=60
Age	10

همانطور که می‌بینید در ریکوئست دوم هدر Age اضافه شده است. این هدر تعداد ثانیه‌های ذخیره شدن شی در cache را نشان می‌دهد و به این معنی است که ما ریسپانس دوم خود را، از cache-store گرفتیم.

اگر در عرض ۶۰ ثانیه، چندین ریکوئست ارسال کنیم، پراپرتی Age افزایش می‌یابد؛ سپس بعد از پایان دوره انقضا، ریسپانس از API ارسال و دوباره Cache خواهد شد بنابراین هدر Age ایجاد نمی‌شود.

علاوه بر این، می‌توان برای اعمال قوانین یکسان در Resource‌های مختلف، از Cache Profiles استفاده کرد.

در تصویر بالا پراپرتی‌های زیادی هست که اگر همه این پراپرتی‌ها را در بالای اکشن متدهای کنترلر قرار دهید، منجر به ناخوانایی کد خواهد شد. بنابراین برای واکشی این پیکربندی‌ها، می‌توانیم از CacheProfiles استفاده کنیم.

برای انجام این پیکربندی باید AddControllers را تغییر دهید.

```
services.AddControllers(config => {
```

```

config.RespectBrowserAcceptHeader = true;
config.ReturnHttpNotAcceptable = true;
config.CacheProfiles.Add("120SecondsDuration", new CacheProfile
{
    Duration = 120
});
}.AddNewtonsoftJson()
.AddXmlDataContractSerializerFormatters()
.AddCustomCSVFormatter();

```

ما فقط مدت زمان را تنظیم کردیم، اما شما در اینجا می‌توانید، پرایپری‌های دیگر را نیز اضافه کنید.

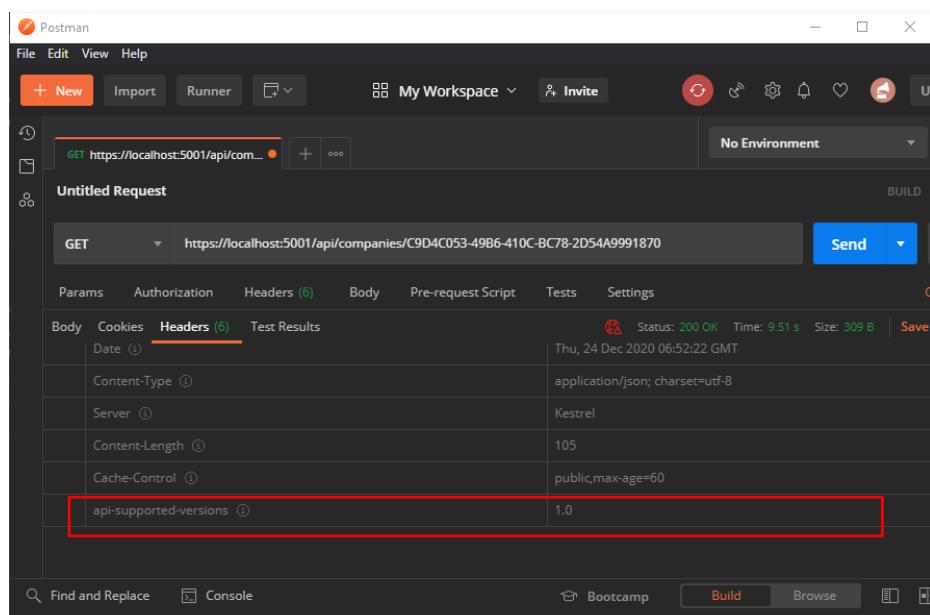
حالا باید این Profile را در بالای CompaniesController بگذاریم.

`[ResponseCache(CacheProfileName = "120SecondsDuration")]`

لازم به ذکر است که این Cache Rule در تمامی اکشن‌های درون کنترلر، به جز مواردی که قبل‌از پرایپری ResponseCache استفاده کرده‌اند (اکشن متدهای GetCompanyAsync) اعمال خواهد شد.

حالا اگر ریکوئست به اکشن متدهای GetCompanyAsync بفرستیم باید پرایپری Age بر روی عدد ۶۰ باشد.

<https://localhost:5001/api/companies/C9D4C053-49B6-410C-BC78-2D54A9991870>



و اگر Request به اکشن متدهای GetCompaniesAsync و GetCompaniesSync بر روی عدد ۱۲۰ باشد.

<https://localhost:5001/api/companies>

The screenshot shows the Postman interface with a successful GET request to `https://localhost:5001/api/companies`. The response status is `200 OK`, and the Headers section displays the following:

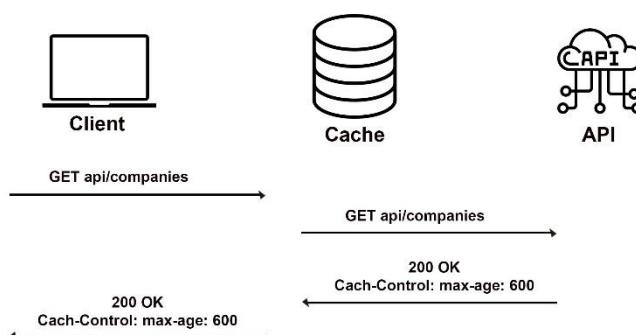
KEY	VALUE
Date	Thu, 24 Dec 2020 07:00:05 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	1027
Cache-Control	public,max-age=120
api-supported-versions	1.0

باید در مورد Validation models و Expiration Model بیشتر صحبت کنیم.

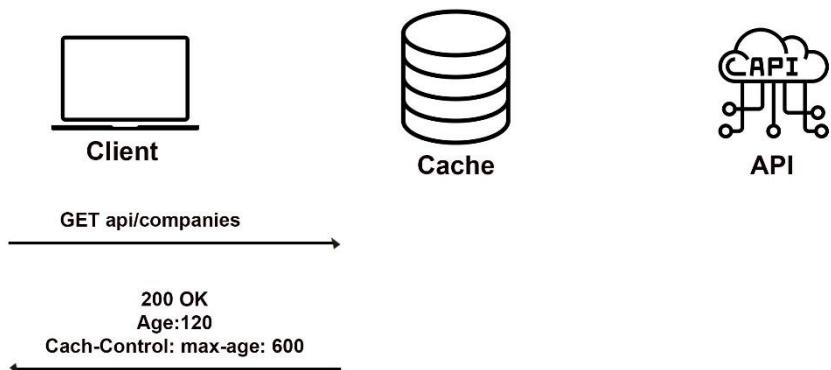
Expiration Model چیست؟

به سرور این امکان را می‌دهد که مشخص کند، ریسپانس منقضی شده یا خیر.

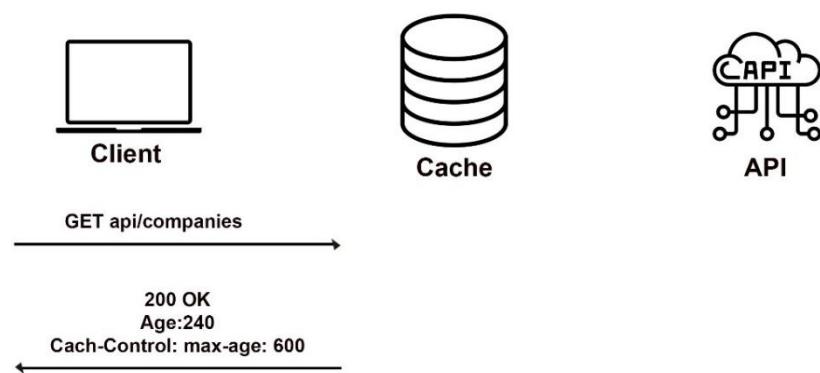
باید ببینیم که Expiration Model چگونه کار می‌کند.



- همانطور که می‌بینید کلاینت ریکوئستی را برای دریافت companies ارسال می‌کند.
- هیچ ورژن cache شده‌ای از این ریسپانس وجود ندارد بنابراین ریکوئست به API ارسال می‌شود.
- سپس API این ریسپانس را همراه با هدر Cache-Control ۶۰۰ ثانیه انقضا، برای کلاینت ارسال و آن را در cache ذخیره می‌کند.
- پس تا زمانی که ریسپانس Fresh باشد، از Cache ارائه شده و برای دستیابی به آن، از هدر Cache-Control استفاده می‌شود.
- اگر بعد از دو دقیقه ریکوئست مشابهی درخواست شود تصویر زیر اتفاق می‌افتد.



- همانطور که می‌بینید، ریسپانس ذخیره شده، با یک هدر Age ۱۲۰ ثانیه برگردانده می‌شود.
- اگر این یک Private Cache باشد، پس ریسپانس در مرورگر ذخیره خواهد شد بنابراین کلاینت دیگر، ریسپانس را از API می‌گیرد.
- اما اگر این یک Shared Cache باشد، کلاینت دیگر، قبل از دو دقیقه زمان اضافه، همان ریسپانس را می‌گیرد.



همانطور که می‌بینید در Shared Cache ریسپانس از cache ارائه می‌شود و دو دقیقه دیگر به هدر Age اضافه خواهد شد.

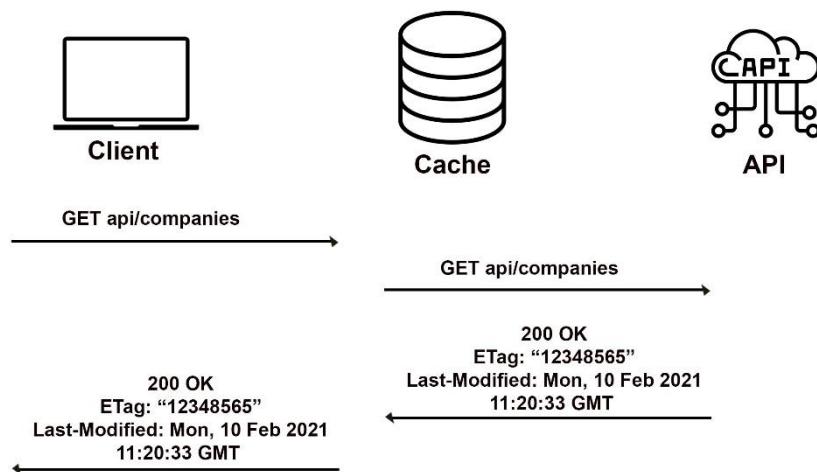
ما با نحوه کار Expiration Model آشنا شدیم حالا باید Validation Model را بررسی کنیم.

Validation Model چیست؟

Validation Model بررسی می‌کند که ریسپانس درون Cache، هنوز قابل استفاده است یا خیر؟ فرض کنیم که یک ریسپانس GetCompany به مدت ۳۰ دقیقه در Shared Cached گذاشته شد است. اگر کسی بعد از پنج دقیقه، آن شرکت را آپدیت کند، کلاینت باید ۲۵ دقیقه باقی‌مانده، ریسپانس اشتباه دریافت کند؛ چون هیچ اعتبارسنجی روی این داده Cache وجود ندارد.

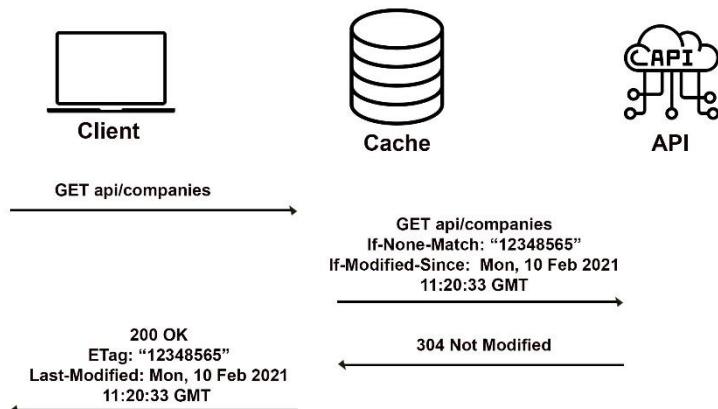
برای جلوگیری از این اتفاق، باید از اعتبارسنجی استفاده کنیم. استاندارد HTTP توصیه می‌کند در صورت امکان، به صورت ترکیبی از اعتبارسنجی‌های LastModified و ETag استفاده کنید.

باید ببینیم که اعتبارسنجی چگونه کار می‌کند.



- کلاینت ریکوئستی را ارسال می‌کند.
- این ریکوئست Cache نمی‌شود و به API ارسال خواهد شد.
- API ریسپانسی را که شامل Etag و Last-Modified است برمی‌گرداند.
- این ریسپانس، Cache شده و به کلاینت ارسال می‌شود.

پس از دو دقیقه، کلاینت همان ریکوئست را ارسال می‌کند.



- همانطور که می‌بینید Cache، ریکوئست را با هدرهای اضافی If-None-Match (که روی مقدار Last-Modified-From تنظیم شده) و If Modified-Since (که به مقدار Last-Modified تنظیم شده) به سمت API برمی‌گرداند.
 - اگر این Request با اعتبارسنجی‌ها بررسی شود، دیگر نیازی به ایجاد دوباره Response از سمت API نیست بنابراین 304 Not Modified status ارسال می‌کند.
 - بعد از آن، Response به طور منظم از Cache ارائه می‌شود.
 - البته اگر این بررسی انجام نشود، پس باید Response جدید ایجاد شود.
 - این تصویر ما را به این نتیجه می‌رساند که در Cache Shared اگر ریسپانس تغییر نکرده باشد، این Response باید فقط یک بار تولید شود.
- باید همه اینها را در یک مثال ببینیم.

پیاده‌سازی اعتبارسنجی

ابتدا باید در پروژه اصلی پکیج Marvin.Cache.Headers را نصب کنیم. این پکیج از هدرهای Expiration، Validation و Last-Modified، Etag، Expires، Cache-Control را پیاده‌سازی می‌کند.

```
Install-Package Marvin.Cache.Headers -Version 5.0.1 -ProjectName
CompanyEmployee.API
```

```
Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Install-Package Marvin.Cache.Headers -Version 5.0.1 -ProjectName CompanyEmployee.API
Restoring packages for D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj...
Installing NuGet package Marvin.Cache.Headers 5.0.1.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj (in 355 ms).
Successfully installed 'Marvin.Cache.Headers 5.0.1' to CompanyEmployee.API
Successfully installed 'Microsoft.AspNetCore.Http 2.2.2' to CompanyEmployee.API
Successfully installed 'Microsoft.AspNetCore.Http.Abstractions 2.2.0' to CompanyEmployee.API
Successfully installed 'Microsoft.AspNetCore.Http.Features 2.2.0' to CompanyEmployee.API
Successfully installed 'Microsoft.AspNetCore.Mvc.Abstractions 2.2.0' to CompanyEmployee.API
Successfully installed 'Microsoft.AspNetCore.Routing.Abstractions 2.2.0' to CompanyEmployee.API
131 %
```

در مرحله بعد پايد اکستنشن متده پايين را در کلاس ServiceExtensions اضافه کنيد.

```
public static void ConfigureHttpCacheHeaders(this IServiceCollection services) => services.AddHttpCacheHeaders();
```

خب حالا باید این سرویس در متدهای `ConfigureServices` و `Configure` رجیستر شود.

```
services.ConfigureHttpCacheHeaders();
```

و سپس Middleware پایین را در متدهای `Configure` اضافه نمایید.

```
app.UseResponseCaching();
```

```
app.UseHttpCacheHeaders();
```

حالا نوبت تست است.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>

Body	Cookies	Headers (10)	Test Results	Status: 200 OK	Time: 8.70 s	Size: 476 B	Save Response
KEY	VALUE						
Date ①	Sat, 26 Dec 2020 11:14:56 GMT						
Content-Type ①	application/json; charset=utf-8						
Server ①	Kestrel						
Content-Length ①	97						
Cache-Control ①	public,max-age=60						
Expires ①	Sat, 26 Dec 2020 11:15:56 GMT						
Last-Modified ①	Sat, 26 Dec 2020 11:14:56 GMT						
ETag ①	"EE46C0C003967241F0C9D81E52678187"						
Vary ①	Accept, Accept-Language, Accept-Encoding						
api-supported-versions ①	1.0						

همانطور که می‌بینید، ما تمام هدرهای موردنیاز را ایجاد کردیم. انقضای پیش فرض روی ۶۰ ثانیه تنظیم شده است.

اگر این Request را یک بار دیگر ارسال کنیم، یک هدر Age می‌گیریم.

KEY	VALUE
Date	Sat, 26 Dec 2020 11:21:42 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	97
Cache-Control	public,max-age=60
Expires	Sat, 26 Dec 2020 11:22:42 GMT
Last-Modified	Sat, 26 Dec 2020 11:21:42 GMT
Age	8
ETag	"EE46C0C003967241F0C9D81E52678187"
Vary	Accept, Accept-Language, Accept-Encoding

پیکربندی هدرهای Validation و Expiration

برای پیکربندی هدرهای Validation و Expiration باید تغییراتی در متد ConfigureHttpCacheHeaders دهیم.

ابتدا ServiceExtensions Namespace را به کلاس پایین را به اضافه کنید.

```
using Marvin.Cache.Headers;
```

و بعد متد ConfigureHttpCacheHeaders را تغییر دهید.

```
public static void ConfigureHttpCacheHeaders(this IServiceCollection services) =>
    services.AddHttpCacheHeaders(
        (expirationOpt) =>
    {
        expirationOpt.MaxAge = 65;
        expirationOpt.CacheLocation = CacheLocation.Private;
    },
        (validationOpt) =>
    {
        validationOpt.MustRevalidate = true;
    });
});
```

این پیکربندی یک Private cache با طول عمر ۶۵ ثانیه ایجاد می‌کند.

خب حالا دوباره Request را ارسال کنید.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>

KEY	VALUE
Date ①	Sat, 26 Dec 2020 11:31:22 GMT
Content-Type ①	application/json; charset=utf-8
Server ①	Kestrel
Content-Length ①	97
Cache-Control ①	private,max-age=65,must-revalidate
Expires ①	Sat, 26 Dec 2020 11:32:28 GMT
Last-Modified ①	Sat, 26 Dec 2020 11:31:23 GMT
ETag ①	"EE46C0C003967241F0C9D81E52678187"
Vary ①	Accept, Accept-Language, Accept-Encoding
api-supported-versions ①	1.0

می‌بینید که تغییرات اعمال شده است. از آنجا که این یک Private cache است پس API ما آن را Cache نمی‌کند.

ما می‌توانیم این پیکربندی را با استفاده از اtribut‌های `HttpCacheExpiration` و `HttpCacheValidation` روی اکشن متدها کنترل بگذاریم.

```
[HttpGet("{id}", Name = "CompanyById")]
[HttpCacheExpiration(CacheLocation = CacheLocation.Public, MaxAge = 60)]
[HttpCacheValidation(MustRevalidate = false)]
public async Task<IActionResult> GetCompanyAsync(Guid id)
```

البته `namespace` پایین را هم در این کنترلر اضافه کنید.

```
using Marvin.Cache.Headers;
```

برای استفاده از پیکربندی سطح `Resource`, باید از اtribut‌های `HttpCacheExpiration` استفاده کنیم.

حالا اگر `Request` پایین را ارسال کنید مقادیر `Global` را می‌بینید.

<https://localhost:5001/api/companies>

Body	Cookies	Headers (10)	Test Results
KEY	VALUE		
Date ①	Sun, 27 Dec 2020 06:30:18 GMT		
Content-Type ①	application/json; charset=utf-8		
Server ①	Kestrel		
Content-Length ①	2039		
Cache-Control ①	private,max-age=65,must-revalidate		
Expires ①	Sun, 27 Dec 2020 06:31:23 GMT		
Last-Modified ①	Sun, 27 Dec 2020 06:30:18 GMT		
ETag ①	"BEC6467BFDAD2C7CC9C3395BF7A3761E"		
Vary ①	Accept, Accept-Language, Accept-Encoding		
api-supported-versions ①	1.0		

و اگر Request پایین را بفرستید نتیجه متفاوت خواهد بود.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>

Body	Cookies	Headers (10)	Test Results
KEY	VALUE		
Date ①	Sun, 27 Dec 2020 06:24:43 GMT		
Content-Type ①	application/json; charset=utf-8		
Server ①	Kestrel		
Content-Length ①	97		
Cache-Control ①	public,max-age=60		
Expires ①	Sun, 27 Dec 2020 06:25:43 GMT		
Last-Modified ①	Sun, 27 Dec 2020 06:24:43 GMT		
ETag ①	"EE46C0C003967241FOC9D81E52678187"		
Vary ①	Accept, Accept-Language, Accept-Encoding		
api-supported-versions ①	1.0		

چیست؟ Rate Limiting

Rate Limiting، ترافیک ورودی وب سایت را مدیریت می کند. این قابلیت به ما امکان می دهد تا از API در برابر تعداد ریکوئست های زیاد، که باعث از بین رفتن پرフォرمنس می شوند محافظت کنیم.

به عنوان مثال:

ما می توانیم مشخص کنیم که هر کلاینت در هر ساعت فقط ۱۰۰ ریکوئست به API داشته باشند. یا اینکه هر کلاینت را به حداقل ۱۰۰۰ ریکوئست در روز محدود کنیم.

ریکوئست های مجاز با XRate-Limit شروع می شوند. اطلاعات این ریکوئست ها را باید در هدرهای ریسپانس بررسی کنید.

هدر ریکوئست‌های مجاز شامل اطلاعات زیر هستند :

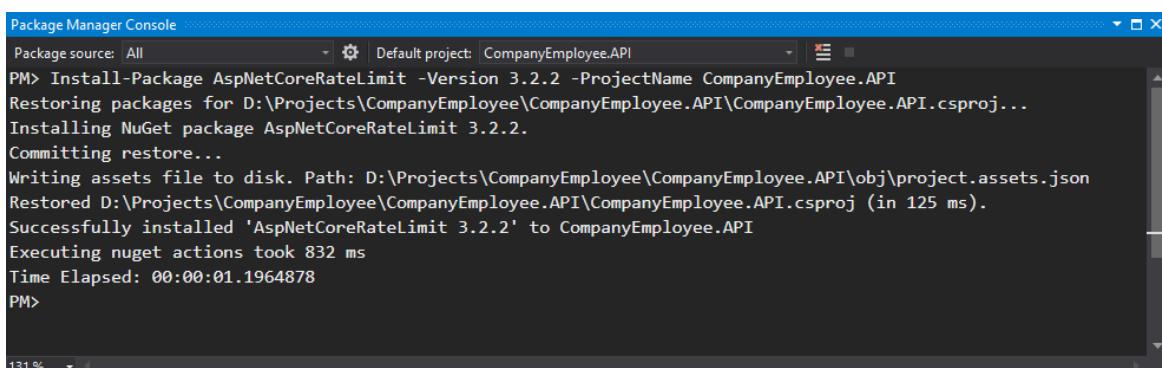
- **X-Rate-Limit-Limit** : دوره Rate-Limit را مشخص می‌کند.
- **X-Rate-Limit-Remaining** : تعداد ریکوئست‌های باقیمانده را مشخص می‌کند.
- **X-Rate-Limit-Reset** : اطلاعات تاریخ و زمان در مورد محدودیت ریکوئست است.

اگر تعداد درخواست‌ها بیش از حد مجاز شود، 429 status code برگردانده می‌شود و هدر Retry-After به هدرهای ریسپانس اضافه خواهد شد.

پیاده‌سازی Rate-Limit

برای شروع، باید پکیج AspNetCoreRateLimit را در پروژه اصلی نصب کنید.

```
Install-Package AspNetCoreRateLimit -Version 3.2.2 -ProjectName  
CompanyEmployee.API
```



The screenshot shows the Package Manager Console window with the following output:

```
Package Manager Console  
Package source: All Default project: CompanyEmployee.API  
PM> Install-Package AspNetCoreRateLimit -Version 3.2.2 -ProjectName CompanyEmployee.API  
Restoring packages for D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj...  
Installing NuGet package AspNetCoreRateLimit 3.2.2.  
Committing restore...  
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\CompanyEmployee.API\obj\project.assets.json  
Restored D:\Projects\CompanyEmployee\CompanyEmployee.API\CompanyEmployee.API.csproj (in 125 ms).  
Successfully installed 'AspNetCoreRateLimit 3.2.2' to CompanyEmployee.API  
Executing nuget actions took 832 ms  
Time Elapsed: 00:00:01.1964878  
PM>
```

حالا باید این سرویس را رجیستر کنیم. اما از آنجاییکه این پکیج برای ذخیره شمارندها و Rule‌های خود از MemoryCache استفاده می‌کند پس باید MemoryCache را نیز در متدهای Configuration Register کنیم.

```
services.AddMemoryCache();
```

خب حالا باید یک اکستنشن متدهای Rate-Limit داشته باشیم. بنابراین اکستنشن متدهای ServiceExtensions را در کلاس Configuration Register کنید.

نام Namespace موردنیاز :

```
using AspNetCoreRateLimit;
```

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

کدهای اکستنشن Metd ConfigurationRateLimitingOptions :

```

public static void ConfigureRateLimitingOptions(this IServiceCollection
services)
{
    var rateLimitRules = new List<RateLimitRule>
    {
        new RateLimitRule
        {
            Endpoint = "*",
            Limit= 3,
            Period = "5m"
        }
    };
    services.Configure<IpRateLimitOptions>(opt =>
    {
        opt.GeneralRules = rateLimitRules;
    });
    services.AddSingleton<IRateLimitCounterStore,
MemoryCacheRateLimitCounterStore>();

    services.AddSingleton<IIpPolicyStore, MemoryCacheIpPolicyStore>();

    services.AddSingleton<IRateLimitConfiguration,
RateLimitConfiguration>();
}

```

بررسی کد :

- ما در این کد یک RateLimitRule ایجاد کردیم که سه ریکوئست را در یک دوره ۵ دقیقه ای برای تمام Endpoint ها مجاز می کند.
- سپس برای اضافه کردن Rule بالا، IpRateLimitOptions را پیکربندی کردیم.
- در پایان باید IRateLimitConfiguration و IIpPolicyStore و IRateLimitCounterStore را به صورت Singleton Register کنیم.

حالا نوبت استفاده کردن این اکستنشن متده ConfigurServices است.

```

services.AddMemoryCache();
services.ConfigureRateLimitingOptions();
services.AddHttpContextAccessor();

```

البته باید `ConfigureMiddleware` را در متدهای `Startup` اضافه کنید.

```

app.UseIpRateLimiting();
app.UseRouting();

```

فراموش نکنید که `Startup` نامداری `Namespace` را هم در کلاس `Startup` قرار دهید.

```
using AspNetCoreRateLimit;
```

خب حالا نوبت تست است.

<https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3>

KEY	VALUE
Date	Sun, 27 Dec 2020 08:03:43 GMT
Content-Type	application/json; charset=utf-8
Server	Kestrel
Content-Length	99
Cache-Control	public,max-age=60
Expires	Sun, 27 Dec 2020 08:04:43 GMT
Last-Modified	Sun, 27 Dec 2020 08:03:43 GMT
ETag	"E86602264FF567A7242381092065D511"
Vary	Accept, Accept-Language, Accept-Encoding
api-supported-versions	1.0
X-Rate-Limit-Limit	5m
X-Rate-Limit-Remaining	2
X-Rate-Limit-Reset	2020-12-27T08:08:39.5720801Z

همانطور که می‌بینید در مدت زمان ۵ دقیقه، دو درخواست باقی مانده است. اگر در مدت زمان پنج دقیقه، سه درخواست اضافی ارسال کنیم ریسپانس متفاوتی خواهیم گرفت.

KEY	VALUE
Date	Sun, 27 Dec 2020 08:12:33 GMT
Content-Type	text/plain
Server	Kestrel
Cache-Control	private,max-age=65,must-revalidate
Transfer-Encoding	chunked
Expires	Sun, 27 Dec 2020 08:13:38 GMT
Retry-After	290
Vary	Accept, Accept-Language, Accept-Encoding

همانطور که می‌بینید 429 Too Many Requests دریافت کردیم.
اگر body را بررسی کنیم باید نتیجه پایین را ببینیم.

The screenshot shows a browser-based API testing tool. At the top, there are tabs for Body, Cookies, Headers (8), and Test Results. The Body tab is selected. Below the tabs, there are buttons for Pretty, Raw, Preview, Visualize, Text, and a copy icon. The status bar at the top right indicates Status: 429 Too Many Requests, Time: 18 ms, and Size: 351 B. A 'Save Response' button is also present. The main content area contains the following text:
1 API calls quota exceeded! maximum admitted 3 per 5m.

فصل سیزدهم : Identity و JWT

آنچه خواهید آموخت:

- ASP.NET Identity چیست? ➤
- Authorization و Authentication چیست? ➤
- JWT چیست? ➤

Identity و JWT

User Authentication بخش مهمی از هر اپلیکیشن است و به پروسه تأیید هویت کاربران یک اپلیکیشن، اشاره دارد.

اگر به روند کار آشنا نباشید، پیاده‌سازی این ویژگی می‌تواند کار سختی باشد و زمان زیادی از شما بگیرد.

بنابراین در این بخش، می‌خواهیم Authentication و Authorization در ASP.NET Core را با استفاده از Identity و JWT (Json Web Token) بیاموزیم.

ما می‌خواهیم گام به گام نحوه ادغام Identity در پروژه موجود و سپس نحوه پیاده‌سازی JWT را برای عملیات Authentication و Authorization توضیح دهیم.

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 نالمید کننده به نظر برسد. اما نگران نباشید، با دانستن برخی موضوعات می‌توانیم از بسیاری از این حملات جلوگیری کنیم.

در این فصل می‌خواهیم به نحوه 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 لایکین شده امکانات متفاوتی ارائه دهید.

ASP.NET Identity یک Membership system در اپلیکیشن‌های وب است که قابلیت عضویت، ورود به سیستم و داده‌های کاربر را اضافه می‌کند. Identity می‌تواند با استفاده از

دیتابیس SQL Server پیکربندی شود تا نام کاربر، کلمه عبور و اطلاعات پروفایل را ذخیره نماید.

باید کمی بیشتر به این موضوع پردازیم و این قابلیت فوق العاده را به این اپلیکیشن اضافه نماییم.

چیست؟ Authorization و Authentication

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

- فرآیند ایجاد کاربر برای ورود به برنامه است.
- میزان دسترسی کاربران و کنترل میزان دسترسی آنها به اپلیکیشن را مشخص می نماید.

به عبارت دیگر، Authorization مشخص می کند چه کسی وارد سیستم شده و می گوید، به چه چیزهایی باید دسترسی داشته باشد.

ساده ترین حالت برای Authorization این است که حداقل، کاربر باید Authenticate شده باشد. این کار توسط اضافه کردن اtribut [Authorize] به بالای اکشن متدها یا Controller ها انجام می شود.

نکته!!

اتribut [Authorize] را می توان در هر جای برنامه به کار برد، اما استفاده از آن در بالای کنترلرهای اکشن متدها به این دلیل است که کنترل کنیم چه کاربری به چه اکشنی دسترسی دارد.

باید با هم Identity را به این اپلیکیشن اضافه نماییم.

- پکیج Microsoft.AspNetCore.Identity.EntityFrameworkCore را در پروژه Entities اضافه کنید.

```
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore -Version 5.0.2 -ProjectName Entities
```

```

Package Manager Console
Package source: All | Default project: CompanyEmployeeAPI
PM> Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore -Version 5.0.2 -ProjectName Entities
Restoring packages for D:\Projects\CompanyEmployee\Entities\Entities.csproj...
Installing NuGet package Microsoft.AspNetCore.Identity.EntityFrameworkCore 5.0.2.
Committing restore...
Writing assets file to disk. Path: D:\Projects\CompanyEmployee\Entities\obj\project.assets.json
Restored D:\Projects\CompanyEmployee\Entities\Entities.csproj (in 24 ms).
Successfully installed 'Microsoft.AspNetCore.Cryptography.Internal 5.0.2' to Entities
Successfully installed 'Microsoft.AspNetCore.Cryptography.KeyDerivation 5.0.2' to Entities
Successfully installed 'Microsoft.AspNetCore.Identity.EntityFrameworkCore 5.0.2' to Entities
Successfully installed 'Microsoft.EntityFrameworkCore.Relational 5.0.2' to Entities
Successfully installed 'Microsoft.Extensions.Configuration.Abstractions 5.0.0' to Entities
Successfully installed 'Microsoft.Extensions.Identity.Core 5.0.2' to Entities
131%

```

- پس از نصب، در فolder در نام User / Models کلاسی با نام Entities ایجاد نمایید.

```
using Microsoft.AspNetCore.Identity;
```

```
namespace Entities.Models
{
    public class User : IdentityUser
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

این کلاس باید از کلاس ASP.NET Core Identity `IdentityUser` ارائه شده ارث بری کند.

- از EF Core برای ذخیره User Account استفاده می‌کند. به همین جهت بعد از اضافه نمودن User، باید DbContext را هم کمی تغییر دهیم.

قبل از تغییر کلاس CompanyEmployeeDbContext، یک کلاس با نام Entities/Configuration در مسیر `IdentityUserLoginConfiguration` ایجاد کنید.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
```

```
namespace Entities.Configuration
{
```

```

public class IdentityUserLoginConfiguration :
    IEntityTypeConfiguration<IdentityUserLogin<int>>
{
    public void Configure(EntityTypeBuilder<IdentityUserLogin<int>>
        builder)
    {
        builder.HasKey();
    }
}

```

: CompanyEmployeeDbContext کلاس

```

using Entities.Configuration;
using Entities.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Entities
{
    public class CompanyEmployeeDbContext : IdentityDbContext<User>
    {

        public CompanyEmployeeDbContext(DbContextOptions options):
            base(options)
        {

        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.ApplyConfiguration(new CompanyConfiguration());
            modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
            modelBuilder.ApplyConfiguration(new
                IdentityUserLoginConfiguration());
            base.OnModelCreating(modelBuilder);
        }

        public DbSet<Company> Companies { get; set; }
    }
}

```

```

        public DbSet<Employee> Employees { get; set; }

    }

}

```

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

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

- حالا باید Identity خود را پیکربندی کنید.

ابتدا پکیج پایین را در پروژه اصلی نصب کنید.

```
Install-Package Microsoft.AspNetCore.Authentication.Cookies -Version 2.2.0 -ProjectName CompanyEmployee.API
```

The screenshot shows the Package Manager Console window with the command "Install-Package Microsoft.AspNetCore.Authentication.Cookies -Version 2.2.0 -ProjectName CompanyEmployee.API" entered. The output pane displays the progress of the package restoration and installation, showing successful installs for various Microsoft.AspNetCore.Authentication.* packages.

سپس سه کلاس در پایین را در کلاس Namespace اضافه کنید.

```

using Entities.Models;
using Microsoft.AspNetCore.Identity;

```

خب حالا یک اکستنشن متدهای ServiceExtensions را در کلاس ConfigureIdentity اضافه نمایید.

```

public static void ConfigureIdentity(this IServiceCollection services)
{
    var builder = services.AddIdentityCore<User>(o =>
    {
        o.Password.RequireDigit = true;
        o.Password.RequireLowercase = false;
        o.Password.RequireUppercase = false;
    });
}

```

```

    o.Password.RequireNonAlphanumeric = false;
    o.Password.RequiredLength = 10;
    o.User.RequireUniqueEmail = true;
}

builder = new IdentityBuilder(builder.UserType, typeof(IdentityRole),
builder.Services);
builder.AddEntityFrameworkStores<CompanyEmployeeDbContext>()
.AddDefaultTokenProviders();
}

```

بررسی کد:

- در این کد Identity را با استفاده از متده AddIdentityCore اضافه و آن را برای نوع User پیکربندی کردیم.
- همانطور که می بینید پارامترهای مختلف پیکربندی را در متده AddIdentityCore استفاده کردیم.
- در پایان یک IdentityBuilder را همراه با token EntityFrameworkStores ایجاد و این را همراه با provider پیش فرض اضافه کردیم.

خب حالا این متده را باید در متده ConfigureServices ثبت کنیم.

```

services.AddAuthentication();
services.ConfigureIdentity();

```

حالا دو پایین را در متده Configure Middleware اضافه کنید.

```

app.UseAuthentication();
app.UseAuthorization();

```

ایجاد جداول و اضافه کردن Role ها

ما تمام چیزهایی که لازم بود را آماده کردیم حالا باید به سراغ اعمال تغییرات دیتابیس برویم. تمام کاری که باید انجام دهیم ایجاد و اعمال Migration است.

Add-Migration CreatingIdentityTables -Project CompanyEmployee.API

```
Package Manager Console
PM> Add-Migration CreatingIdentityTables -Project CompanyEmployee.API
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```

حالا دیتابیس را آپدیت کنید.

Update-Database

```
Package Manager Console
PM> Update-Database
Build started...
Build succeeded.
Done.
PM>
```

باید با هم نمای دیتابیس و جداول اضافه شده را ببینیم.

The screenshot shows the Object Explorer window in SSMS. It is connected to a local SQL Server instance (15.0.2000.5 - sa). Under the 'CompanyEmployee' database, the 'Tables' node is expanded, showing several system and application tables. A red box highlights the following tables:

- dbo._EFMigrationsHistory
- dbo.AspNetRoleClaims
- dbo.AspNetRoles
- dbo.AspNetUserClaims
- dbo.AspNetUserLogins
- dbo.AspNetUserRoles
- dbo.AspNetUsers
- dbo.AspNetUserTokens
- dbo.Companies
- dbo.Employees
- dbo.IdentityUserLogin<int>

اگر جدول AspNetUsers را باز کنید، ستون‌های LastName و FirstName را همراه با سایر ستون‌ها می‌بینید.

	Id	FirstName	LastName	UserName	NormalizedUs...	Email
*	NULL	NULL	NULL	NULL	NULL	NULL

حالا باید در جدول AspNetRoles، چند Role وارد کنیم. برای انجام این کار، در فولدر Entities / Configuration یک کلاس با نام RoleConfiguration ایجاد کنید.

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

```

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Entities.Configuration
{
    public class RoleConfiguration :
        IEntityTypeConfiguration<IdentityRole>
    {
        public void Configure(EntityTypeBuilder<IdentityRole> builder)
        {
            builder.HasData(
                new IdentityRole
                {
                    Name = "Manager",
                    NormalizedName = "MANAGER"
                },
                new IdentityRole
                {
                    Name = "Administrator",
                    NormalizedName = "ADMINISTRATOR"
                }
            );
        }
    }
}

```

حالا این کلاس را در متدهای OnModelCreating و OnModelCreatingCore اضافه کنید.

```

using Entities.Configuration;
using Entities.Models;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace Entities
{
    public class CompanyEmployeeDbContext : IdentityDbContext<User>

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

```

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new CompanyConfiguration());
    modelBuilder.ApplyConfiguration(new EmployeeConfiguration());
    modelBuilder.ApplyConfiguration(new
        IdentityUserLoginConfiguration());
    modelBuilder.ApplyConfiguration(new RoleConfiguration());

    base.OnModelCreating(modelBuilder);
}

public DbSet<Company> Companies { get; set; }
public DbSet<Employee> Employees { get; set; }

}
}

```

سپس Migration پایین را اجرا و دیتابیس را آپدیت کنید.

Add-Migration AddedRolesToDb -Project CompanyEmployee.API

```

Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Add-Migration AddedRolesToDb -Project CompanyEmployee.API
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
131%

```

Update-Database

```

Package Manager Console
Package source: All Default project: CompanyEmployee.API
PM> Update-Database
Build started...
Build succeeded.
Done.
PM>
121%

```

اگر جدول AspNetRoles را بررسی کنید باید این دو Role جدید را بینید.

DESKTOP-RPDOFNO....dbo.AspNetRoles				
	Id	Name	NormalizedNa...	ConcurrencySt...
▶	2fbe8ac8-df37...	Manager	MANAGER	252e2474-0a17...
	ce2c34b0-9c60...	Administrator	ADMINISTRATOR	392fd69e-ff07-4...
*	NULL	NULL	NULL	NULL

ایجاد User

برای ایجاد User، ابتدا یک کنترلر جدید با نام AuthenticationController ایجاد کنید.

```
using AutoMapper;
using Contracts.IServices;
using Entities.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/authentication")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;
        private readonly UserManager<User> _userManager;
        public AuthenticationController(ILoggerManager logger, IMapper mapper, UserManager<User> userManager)
        {
            _logger = logger;
            _mapper = mapper;
            _userManager = userManager;
        }
    }
}
```

: بررسی کد

- به جز بخش `<TUser>`، تمام کدهای بالا آشنا است.
- `UserManager` سرویسی برای مدیریت کاربران است که توسط `Identity` ارائه شده.
- در اینجا نیازی به ریپازیتوری نداریم چون `UserManager` تمام نیازمندی‌های ما را فراهم می‌کند.

حالا نوبت ایجاد کلاس `DataTransferObjects` در فolder `UserForRegistrationDto` است.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public class UserForRegistrationDto
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        [Required(ErrorMessage = "Username is required")]
        public string UserName { get; set; }
        [Required(ErrorMessage = "Password is required")]
        public string Password { get; set; }
        public string Email { get; set; }
        public string PhoneNumber { get; set; }
        public ICollection<string> Roles { get; set; }
    }
}
```

سپس یک `MappingProfile` در کلاس `Mapping Rule` اضافه کنید..

```
CreateMap<UserForRegistrationDto, User>();
```

در پایان باید اکشن‌متد `RegisterUser` را ایجاد کنیم.

```
using AutoMapper;
using CompanyEmployee.API.Infrastructure.ActionFilters;
using Contracts.IServices;
using Entities.DataTransferObjects;
```

```

using Entities.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/authentication")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;
        private readonly UserManager<User> _userManager;
        public AuthenticationController(ILoggerManager logger, IMapper mapper, UserManager<User> userManager)
        {
            _logger = logger;
            _mapper = mapper;
            _userManager = userManager;
        }

        [HttpPost]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> RegisterUserAsync([FromBody]
UserForRegistrationDto userForRegistration)
{
    var user = _mapper.Map<User>(userForRegistration);
    var result = await _userManager.CreateAsync(user,
userForRegistration.Password);

    if (!result.Succeeded)
    {
        foreach (var error in result.Errors)
        {

```

```

        ModelState.TryAddModelError(error.Code,
            error.Description);
    }

    return BadRequest(ModelState);
}

await _userManager.AddToRolesAsync(user,
    userForRegistration.Roles);

return StatusCode(201);
}
}
}

```

بررسی کد :

- ابتدا برای اعتبارسنجی مدل، یک اکشن فیلتر را بالای اکشن متده خود قرار دادیم.
- سپس مدل ورودی را به نوع User مپ کردیم.
- حالا برای ایجاد کاربر در دیتابیس، متده CreateAsync را صدای زنیم.
- اگر عمل ایجاد کاربر موفقیت آمیز باشد، کاربر در دیتابیس ذخیره می شود. در غیر این صورت پیغام خطأ را برمی گرداند.
- اگر پیغام خطأ برگردانده شود باید آنها را به ModelState اضافه کنیم.
- در پایان اگر کاربری ایجاد شود باید آن را به Role های خود متصل کنیم و سپس 201 Created را برگردانیم.

نکته!!

در صورت تمایل، قبل از صدا زدن AddToRoleAsync یا AddToRolesAsync، می توانید **Role** های موجود در دیتابیس را بررسی کنید. البته برای این کارباید **RoleManager <TRole>** را در کنترلر استفاده کنید.

حالا می توانیم این را تست کنیم.

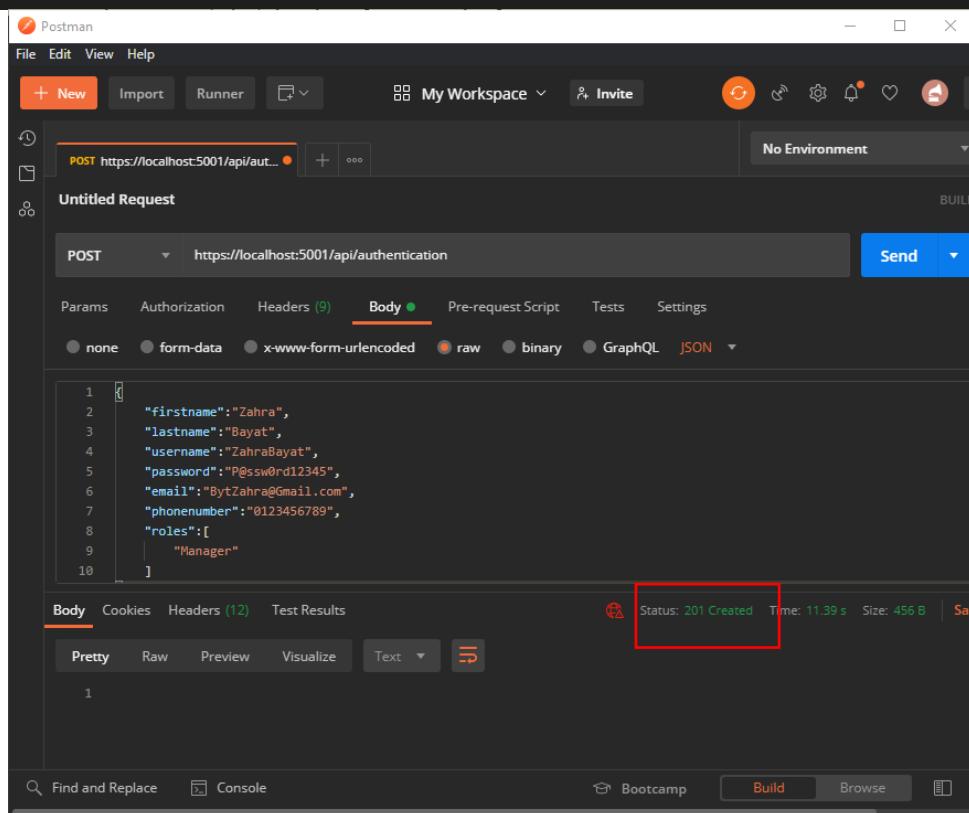
قبل از تست، بهتر است مقدار Rate Limit را از ۳۰ به ۳ (کلاس ServiceExtensions، متدهای ConfigureRateLimitingOptions) افزایش دهیم تا تست پیچیده نشود.

ابتدا یک درخواست معتبر ارسال می‌کنیم.

<https://localhost:5001/api/authentication>

body

```
{  
    "firstname": "Zahra",  
    "lastname": "Bayat",  
    "username": "ZahraBayat",  
    "password": "P@ssw0rd12345",  
    "email": "BytZahra@gmail.com",  
    "phonenumber": "0123456789",  
    "roles": [  
        "Manager"  
    ]  
}
```



همانطور که می‌بینید 201 گرفتیم. این یعنی کاربر ایجاد و به Role اضافه شده است.

خب حالا باید با مدل‌های نامعتبر این API را تست کنیم.

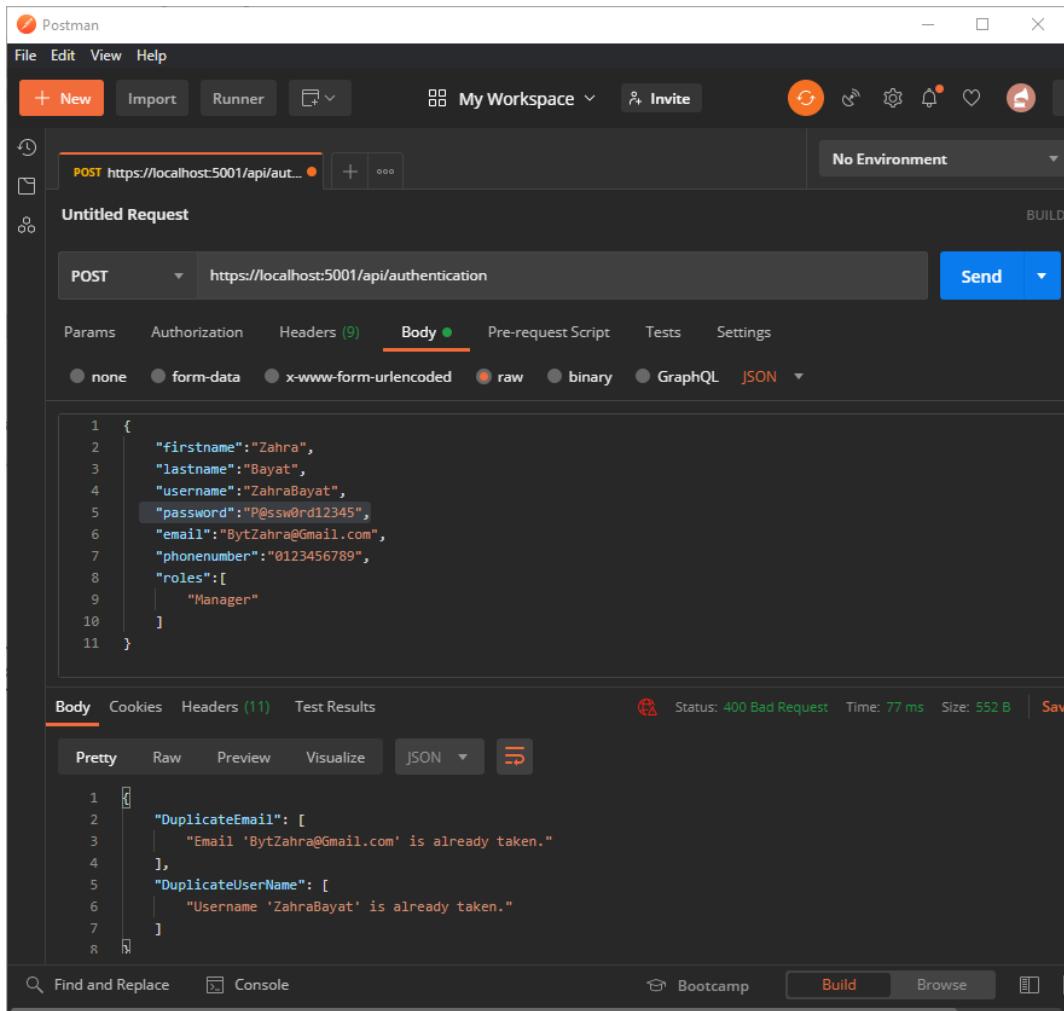
اگر UserName خالی باشد:

The screenshot shows the Postman application interface. A POST request is being made to `https://localhost:5001/api/authentication`. The request body is a JSON object with fields: `firstname`, `lastname`, `password`, `email`, `phonenumber`, and `roles`. The `password` field is currently empty. The response status is 422 Unprocessable Entity, indicating a validation error. The error message in the response body is: `"Username": ["Username is required"]`.

اگر Password خالی باشد :

This screenshot is identical to the one above, showing a POST request to `https://localhost:5001/api/authentication` with an empty `password` field. The response status is 422 Unprocessable Entity, and the error message in the response body is: `"Password": ["Password is required"]`.

در پایان اگر کاربری با همان نام کاربری و ایمیل ایجاد کنیم.

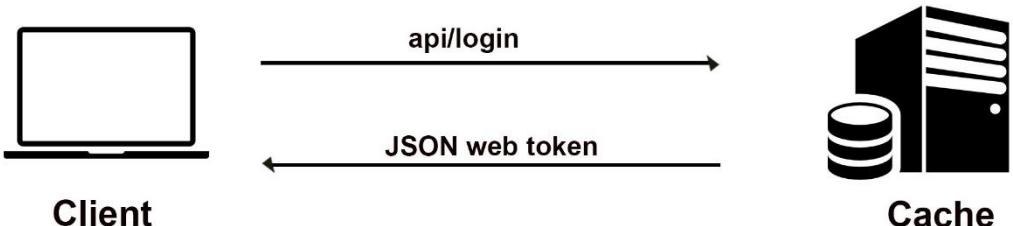


عالی شد همه چیز طبق برنامه کار می‌کند. حالا می‌توانیم به سراغ JWT برویم.

فرآیند Login

قبل از پیاده‌سازی Authentication و Authorization باید نگاهی اجمالی به فرآیند Login داشته باشیم.

اپلیکیشن یک فرم ورود به سیستم دارد که در آن، کاربر نام کاربری و رمز عبور خود را وارد و دکمه ورود را می‌زند. پس از زدن دکمه ورود، کلاینت (به عنوان مثال : مرورگر وب) داده‌های کاربر را به API سرور می‌فرستد.



وقتی سرور اعتبار کاربر را بررسی و تأیید کرد، باید یک JWT Token برای کلاینت ارسال کند.

JWT چیست؟

JWT یک شی JavaScript است که حاوی برخی از اطلاعات کاربر لاینین شده (مثل نام کاربری، Role‌های کاربر یا برخی اطلاعات دیگر) می‌باشد. این شی به صورت ایمن، داده‌ها را بین دو طرف منتقل می‌کند.

شکل کلی JWT به صورت پایین است.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHM
I6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

همانطور که می‌بینید JWT از سه قسمت تشکیل شده که هر قسمت با رنگ متفاوت نشان داده شده است.

- **Header:** اولین قسمت JWT هدر است که یک شی JSON رمزگذاری شده با فرمت Base64 می‌باشد. این قسمت اطلاعاتی مانند نوع token و نام الگوریتم را در خود دارد.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- **Payload:** بعد از هدر، ما یک Payload داریم که این هم یک شی JSON رمزگذاری شده با فرمت Base64 می‌باشد. Payload حاوی اطلاعاتی مانند نام کاربر لاینین شده است. به عنوان مثال :

می‌تواند حاوی `Id` کاربر، `Subject` کاربر و اطلاعاتی در مورد اینکه آیا کاربر Admin است یا خیر؟ باشد.

توجه داشته باشید که JWT، رمزگذاری شده نیست و می‌تواند با هر Base64 رمزگشای شود. بنابراین هرگز اطلاعات حساس را در Payload قرار ندهید.

```
{  
  "sub": "1234567890",  
  "name": "Zahra Bayat",  
  "iat": 1516239022  
}
```

در پایان، قسمت **Signature** را داریم. معمولاً سرور از این قسمت برای برسی اینکه آیا توکن حاوی اطلاعات معتبری است یا خیر استفاده می‌کند. این امضای دیجیتال، از ترکیب Header و Payload و بر اساس یک Secret Key که فقط سرور می‌داند تولید می‌شود.

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  superSecretKey  
)  secret base64 encoded
```

وقتی کلاینت سعی در تغییر مقادیر موجود در Payload داشته باشد باید Signature را دوباره تولید کند بنابراین به Secret key که سرور آن را می‌داند، نیاز دارد.

وقتی کلاینت این Signature جعلی را ایجاد و ارسال کرد، در سمت سرور Signature اصلی با Signature دریافت شده مقایسه می‌شود و به راحتی می‌توان از جعلی بودن اطلاعات مطمئن شد.

بنابراین ما به راحتی می‌توانیم، با مقایسه امضاهای دیجیتالی، از یکپارچگی داده‌های خود اطمینان حاصل کنیم. این دلیل استفاده ما از JWT است.

پیکربندی JWT

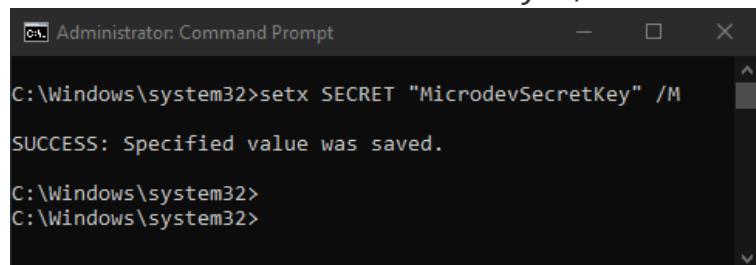
برای پیکربندی JWT، اولین قدم ذخیره اطلاعات audience و issuer در فایل appsettings.json است.

```
{  
  "ConnectionStrings": {  
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated  
    Security=true"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  "JwtSettings": {  
    "validIssuer": "MicrodevAPI",  
    "validAudience": "https://localhost:5001"  
  },  
  "AllowedHosts": "*"  
}
```

همانطور که بالاتر گفتیم، در سمت سرور به یک secret key نیاز داریم. بنابراین باید این Key را ایجاد و در یک متغیر سیستمی ذخیره کنیم.

برای ایجاد یک متغیر سیستمی cmd را به صورت Admin باز و دستور زیر را اجرا کنید.

```
setx SECRET "MicrodevSecretKey" /M
```



The screenshot shows an 'Administrator: Command Prompt' window. The command 'setx SECRET "MicrodevSecretKey" /M' is entered and executed. The output shows 'SUCCESS: Specified value was saved.' The prompt then returns to the command line with 'C:\Windows\system32>'.

این دستور یک متغیر سیستمی با نام MicrodevSecretKey و با مقدار SECRET ایجاد می کند. با استفاده از M/ مشخص می کنیم که متغیر باید از نوع سیستم باشد نه local. حالا باید اکستنشن متدها برای پیکربندی JWT در کلاس ServiceExtensions ایجاد کنیم. Namespace های پایین را قبل از نوشتن این اکستنشن متدها در کلاس ServiceExtensions اضافه کنید.

```
using System;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

    : ConfigureJWT متدهای اکستنشن متدها
```

```
public static void ConfigureJWT(this IServiceCollection services,
IConfiguration configuration)
{
    var jwtSettings = configuration.GetSection("JwtSettings");
    var secretKey = Environment.GetEnvironmentVariable("SECRET");
    services.AddAuthentication(opt => {
        opt.DefaultAuthenticateScheme =
        JwtBearerDefaults.AuthenticationScheme;
        opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = jwtSettings.GetSection("validIssuer").Value,
            ValidAudience =
                jwtSettings.GetSection("validAudience").Value,
        }
    });
}
```

```

IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey))
};

}

}

```

بررسی کد :

- ابتدا از فایل appsettings.json مقدار JwtSettings را می‌گیریم و سپس مقدار secretKey را کشی می‌کنیم.
- صدا زدن متد AddAuthentication برای رجیستر کردن JWT Middleware است.
- برای Authentication و JwtBearerDefaults.AuthenticationScheme باید AuthenticationChallengeScheme را مشخص کنیم.

در اینجا برخی پارامترهایی که هنگام تأیید JWT مورد نیاز هست آورده شده است.

- issuer سروری است که رمز را ایجاد کرده است. (ValidateIssuer = true)
- گیرنده رمز یک گیرنده معتبر است. (ValidateAudience=true)
- توکن منقضی نشده است. (ValidateLifetime = true)
- signing key معتبر و مورد اعتماد سرور است. (ValidateIssuerSigningKey=true)
- علاوه بر این، مقادیر audience، issuer و secret key که سرور برای تولید Signature استفاده می‌کند را ارائه می‌دهیم.

حالا باید این اکستنشن متد را در متد ConfigureServices صدا بزنیم.

```

services.ConfigureIdentity();
services.ConfigureJWT(Configuration);

```

سپس برای محافظت از API، CompanyController را باز کرده و یک اتریبوت Authorize در بالای اکشن متد GetCompaniesAsync اضافه کنید.

```

[HttpGet(Name = "GetCompanies"), Authorize]
public async Task<IActionResult> GetCompaniesAsync()
{

```

```

var companies = await
_repository.Company.GetAllCompaniesAsync(trackChanges:
false);

return Ok(companies);
}

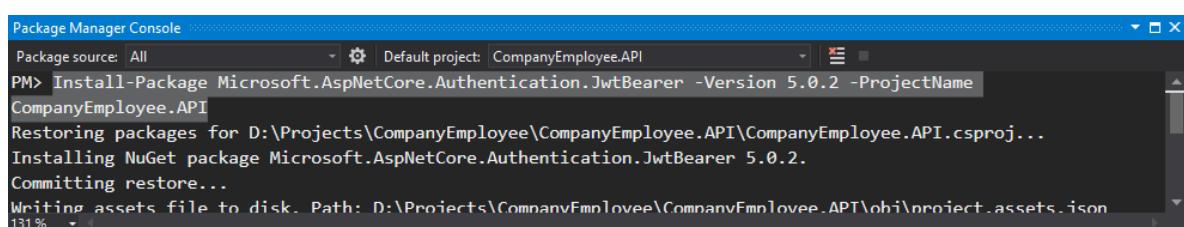
```

فراموش نکنید که CompanyController Namespace پایین را در اضافه نمایید.

```
using Microsoft.AspNetCore.Authorization;
```

قبل از تست، باید پکیج پایین را نصب کنید.

```
Install-Package Microsoft.AspNetCore.Authentication.JwtBearer -Version
5.0.2 -ProjectName CompanyEmployee.API
```



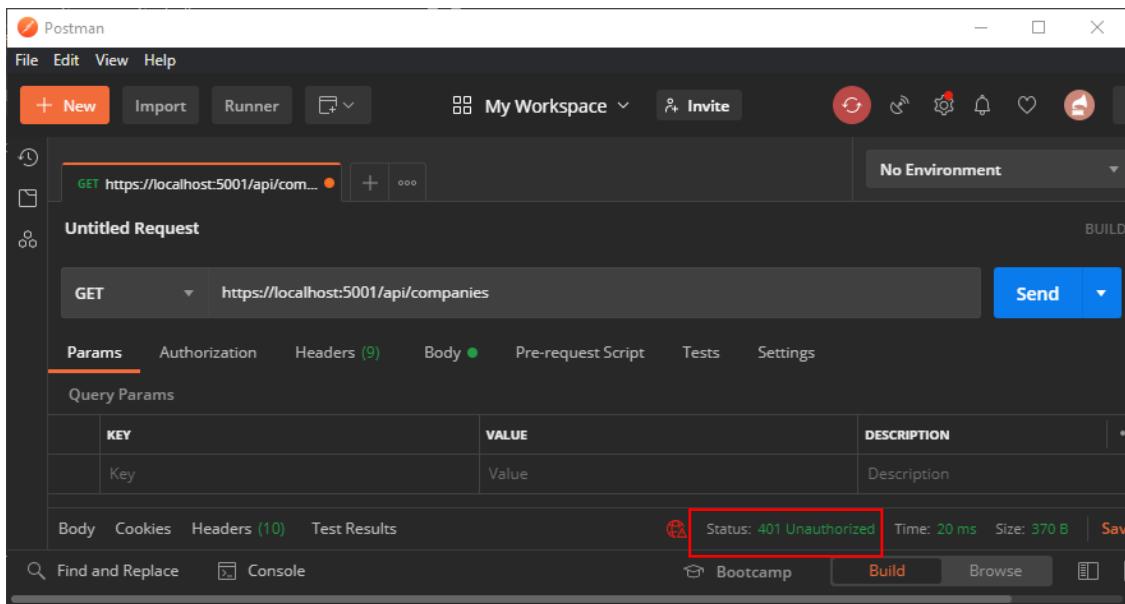
برای تست JWT ریکوئست پایین را به اپلیکیشن ارسال کنید.

<https://localhost:5001/api/companies>

نکته!!

شاید بعد از اجرای این ریکوئست به اکسپشن برخورد کنید. نگران نباشد یکبار ویژوال استدیو را ببندید و دوباره باز کنید.

حالا می توانید ریکوئست را اجرا کنید.



- همانطور که می‌بینید پیام 401 Unauthorized دریافت کردیم چون یک کاربر غیر مجاز می‌خواهد به API دسترسی داشته باشد.

خب ما باید کاری کنیم که کاربر، Authenticate شود و یک رمز معتبر داشته باشد.

پیاده‌سازی Authentication

اولین گام برای پیاده‌سازی Authentication، داشتن یک کلاس برای نگهداری UserName و Password است. بنابراین در فolder DataTransferObjects یک کلاس با نام UserForAuthenticationDto ایجاد کنید.

```
using System.ComponentModel.DataAnnotations;

namespace Entities.DataTransferObjects
{
    public class UserForAuthenticationDto
    {
        [Required(ErrorMessage = "User name is required")]
        public string UserName { get; set; }

        [Required(ErrorMessage = "Password name is required")]
        public string Password { get; set; }
    }
}
```

می خواهیم منطق پیچیده‌ای برای Authentication و تولید Token داشته باشیم. بنابراین بهتر است این اکشن‌ها را در سرویس دیگری واکشی کنید.

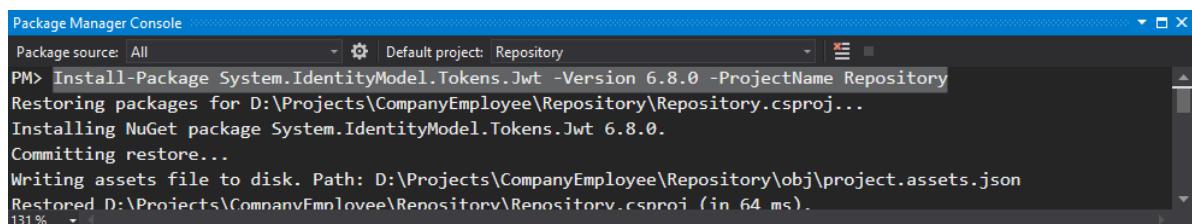
پس در مسیر Contracts/IServices یک اینترفیس جدید با نام IAuthenticationManager ایجاد نمایید.

```
using Entities.DataTransferObjects;
using System.Threading.Tasks;

namespace Contracts.IServices
{
    public interface IAuthenticationManager
    {
        Task<bool> ValidateUserAsync(UserForAuthenticationDto
            userForAuth);
        Task<string> CreateTokenAsync();
    }
}
```

حالا نیاز است تا پکیج پایین را در پروژه Repository نصب کنید.

```
Install-Package System.IdentityModel.Tokens.Jwt -Version 6.8.0 -  
ProjectName Repository
```



در مرحله بعد باید در پروژه Repository کلاس AuthenticationManager را برای پیاده‌سازی این اینترفیس ایجاد نمایید.

```
using Contracts.IServices;
using Entities.DataTransferObjects;
using Entities.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
```

```
using Microsoft.IdentityModel.Tokens;
using System;
using System.Collections.Generic;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using System.Threading.Tasks;

namespace Repository
{
    public class AuthenticationManager : IAuthenticationManager
    {
        private readonly UserManager<User> _userManager;
        private readonly IConfiguration _configuration;
        private User _user;
        public AuthenticationManager(UserManager<User> userManager,
            IConfiguration configuration)
        {
            _userManager = userManager;
            _configuration = configuration;
        }
        public async Task<bool>
        ValidateUserAsync(UserForAuthenticationDto userForAuth)
        {
            _user = await
            _userManager.FindByNameAsync(userForAuth.UserName);

            return (_user != null && await
            _userManager.CheckPasswordAsync(_user,
            userForAuth.Password));
        }

        public async Task<string> CreateTokenAsync()
        {
            var signingCredentials = GetSigningCredentials();
            var claims = await GetClaims();
        }
    }
}
```

```

        var tokenOptions = GenerateTokenOptions(signingCredentials,
claims);

        return new JwtSecurityTokenHandler().WriteToken(tokenOptions);
    }

private SigningCredentials GetSigningCredentials()
{
    var key =
Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable("SEC
RET"));

    var secret = new SymmetricSecurityKey(key);

    return new SigningCredentials(secret,
SecurityAlgorithms.HmacSha256);
}

private async Task<List<Claim>> GetClaims()
{
    var claims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, _user.UserName)
    };
    var roles = await _userManager.GetRolesAsync(_user);

    foreach (var role in roles)
    {
        claims.Add(new Claim(ClaimTypes.Role, role));
    }

    return claims;
}

private JwtSecurityToken GenerateTokenOptions(SigningCredentials
signingCredentials, List<Claim> claims)
{

```

```

        var jwtSettings = _configuration.GetSection("JwtSettings");
        var tokenOptions = new JwtSecurityToken
        (
            issuer: jwtSettings.GetSection("validIssuer").Value,
            audience: jwtSettings.GetSection("validAudience").Value,
            claims: claims,
            expires:

            DateTime.Now.AddMinutes(Convert.ToDouble(jwtSettings.GetSection("expires").Value)),
            signingCredentials: signingCredentials
        );

        return tokenOptions;
    }
}
}

```

بررسی کد :

- در متدهای ValidateUser و FindByNameAsync کاربر در دیتابیس وجود دارد و آیا رمز ورود با رمز دیتابیس مطابق است.
- برای یافتن کاربر از متدهای UserManager<User>.FindByNameAsync و برای چک کردن پسورد، از متدهای FindByNameAsync استفاده کردیم.
- متدهای CreateToken و GenerateTokenOptions با جمع آوری اطلاعاتی از متدهای GetClaims و GetSignInCredentials) و WriteToken، یک توکن ایجاد می‌کند.
- متد GetSignInCredentials یک Secret key را برمی‌گرداند.
- متد GetClaims لیستی از Claims و Roles را که کاربر به آن تعلق دارد را برمی‌گرداند.
- متد GenerateTokenOptions یک شی با تمام گزینه‌های مورد نیاز را، ایجاد و برمی‌گرداند. توجه داشته باشید که پارامترهای JwtSecurityToken باید

از فایل appsettings.json واکشی شود بنابراین باید پارامتر expires را هم به این فایل اضافه کنید.

```
{  
  "ConnectionStrings": {  
    "sqlConnection": "server=.; database=CompanyEmployee; Integrated  
    Security=true"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    }  
  },  
  
  "JwtSettings": {  
    "validIssuer": "MicrodevAPI",  
    "validAudience": "https://localhost:5001",  
    "expires": 5  
  },  
  "AllowedHosts": "*"  
}
```

حالا باید این کلاس را در متده ConfigureServices رجیستر کنیم.

```
services.AddScoped<IAuthenticationManager, AuthenticationManager>();  
حتماً پایین را در کلاس Startup اضافه نمایید.
```

```
using Repository;  
در پایان باید متده AuthenticateAsync را در AuthenticationController اضافه کنیم.
```

```
using AutoMapper;  
using CompanyEmployee.API.Infrastructure.ActionFilters;  
using Contracts.IServices;  
using Entities.DataTransferObjects;  
using Entities.Models;
```

```

using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace CompanyEmployee.API.Controllers
{
    [Route("api/authentication")]
    [ApiController]
    public class AuthenticationController : ControllerBase
    {
        private readonly ILoggerManager _logger;
        private readonly IMapper _mapper;
        private readonly UserManager<User> _userManager;
        private readonly IAuthenticationManager _authManager;

        public AuthenticationController(ILoggerManager logger, IMapper mapper, UserManager<User> userManager, IAuthenticationManager authManager)
        {
            _logger = logger;
            _mapper = mapper;
            _userManager = userManager;
            _authManager = authManager;
        }

        [HttpPost("login")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> AuthenticateAsync([FromBody] UserForAuthenticationDto user)
        {
            if (!await _authManager.ValidateUserAsync(user))
            {
                _logger.LogWarning($"{nameof(AuthenticateAsync)}: Authentication failed. Wrong user name or password.");
                return Unauthorized();
            }
        }
    }
}

```

```

    }

    return Ok(new { Token = await _authManager.CreateTokenAsync() });
}

[HttpPost]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> RegisterUserAsync([FromBody]
UserForRegistrationDto userForRegistration)
{
    var user = _mapper.Map<User>(userForRegistration);
    var result = await _userManager.CreateAsync(user,
userForRegistration.Password);

    if (!result.Succeeded)
    {
        foreach (var error in result.Errors)
        {
            ModelState.TryAddModelError(error.Code,
error.Description);
        }
    }

    return BadRequest(ModelState);
}

await _userManager.AddToRolesAsync(user,
userForRegistration.Roles);

return StatusCode(201);
}
}
}

```

بررسی کد :

- چیز خاصی در این متده وجود ندارد. اگر اعتبارسنجی انجام نشود ریسپانس 401 Unauthorized را برمی‌گردانیم. در غیر این صورت، توکن ایجاد شده برگردانده می‌شود.

<https://localhost:5001/api/authentication/login>

```

POST https://localhost:5001/api/authentication/login
{
  "username": "ZahraBayat",
  "password": "@sw0r12345"
}
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vCzN0Zm1hcy54bNxzb2FwLm9yZy93cyc8YMDA1LzA1L21kZW50aXR5L2NsYWltcy9uYWI1jo1WmFocFCYX1hdCis1mh0dHA6Ly9zY2h1bmFzLm1pY3Jvc29mdC5jb20vd3MvMjAwOC8Nl9pZGVudGl0es9jbGFpbXcm9sZS16Ik1hbmfzX1i1Cj1eHA1OjE2M0kyNDM2NTAsIm1zcyI6Ik1pY3JvZGV2QVBJ1wiYXk1joiaHR0cHM6Ly9sb2Nhbgvhc3Q6NTAwMSJ9.mP_SaK1OPgehCLIahmYpwJh0zq9npHALD1pGgtQMK8"
}
  
```

همانطور که می‌بینید یک توکن تولید شد. حالا بباید با پسورد نامعتبر امتحان کنیم.

```

POST https://localhost:5001/api/authentication/login
{
  "username": "ZahraBayat",
  "password": "P@ssw0rd"
}
{
  "type": "https://tools.ietf.org/html/rfc7235#section-3.1",
  "title": "Unauthorized",
  "status": 401,
  "traceId": "00-f593d8ad3013424f80206532c5941402-0433f591a9c19b41-00"
}
  
```

یک پیام 401 Unauthorized دریافت می‌کنیم.

اًلن با هر ریکوئستی، باید این توکن ارسال شود و گرنه همچنان پاسخ 401 Unauthorized دریافت خواهیم کرد.

<https://localhost:5001/api/companies>

The screenshot shows the Postman interface with an 'Untitled Request' for a GET request to <https://localhost:5001/api/companies>. The 'Authorization' tab is selected, and the 'Type' dropdown is set to 'Bearer Token'. A warning message states: 'Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables.' Below it, a 'Token' field contains a long, redacted JWT token. The 'Send' button is visible at the top right.

اگر ریکوئست را بفرستیم، باید نتیجه زیر را ببینیم.

The screenshot shows the Postman interface with the same 'Untitled Request' for a GET request to <https://localhost:5001/api/companies>. The 'Headers' tab is selected, showing an 'Authorization' header with the previously copied JWT token and a 'Postman-Token' header. The 'Send' button has been clicked, and the response body is displayed in JSON format:

```

1  [
2   {
3     "id": "838ed3ad-fb62-431d-8bdd-08d891d8177e",
4     "name": "Fara_Ltd",
5     "address": "Tehran, Navab Iran",
6     "country": "Iran",
7     "employees": null
8   },
9   {
10    "id": "f28a58f3-b487-4f2a-8bde-08d891d8177e",
11    "name": "Fara_Ltd",
12    "address": "Tehran, Navab Iran".
13  ]

```

The status bar at the bottom indicates a 200 OK response with a 124 ms time and 2.48 KB size.

اعتبارسنجی براساس Role

در حال حاضر، هر کاربر Authenticate شده می‌تواند به هر اکشنی دسترسی پیدا کند. اگر بخواهیم برخی افراد به برخی اکشن‌های دسترسی نداشته باشند باید از Role افراد کمک بگیریم. به عنوان مثال می‌خواهیم فقط مدیر بتواند اکشن متدهای GetCompaniesAsync را صدابزند.

تنها کاری که باید انجام دهیم اضافه کردن کد پایین به اکشن متدهای GetCompaniesAsync است.

```
[HttpGet(Name = "GetCompanies"), Authorize(Roles = "Manager")]
public async Task<IActionResult> GetCompaniesAsync()
```

برای تست این مورد باید کاربر دیگری با نقش Administrator ایجاد کنید.

<https://localhost:5001/api/authentication>

body

```
{
    "firstname": "Ali",
    "lastname": "Bayat",
    "username": "AliBayat",
    "password": "P@ssw0rd67894",
    "email": "Ali_programmer88@yahoo.com",
    "phonenumber": "0123456789",
    "roles": [
        "Administrator"
    ]
}
```

The screenshot shows the Postman application interface. A POST request is made to `https://localhost:5001/api/authentication`. The request body is a JSON object:

```
1 {
2   "firstname": "Ali",
3   "lastname": "Bayat",
4   "username": "Alibayat",
5   "password": "P@ssw0rd67894",
6   "email": "Ali_programmer88@yahoo.com",
7   "phonenumbers": "0123456789",
8   "roles": [
9     "Administrator"
10  ]
11 }
12 }
```

The response status is 201 Created, time: 263 ms, size: 456 B.

همانطور که می‌بینید 201 گرفتیم پس به سراغ ریکوئست بعدی می‌رویم.

خب حالا باید توکن جدیدی برای این کاربر داشته باشیم.

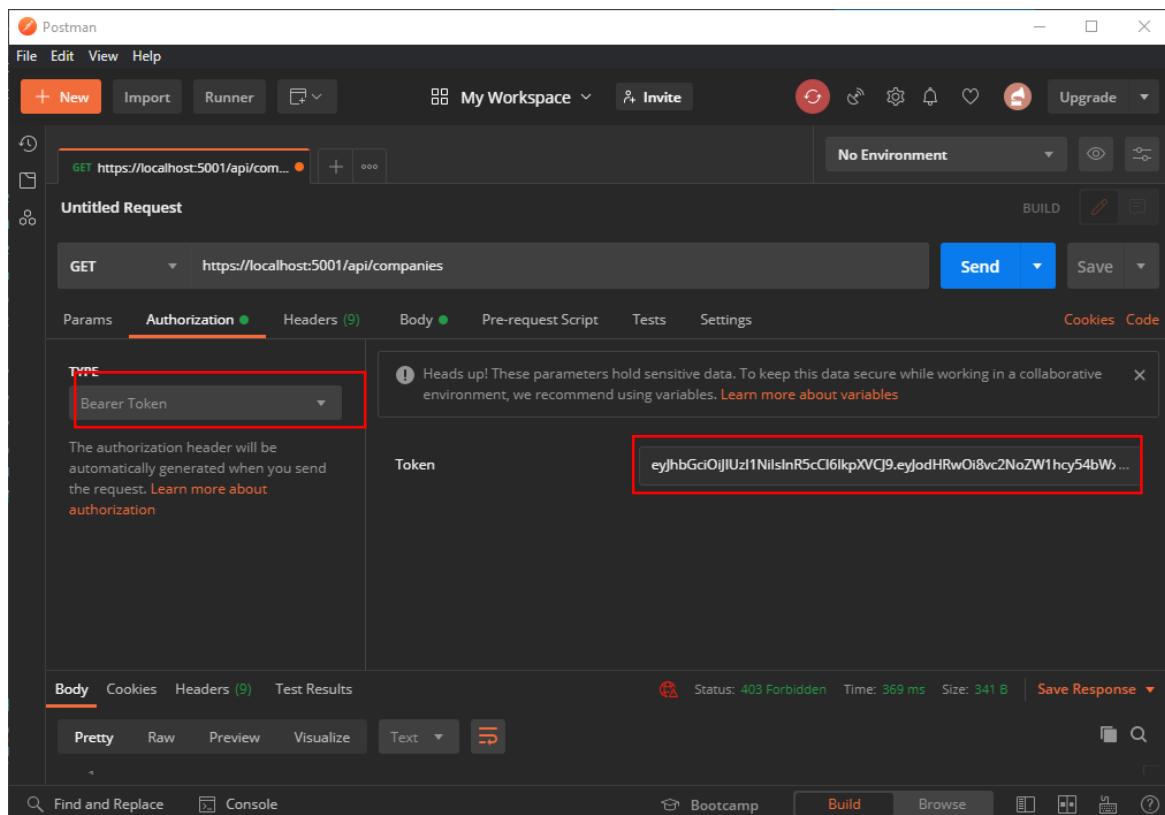
The screenshot shows the Postman application interface. A POST request is made to `https://localhost:5001/api/authentication/login`. The request body is a JSON object:

```
1 {
2   "username": "AliBayat",
3   "password": "P@ssw0rd67894"
4 }
```

The response status is 200 OK, time: 2.14 s, size: 887 B.

حالا این توکن را در ریکوئست بعدی باید استفاده کنیم.

<https://localhost:5001/api/companies>



همانطور که می‌بینید 403 Forbidden گرفتیم چون این کاربر مجاز به دسترسی به این API نیست.

ما می‌توانیم اtribut Authorize را در سطح کنترلر نیز قرار دهیم تا کاربران مجاز، اجازه دسترسی به تمام اکشن‌متد‌های آن کنترلر را داشته باشند.

نکته!!

توکن ما پس از پنج دقیقه از زمان ایجاد، منقضی می‌شود. بنابراین اگر پس از ۵ دقیقه ریکوئست ارسال کنیم مطمئناً وضعیت 401 Unauthorized را دریافت خواهیم کرد.

فصل چهاردهم : ایجاد داکیومنت با Swagger

آنچه خواهید آموخت:

- داکیومنت API چیست؟
- Swagger چیست؟
- تنظیمات Swagger

داتاکیومنت API چیست؟

برنامه‌نویسانی که API را مصرف می‌کنند، نیاز دارند که بدانند چطور باید از آن API استفاده کنند. پس اینجا جایی است که داتاکیومنت API وارد بازی می‌شود.

داتاکیومنت API دستورالعمل‌هایی در مورد نحوه استفاده از API را ارائه می‌دهد بنابراین می‌توان آن را به عنوان یک کتابچه راهنمای، مختصر در نظر گرفت که شامل تمام اطلاعات مورد نیاز برای کار با API است.

داشتن داتاکیومنت مناسب باعث می‌شود تا سایر برنامه‌نویسان بتوانند API‌های ما را با کارهای خود، ادغام و توسعه دهند. همچنین این کار باعث می‌شود تا نگهداری و پشتیبانی از API‌ها، ساده‌تر شود.

Swagger چیست؟

یک ابزار برای توصیف API است که به ما امکان می‌دهد تا بدون بررسی یک سیستم، API‌های آن را ببینیم.

به برنامه‌نویس کمک می‌کند تا با سرعت و دقیق، برای API‌های خود داتاکیومنت تولید کند.

ما می‌توانیم با استفاده از پکیج Swashbuckle، به راحتی Swagger را در پروژه Web API اضافه کنیم.

سه کامپوننت اصلی در پکیج Swashbuckle وجود دارد :

- **Swashbuckle.AspNetCore.Swagger** : این کامپوننت شامل مدل Swagger و کامپوننت شامل است.
- **Swashbuckle.AspNetCore.SwaggerGen** : یک Swagger generator است که Middleware برای نمایش SwaggerDocument را مستقیماً از Controllerها و Modelها می‌سازد.
- **Swashbuckle.AspNetCore.SwaggerUI** : یک ورژن از UI است که Swagger JSON را ترجمه می‌کند. این کامپوننت برای توصیف Web API است.

اگر به متدهای `Configure` و `ConfigureServices` کلاس `Startup` نگاه کنید، حتماً این دو خط دستور را می‌بینید.

: **ConfigureServices** متدهای

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "CompanyEmployee.API",
    Version = "v1" });
});
```

: **Configure** متدهای

```
app.UseSwagger();
app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
    "CompanyEmployee.API v1"));
```

ما در پروژه، `Swagger` را داشتیم اما در این قسمت می‌خواهیم این قابلیت را برای ورژن‌های API سفارشی کنیم.

بنابراین دستورات پایین را در متدهای `Configure` و `ConfigureServices` اضافه کنید.

: **ConfigureServices** متدهای

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "CompanyEmployee.API",
    Version = "v1" });
    c.SwaggerDoc("v2", new OpenApiInfo { Title = "CompanyEmployee.API",
    Version = "v2" });
});
```

: **Configure** متدهای

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
```

```

    c.SwaggerEndpoint("/swagger/v1/swagger.json", "CompanyEmployee.API v1");
    c.SwaggerEndpoint("/swagger/v2/swagger.json", "CompanyEmployee.API v2");
})

```

حالا باید اتریبیوت پایین را در بالای CompaniesController و CompaniesV2Controller قرار دهیم.

: CompaniesController

```

[Route("api/companies")]
[ResponseCache(CacheProfileName = "120SecondsDuration")]
[ApiController]
[ApiExplorerSettings(GroupName = "v1")]
public class CompaniesController : ControllerBase

```

: CompaniesV2Controller

```

[Route("api/companies")]
[ApiController]
[ApiExplorerSettings(GroupName = "v2")]
public class CompaniesV2Controller : ControllerBase

```

با این تغییرات، می‌گوییم که CompaniesController متعلق به گروه v1 و CompaniesV2Controller متعلق به گروه v2 است. باقی کنترلرهای هم در هر دو گروه قرار می‌گیرند زیرا ورژن ندارند.

این تمام کاری است که باید انجام می‌دادیم. حالا اپلیکیشن را اجرا کنید و در یک مرورگر آدرس پایین را وارد کنید.

<https://localhost:5001/Swagger/index.html>

The screenshot shows the Swagger UI interface for the 'CompanyEmployee.API'. At the top, the URL is <https://localhost:7001/swagger/index.html>. A dropdown menu titled 'Select a definition' is open, showing three definitions: 'CompanyEmployee.API v1' (highlighted in green), 'CompanyEmployee.API v1', and 'CompanyEmployee.API v2'. The main content area displays the API documentation for 'CompanyEmployee.API' version V1 OAS3. It includes sections for 'Authentication', 'Companies', and 'Employees'. The 'Companies' section is expanded, showing methods like /api/companies, /api/companies/{id}, and /api/companies/collection.

در این صفحه یک داکیومنت json می‌بینید که شامل تمام کنترلرهای است. همانطور که می‌بینید در بالای این صفحه در قسمت Select a definition، می‌توانید ورژن API را مشخص کنید.

اگر v1 را به v2 تغییر دهید نتیجه پایین را می‌بینید.

The screenshot shows the Swagger UI interface for the 'CompanyEmployee.API' version 2. At the top, there's a dropdown menu labeled 'Select a definition' with 'CompanyEmployee.API v2' selected. Below it, the title 'CompanyEmployee.API' is displayed along with 'v2 OAS3'. The interface is organized into sections: 'Authentication', 'CompaniesV2', and 'Employees'. Under 'CompaniesV2', a red box highlights the 'GET /api/companies' action. Under 'Employees', several actions are listed: 'GET /api/companies/{companyId}/employees', 'POST /api/companies/{companyId}/employees', 'DELETE /api/companies/{companyId}/employees/{id}', 'PUT /api/companies/{companyId}/employees/{id}', and 'PATCH /api/companies/{companyId}/employees/{id}'. There's also a 'Schemas' section at the bottom.

با کلیک بر روی هر اکشن متده، می‌توانیم اطلاعات دقیق پارامترها، ریسپانس و مقادیر مثال را مشاهده کنیم. همچنین با کلیک بر روی دکمه Try it out می‌توان هر یک از اکشن‌های متدها را تست کنید.

This screenshot provides a detailed view of the 'GET /api/companies' action. It includes sections for 'Parameters' (which is currently empty), 'Responses' (listing a 200 status code with a 'Success' description), and 'Links' (which is also empty). A 'Try it out' button is located in the top right corner of the action card.

خب برای تست این API روی دکمه Try it out کلیک و سپس دکمه Execute را بزنید.

The screenshot shows the Microsoft Swashbuckle API Explorer interface. At the top, there's a header 'Companies' and a 'GET /api/companies' button. Below it, a 'Parameters' section says 'No parameters'. To the right, there's a red-bordered 'Cancel' button. In the center, there's a red-bordered 'Execute' button and a 'Clear' button. Under 'Responses', there's a 'Curl' section with the command 'curl -X GET "https://localhost:7001/api/companies" -H "accept: */*' and a 'Request URL' section with 'https://localhost:7001/api/companies'. Below that is a 'Server response' section. It shows a table with two rows: one for '401 Error' (Undocumented) and one for '200 Success'. The '401 Error' row has a red border around its content. The 'Error' column contains '401 Error:' and the 'Details' column contains 'Undocumented'. The '200 Success' row has the 'Code' column as '200' and the 'Description' column as 'Success'. To the right of the '200' row is a 'Links' section with 'No links'. The 'Server response' table also has a red border around its content. It lists various HTTP headers: cache-control: private,max-age=65,must-revalidate; content-length: 0; date: Tue05 Jan 2021 07:29:31 GMT; expires: Tues05 Jan 2021 07:30:36 GMT; server: Kestrel; vary: Accept;Accept-Language;Accept-Encoding; www-authenticate: Bearer; x-rate-limit-limit: 5m; x-rate-limit-remaining: 299; x-rate-limit-reset: 2021-01-05T07:34:31.787Z.

ما

لاگین نکردیم پس خطای 401 می‌گیریم.

خب حالا برای فعال کردن Authorization، باید برخی تغییرات را اعمال کنیم.

چون تعداد خط کدهای Authorization کمی زیاد است و باعث شلوغی متدهای ServiceExtensions می‌شود؛ پس بهتر است در کلاس ConfigureServices یک اکستنشن داشته باشیم.

Namespace پایین را در این کلاس اضافه کنید.

```
using Microsoft.OpenApi.Models;
```

و سپس متد پایین را بنویسید.

```
public static void ConfigureSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(s =>
    {
```

```

s.SwaggerDoc("v1", new OpenApiInfo
{
    Title = "CompanyEmployee.API",
    Version = "v1"
});
s.SwaggerDoc("v2", new OpenApiInfo
{
    Title = "CompanyEmployee.API",
    Version = "v2"
});
s.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
{
    In = ParameterLocation.Header,
    Description = "Place to add JWT with Bearer",
    Name = "Authorization",
    Type = SecuritySchemeType.ApiKey,
    Scheme = "Bearer"
});
s.AddSecurityRequirement(new OpenApiSecurityRequirement()
{
    {
        new OpenApiSecurityScheme
        {
            Reference = new OpenApiReference
            {
                Type = ReferenceType.SecurityScheme,
                Id = "Bearer"
            },
            Name = "Bearer",
            },
        new List<string>()
    }
});
});
}

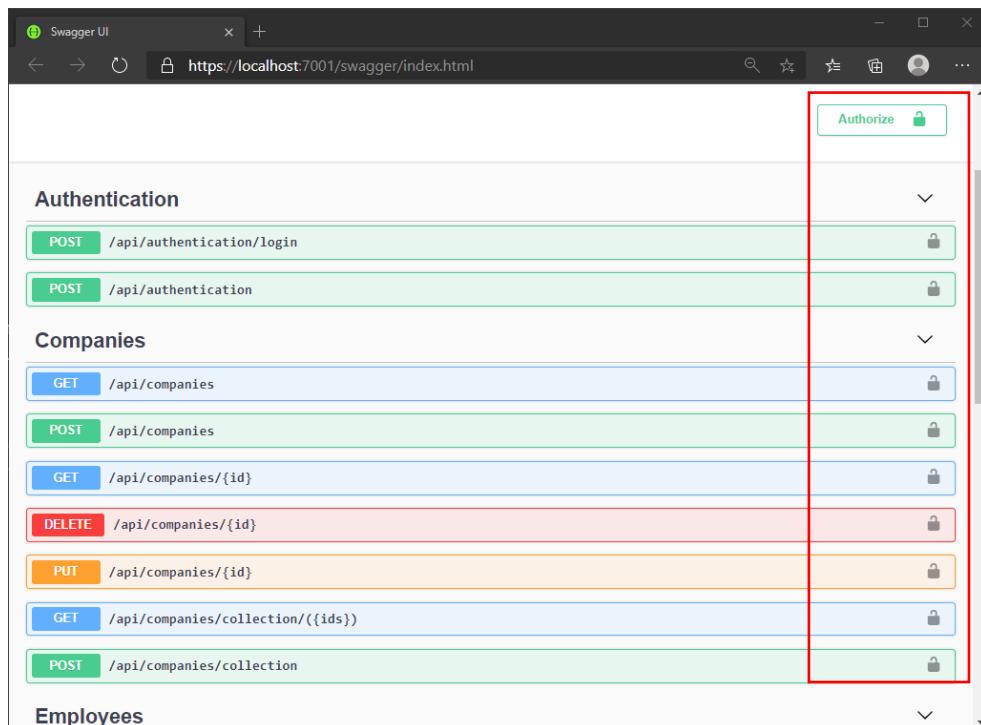
```

خب حالا دستور ConfigureServices را در متده AddSwaggerGen حذف و سپس متده ConfigureSwagger را رजیستر کنید.

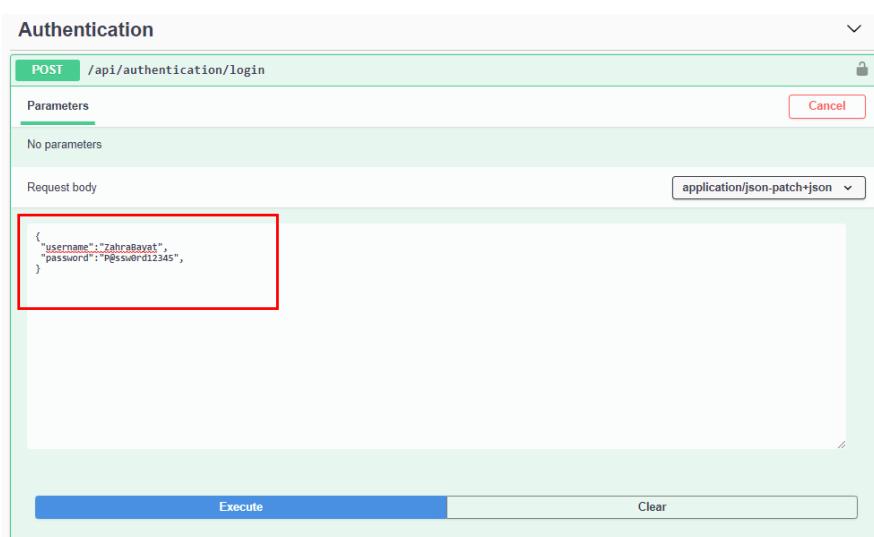
```
services.ConfigureSwagger();
```

برای دیدن تغییرات، اپلیکیشن را دوباره استارت کنید.

اولین چیزی که می بینید اضافه شدن گزینه Authorize به ریکوئست هاست.



برای استفاده از Authorize ابتدا باید توکن بگیریم. پس اکشن api/authentication/login را باز و بعد از وارد کردن User و Password، توکن دریافت شده را کپی کنید.



Curl

```
curl -X POST "https://localhost:7001/api/authentication/login" -H "accept: /*" -H "Content-Type: application/json-patch+json" -d "{ \"username\":\"zahraBayat\", \"password\":\"@ssword12345\" }"
```

Request URL

<https://localhost:7001/api/authentication/login>

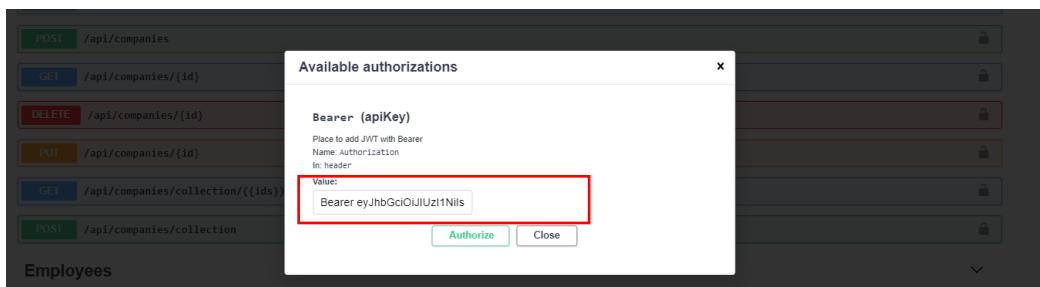
Server response

Code	Details						
200	<p>Response body</p> <pre>{ "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJodHRwOi8vc2NoZHdhcyS4bixzb2FwLbyZy93cY8yMDA1LzA1L21KZh50aXRSL2NSYltcg9uVn1IjojMmFocWFcyXlhCtsinh8HsG1y2zYzh1mfz1mptv3Vc29mIC5jb20d3MWhjwOCBwV1pZGVudG1oeS9jbgFpbXWvc9sZ5161k1hbmFnZXiiLC1eH410jeZMBkANDY5NzksImZcyTgk1nV3v7oG2n0r31xiYXVktjoiaHR0cHMGLy9sb2hhbGhvC3Q6NTAwMSJ9.vVLeq7tsgq51f1dMRAQX1wYbmB0MPS1AdriXvrp034" }</pre> <p>Download</p> <p>Response headers</p> <pre>access-control-allow-origin: * api-supported-versions: 1.0 cache-control: private,max-age=65,must-revalidate content-length: 381 content-type: application/json; charset=utf-8 date: Tue, 05 Jan 2021 11:37:59 GMT etag: "3f21a20807c3040f6ef90c44bf70fd" expires: Tue, 05 Jan 2021 11:39:04 GMT last-modified: Tue, 05 Jan 2021 11:37:59 GMT server: Kestrel vary: AcceptAccept-LanguageAccept-Encoding x-rate-limit-limit: 5m x-rate-limit-remaining: 299 x-rate-limit-reset: 2021-01-05T11:42:55.8865007Z</pre> <p>Responses</p> <table border="1"> <thead> <tr> <th>Code</th> <th>Description</th> <th>Links</th> </tr> </thead> <tbody> <tr> <td>200</td> <td>Success</td> <td>No links</td> </tr> </tbody> </table>	Code	Description	Links	200	Success	No links
Code	Description	Links					
200	Success	No links					

خب حالا روی دکمه Authorization ریکوئست api/companies/ کلیک کنید.

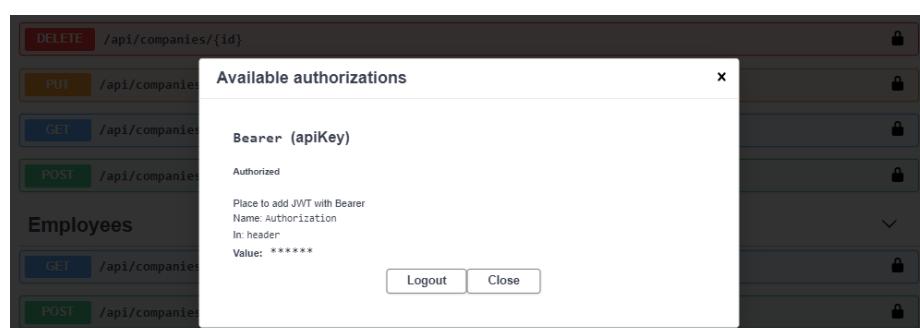


در کادر باز شده این توکن را در مقابل Bearer پیست و روی Authorize کلیک کنید.



توجه داشته باشید که حتما Bearer را در اول توکن بگذارید.

پس از کلیک بر روی دکمه Authorize بر روی دکمه Close کلیک کنید.



حالا می‌توانیم این API را تست کنیم.

The screenshot shows the Swagger UI interface for a GET request to the endpoint `/api/companies`. The request URL is `https://localhost:7001/api/companies`. The response code is 200, and the response body contains the following JSON data:

```
[{"id": "838ed3ad-fd62-431d-80dd-08d891d8177e", "name": "Fara Ltd", "address": "Tehran, Navab Iran", "country": "Iran", "employees": null}, {"id": "f2a5af3f-b487-4f2a-8bde-08d891d8177e", "name": "Fara Ltd", "address": "Tehran, Navab Iran", "country": "Iran", "employees": null}, {"id": "b3a872d0-38b3-4928-95c5-08d891d9719b", "name": "Fara Ltd", "address": "Tehran, Navab Iran", "country": "Iran", "employees": null}, {"id": "9397fd4d-e1f1-405f-f20c-08d891d9cf20", "name": "Fara Ltd", "address": "Tehran, Navab Iran", "country": "Iran", "employees": null}]
```

تنظیمات

گزینه‌هایی برای توسعه داکیومنت و سفارشی کردن UI دارد که می‌خواهم آن را با هم بررسی کنیم.

متدهای AddSwaggerGen و Contact از جمله Description و License می‌توانند اطلاعاتی را ارائه دهد.

```
public static void ConfigureSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(s =>
    {
        s.SwaggerDoc("v1", new OpenApiInfo
        {
            Title = "CompanyEmployee.API",
            Version = "v1",
            Description = "CompanyEmployees API by Zahra Bayat",
            Contact = new OpenApiContact
            {
                Name = "Zahra Bayat",
                Email = "BytZahra@gmail.com",
            }
        });
    });
}
```

```

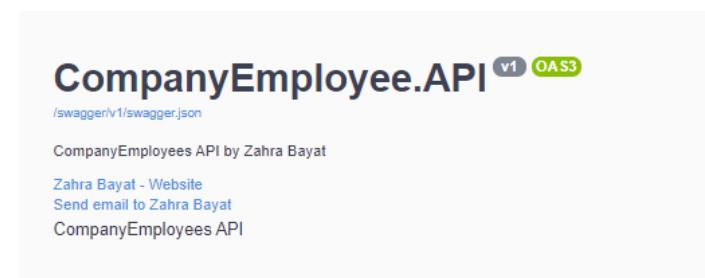
        Url = new
        Uri("https://www.linkedin.com/in/zahrabayat"),
    },
    License = new OpenApiLicense
    {
        Name = "CompanyEmployees API",
    }
});

s.SwaggerDoc("v2", new OpenApiInfo
{
    Title = "CompanyEmployee.API",
    Version = "v2"
});
s.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
{
    In = ParameterLocation.Header,
    Description = "Place to add JWT with Bearer",
    Name = "Authorization",
    Type = SecuritySchemeType.ApiKey,
    Scheme = "Bearer"
});
s.AddSecurityRequirement(new OpenApiSecurityRequirement()
{
    {
        new OpenApiSecurityScheme
        {
            Reference = new OpenApiReference
            {
                Type = ReferenceType.SecurityScheme,
                Id = "Bearer"
            },
            Name = "Bearer",
        },
        new List<string>()
    }
});

```

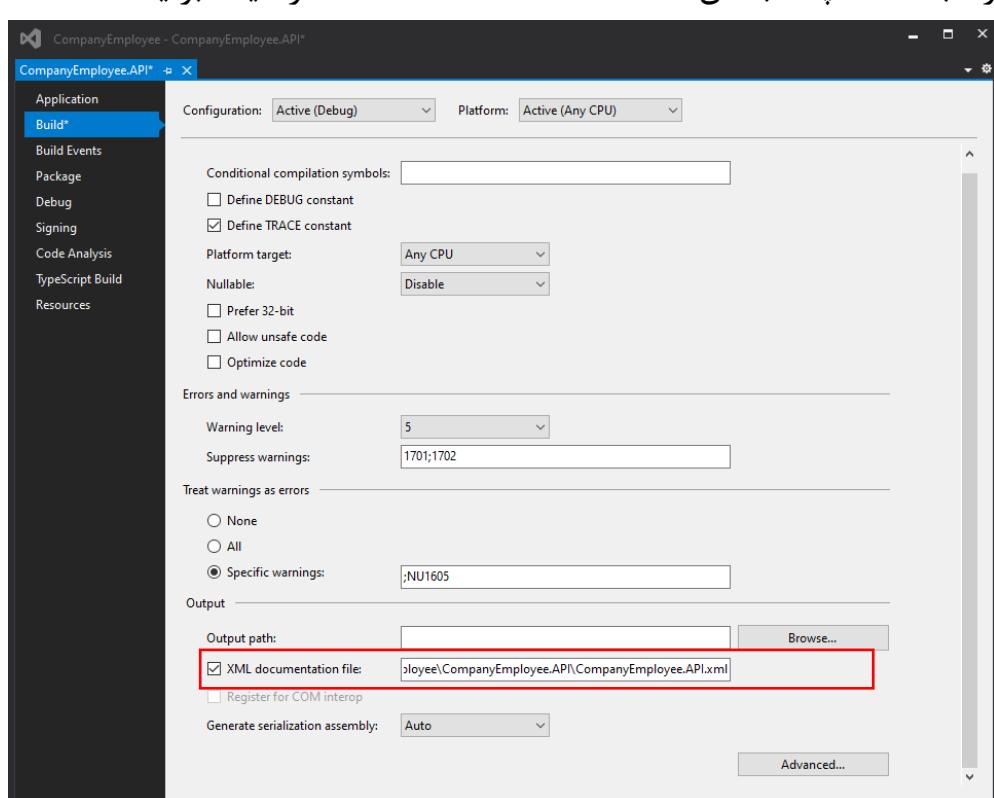
```
}();  
}
```

حالا باید برنامه را یک بار دیگر اجرا کنیم و UI Swagger را بررسی کنیم.



برای فعال کردن XML comment ها باید مراحل زیر را انجام دهیم.

- بر روی پروژه اصلی راست کلیک کنید و از منو باز شده گزینه‌ی Properties را انتخاب نمایید.
- در تب Build چک باکس XML documentation file را تیک بزنید.



حالا باید تغییراتی در متدهای ConfigureSwagger در متدهای ServiceExtensions اضافه کنید.
using System.IO;

```
using System.Reflection;
```

: ConfigureSwagger متد

```
public static void ConfigureSwagger(this IServiceCollection services)
{
    services.AddSwaggerGen(s =>
    {
        s.SwaggerDoc("v1", new OpenApiInfo
        {
            Title = "CompanyEmployee.API",
            Version = "v1",
            Description = "CompanyEmployees API by Zahra Bayat",
            Contact = new OpenApiContact
            {
                Name = "Zahra Bayat",
                Email = "BytZahra@gmail.com",
                Url = new Uri("https://www.linkedin.com/in/zahrabayat"),
            },
            License = new OpenApiLicense
            {
                Name = "CompanyEmployees API ",
            }
        });
        s.SwaggerDoc("v2", new OpenApiInfo
        {
            Title = "CompanyEmployee.API",
            Version = "v2"
        });
        var xmlFile =
            $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(ApplicationContext.BaseDirectory, xmlFile);
        s.IncludeXmlComments(xmlPath);
        s.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
        {
            In = ParameterLocation.Header,
```

```

        Description = "Place to add JWT with Bearer",
        Name = "Authorization",
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });
    s.AddSecurityRequirement(new OpenApiSecurityRequirement()
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                },
                Name = "Bearer",
            },
            new List<string>()
        }
    });
});
}

```

- در مرحله بعد GetCompaniesAsync را به اکشن متده triple-slash comment اضافه کنید، تا توضیحاتی به Swagger UI اضافه شود.

```

/// <summary>
/// Gets the list of all companies
/// </summary>
/// <returns>The companies list</returns>
[HttpGet(Name = "GetCompanies"), Authorize(Roles = "Manager")]
public async Task<IActionResult> GetCompaniesAysnc()

```

حالا برای دیدن نتیجه، اپلیکیشن را اجرا کنید.

Companies

GET	/api/companies Gets the list of all companies
-----	---

معمولًاً برنامه نویسانی که API‌های ما را استفاده می‌کنند به نتیجه‌ای که Return می‌شود، مخصوصاً انواع ریسپانس و کدهای خطا، علاقه‌مند هستند. بنابراین توصیف انواع ریسپانس‌ها بسیار مهم است.

```

/// <summary>
/// Creates a newly created company
/// </summary>
/// <param name="company"></param>
/// <returns>A newly created company</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
/// <response code="422">If the model is invalid</response>
[HttpPost]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(422)]

[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult>
CreateCompanyAsync([FromBody]CompanyForCreationDto company)

```

Responses	
Code	Description
201	Returns the newly created item
400	If the item is null
	Media type text/plain
	Example Value Schema
	{ "type": "string", "title": "string", "status": 0, "detail": "string", "instance": "string" }
422	If the model is invalid

کتاب‌های نوشته شده:

