

تاریخ شروع: ۲/۹/۹۶

تاریخ پایان: ۲/۱۰/۹۶

گردآوری: سیروان عفیفی

Contents

[Building Your First ASP.NET Core Application 5](#_Toc954597)

[Introduction 5](#_Toc954598)

[Setup 5](#_Toc954599)

[A New Project 6](#_Toc954600)

[Command Line Tools and Code 9](#_Toc954601)

[The ASP.NET Core Project Structure 11](#_Toc954602)

[Creating a Greeting Service 14](#_Toc954603)

[Startup and Middleware 17](#_Toc954604)

[Introduction 17](#_Toc954605)

[How Middleware Works 17](#_Toc954606)

[Using IApplicationBuilder 18](#_Toc954607)

[Showing Exception Details 20](#_Toc954608)

[Middleware to Match the Environment 22](#_Toc954609)

[Serving Files 25](#_Toc954610)

[Setting up ASP.NET MVC Middleware 28](#_Toc954611)

[Summary 30](#_Toc954612)

[Controllers in the MVC Framework 31](#_Toc954613)

[The Model View Controller Design Pattern 31](#_Toc954614)

[Routing 32](#_Toc954615)

[Conventional Routes 32](#_Toc954616)

[Attribute Routes 35](#_Toc954617)

[Action Results 37](#_Toc954618)

[Rendering Views 40](#_Toc954619)

[A Table Full of Restaurants 42](#_Toc954620)

[Models in the MVC Framework 45](#_Toc954621)

[Models and View Models 45](#_Toc954622)

[Detail a Restaurant 48](#_Toc954623)

[Create a Restaurant 53](#_Toc954624)

[Accepting Form Input 54](#_Toc954625)

[POST Redirect GET Pattern 55](#_Toc954626)

[Model Validation with Data Annotations 57](#_Toc954627)

[Using the Entity Framework 63](#_Toc954628)

[Introduction 63](#_Toc954629)

[SQL Server LocalDB 63](#_Toc954630)

[Installing the Entity Framework 65](#_Toc954631)

[Implementing a DbContext 67](#_Toc954632)

[Configuring the Entity Framework Services 70](#_Toc954633)

[Entity Framework Migrations 71](#_Toc954634)

[Razor Views 73](#_Toc954635)

[Layout Views 73](#_Toc954636)

[\_ViewStart 75](#_Toc954637)

[\_ViewImports 76](#_Toc954638)

[Razor Pages 77](#_Toc954639)

[An Edit Form 82](#_Toc954640)

[Partial Views 85](#_Toc954641)

[View Components 88](#_Toc954642)

[ASP.NET Identity 90](#_Toc954643)

[Authentication Services and Middlewares 90](#_Toc954644)

[Using the Authorize Attribute 90](#_Toc954645)

[User Registration 91](#_Toc954646)

[Creating a User 95](#_Toc954647)

[Log in and Log Out 97](#_Toc954648)

[Identities and Claims 101](#_Toc954649)

[Front End Frameworks and Tools 102](#_Toc954650)

[Introduction 102](#_Toc954651)

[Front End Tools 102](#_Toc954652)

[Command Line vs. Visual Studio 102](#_Toc954653)

[Setting up npm 103](#_Toc954654)

[Serving File from node\_modules 106](#_Toc954655)

[Enabling Client-side Validation 108](#_Toc954656)

[Using CDNs and Fallbacks 109](#_Toc954657)

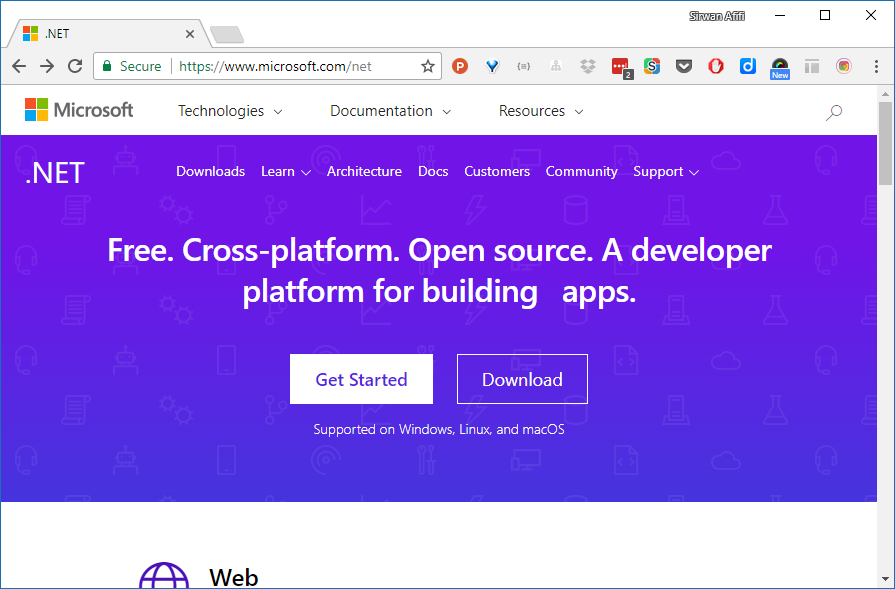
# Building Your First ASP.NET Core Application

## Introduction

در این دوره به ایجاد یک پروژه با استفاده از ASP.NET Core خواهیم پرداخت. این پروژه قرار است اطلاعات یک رستوران را نمایش دهد.

## Setup

برای اولین قدم نیاز است ابتدا مطمئن شویم که .NET Core Framework بر روی سیستم‌مان نصب است. در صفحه microsoft.com/net می‌توانید لیستی از نسخه‌های مختلف .NET Framework را مشاهده نمائید. در صفحه Download باید SDK موردنظرمان را دانلود کنیم؛ SDK حاوی تمامی ابزارهای موردنیاز برای ایجاد یک پروژه .NET می‌باشد:



قدم بعدی انتخاب یک text editor برای نوشتن کدها می‌باشد. در واقع برای پروژه‌های .NET Core از هر ادیتوری می‌توانید استفاده کنید:

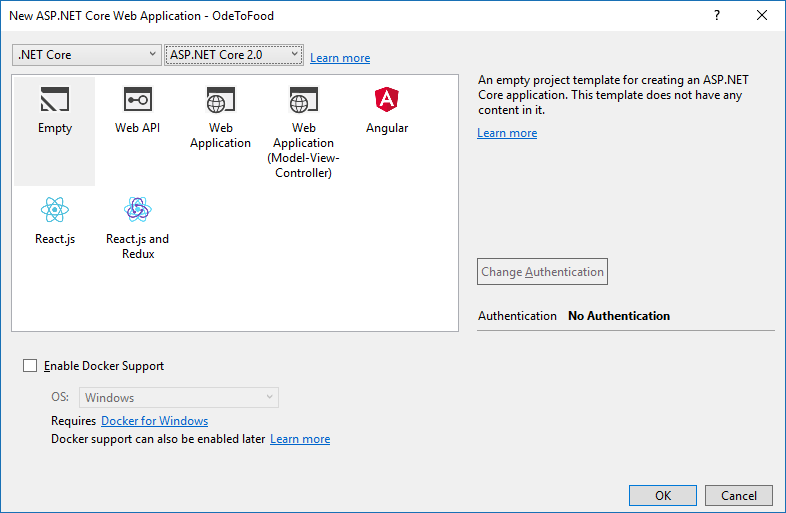


## A New Project

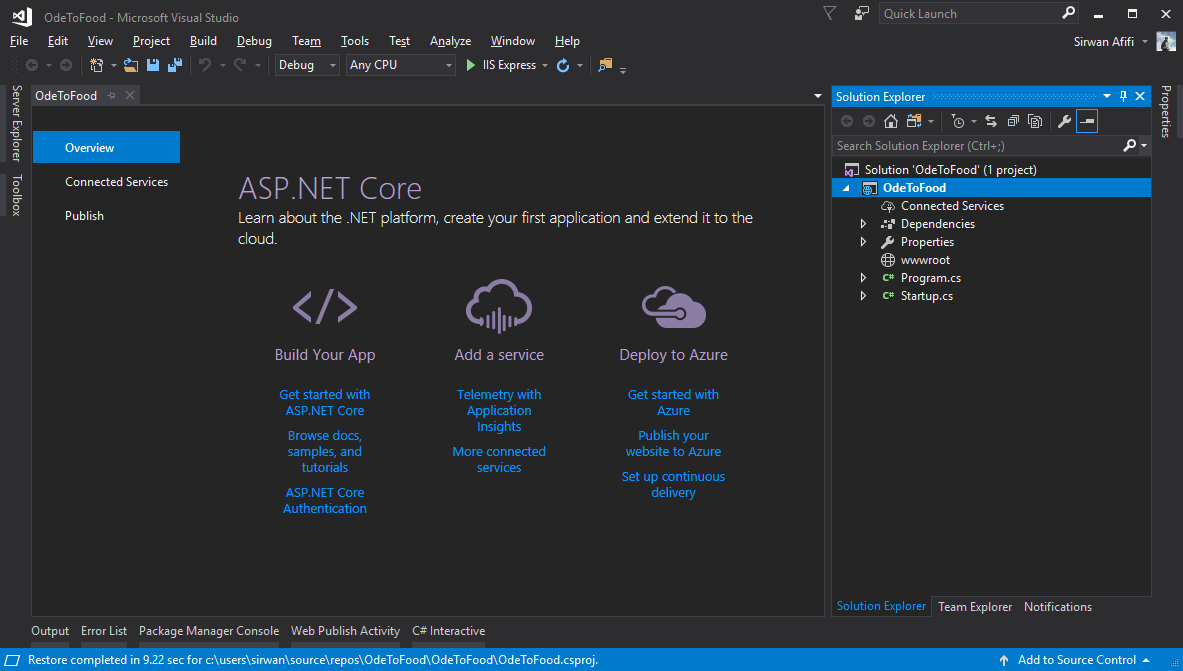
در ادامه یک پروژه جدید ایجاد خواهیم کرد:



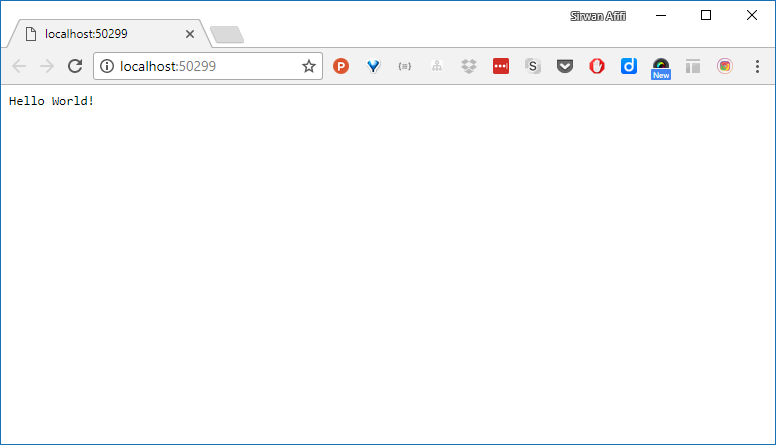
قدم بعدی:



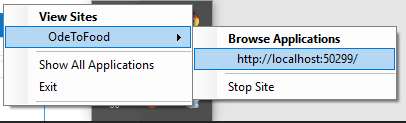
بعد از تائید دیالوگ فوق، پروژه برایمان ایجاد خواهد شد:



اکنون اگر پروژه را اجرا کنیم (Ctrl + F5):

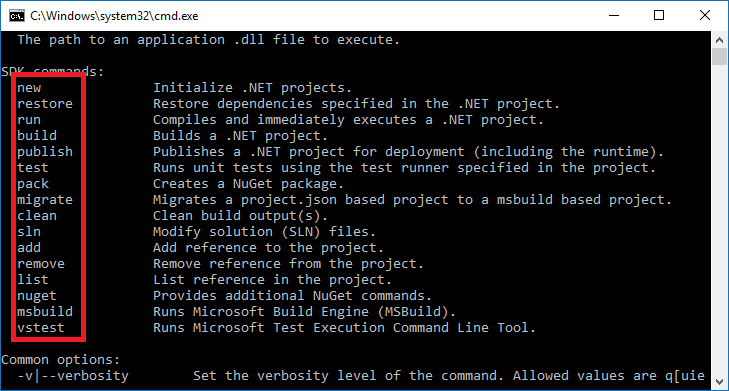


اگر به قسمت System tray توجه کنید آیکن IIS Express را خواهید دید، این در واقع یک Development Web Server است که همراه با Visual Studio عرضه می‌شود. با کلیک بر روی آن خواهید دید که اپلیکیشن‌مان توسط IIS بر روی آدرس localhost به همراه یک پورت تصادفی در حال اجرا می‌باشد:



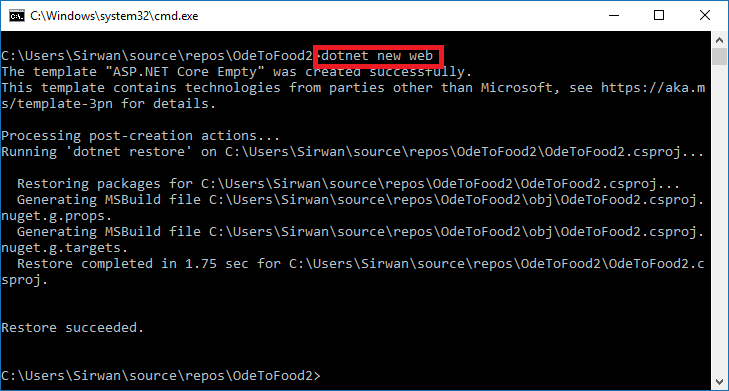
## Command Line Tools and Code

یک روش دیگر ایجاد پروژه از طریق خط فرمان می‌باشد؛ اینکار را می‌توانید توسط dotnet CLI انجام دهید؛ این دستور حاوی پارامترهای زیادی می‌باشد:

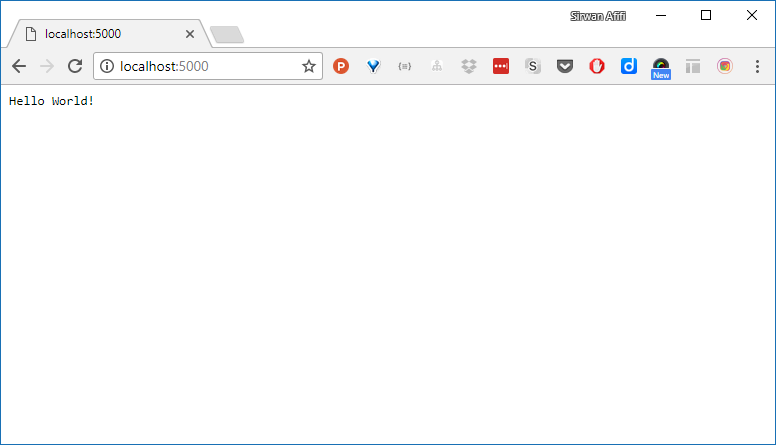


در واقع هر کاری که توسط Visual Studio می‌توان انجام داد را نیز توسط دستور فوق می‌توانیم انجام دهیم؛ که در واقع یک ابزار cross platform می‌باشد.

روش ایجاد پروژه قسمت قبل توسط خط فرمان:



در نهایت توسط دستور dotnet run می‌توانیم پروژه را اجرا کنیم:



توضیح VSCode؛ باز کردن پروژه، نحوه دیباگ و...

## The ASP.NET Core Project Structure

در ادامه می‌خواهیم یکسری Configuration data به اپلیکیشن‌مان اضافه کنیم اما قبل از آن می‌خواهیم محتوای فایل Program.cs را بررسی نمائیم بخصوص متد **BuildWebHost**:

public static IWebHost BuildWebHost(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>()

.Build();

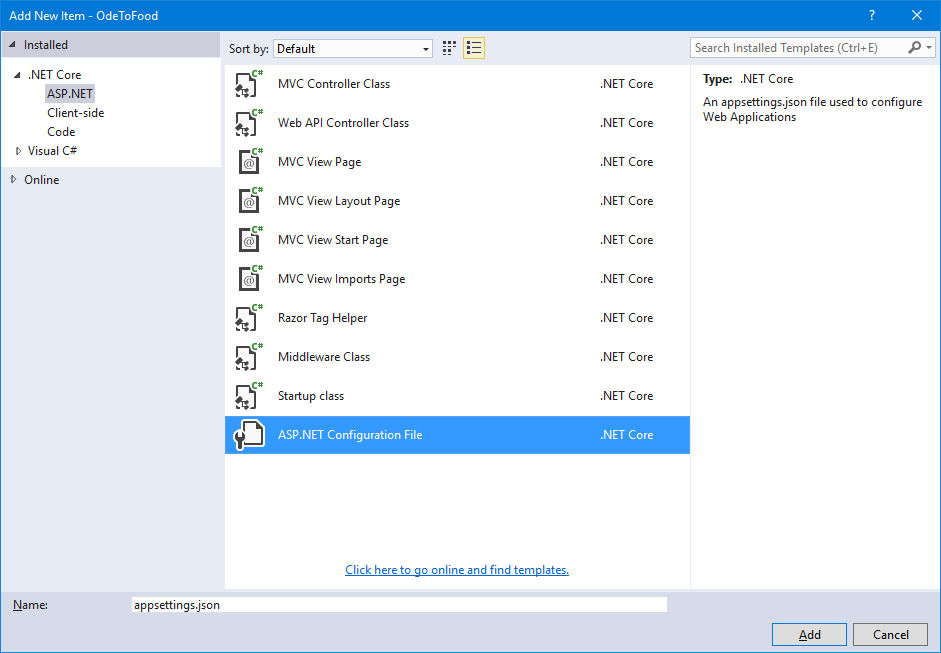
یک WebHostBuilder در واقع شیءی است که می‌داند چطور web server environment را تنظیم کند؛ شیء پیش‌فرض آن محیط موردنیازمان را به یک روش خاص تنظیم خواهد کرد؛ اینها تنظیماتی هستند که ما می‌توانیم توسط نوشتن کدهای بیلدر خودمان تغییر دهیم. اما حالت پیش‌فرض این چنین است:

1. Default web host builder اپلیکیشن را به گونه‌ایی تنظیم خواهد کرد که از Kestrel به عنوان وب‌سرور استفاده کند؛ Kestrel نام وب‌سروری است که به عنوان بخشی از ASP.NET Core عرضه شده است؛ این وب‌سرور به صورت کراس‌پلات‌فرم می‌باشد یعنی بر روی تمامی سیستم‌عامل‌های پشتیبانی شده توسط .NET Core قابل‌اجرا می‌باشد. این همان وب‌سروری است که به Http connections پروسه‌ی ما گوش فرا خواهد داد و همچنین وب‌سروری است که ما از طریق خط فرمان آن را استفاده می‌کنیم.
2. IIS Integration – بیلدر همچنین این بخش را تنظیم خواهد کرد؛ اگر اپلیکیشن‌مان پشت IIS اجرا شود، این همان بخشی است که به IIS اجازه عبور از Windows credentials را خواهد داد.
3. Logging – بیلدر همچنین تعدادی لاگینگ را تنظیم خواهد کرد. وقتی پروژه را اجرا کنید logها را می‌توانید مشاهده نمائید.
4. IConfiguration service را در دسترس قرار خواهد داد – بیلدر پیش‌فرض شیءی که اینترفیس IConfiguration را پیاده‌سازی کرده باشد را وهله‌سازی خواهد کرد. این شیء را می‌توانیم درون اپلیکیشن‌مان در هر جایی استفاده کنیم. کار این سرویس فراهم کردن فایل‌های پیکربندی در اپلیکیشن است، سورس‌های پیش‌فرض این سرویس:
   1. فایلی با نام appsettings.json
   2. User secrets
   3. Environment variables
   4. Command line arguments

خواندن محتوای appsettings.json توسط سرویس IConfiguration:

می‌خواهیم متن hardcode شده‌ی Hello World در مثال قبل را به درون یک فایل Configuration منتقل کرده و از طریق آن این متن را نمایش دهیم. قبلا تنظیمات را درون فایل web.config ذخیره می‌کردیم؛ این فایل در نسخه‌ی ASP.NET Core تنها برای تعیین تنظیمات برای IIS مورداستفاده قرار خواهد گرفت.

در ادامه فایل appsettings.json را ایجاد خواهیم کرد:



درون این فایل می‌توانیم تنظیمات موردنظرمان را اضافه کنیم:

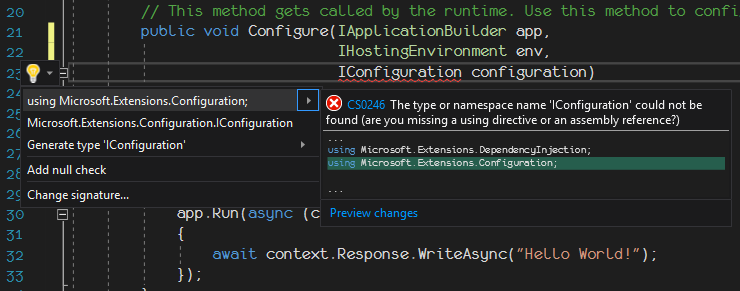
{

"Greeting": "Hello!!",

//....

}

افزودن سرویس عنوان شده:



خواندن تنطمیات توسط سرویس اضافه شده:

app.Run(async (context) =>

{

**var greeting = configuration["Greeting"];**

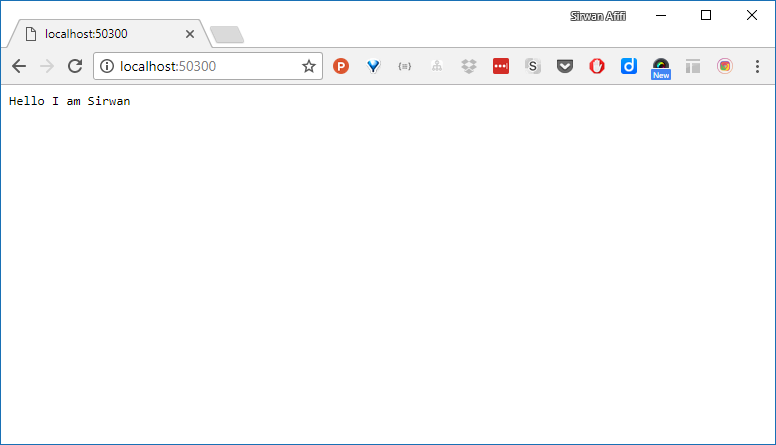
await context.Response.WriteAsync(**greeting**);

});

**نکته:** اگر IConfiguration service کلید مورد جستجو را درون appsettings.json پیدا کند از آن استفاده خواهد کرد در غیراینصورت به دنبال دیگر سورس‌هایی که عنوان شد خواهد رفت. همانطور که عنوان شد آخرین سورس Command line arguments است؛ بنابراین اگر پروژه را اینچنین اجرا کنیم مقدار Greeting از فایل appsettings.json در نظر گرفته نخواهد شد زیرا توسط سورس آخر override شده است:

**dotnet run Greeting="Hello I am Sirwan"**

خروجی:



## Creating a Greeting Service

در واقع نمی‌خواهیم به شیء Configuration به صورت مستقیم درون متد Configure دسترسی داشته باشیم؛ می‌خواهیم اینکار را به یک سرویس محول کنیم.

قدم اول: ایجاد سرویس:

public interface IGreeter

{

string GetMessageOfTheDay();

}

public class Greeter : IGreeter

{

private IConfiguration \_configuration;

public Greeter(IConfiguration configuration)

{

\_configuration = configuration;

}

public string GetMessageOfTheDay()

{

return \_configuration["Greeting"];

}

}

قدم دوم: استفاده از سرویس فوق:

public void Configure(IApplicationBuilder app,

IHostingEnvironment env,

**IGreeter greeter**)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.Run(async (context) =>

{

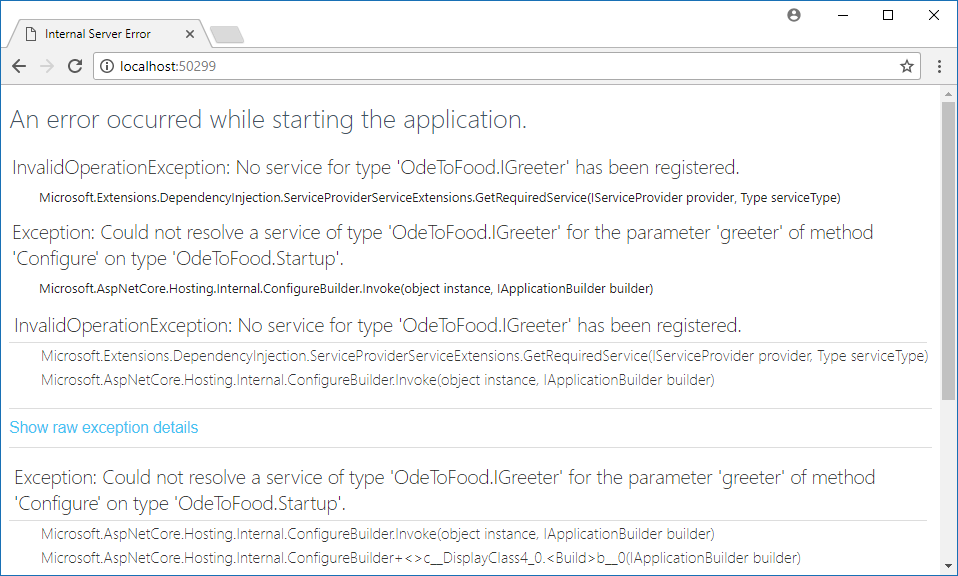
**var greeting = greeter.GetMessageOfTheDay();**

await context.Response.WriteAsync(**greeting**);

});

}

اکنون اگر اپلیکیشن را اجرا کنیم؛ خطای زیر را دریافت خواهیم کرد:



همانطور که مشاهده می‌کنید سرویس IGreeter ریجستر نشده است، برای اینکار باید این سرویس را درون متد ConfigureServices ریجستر کنیم:

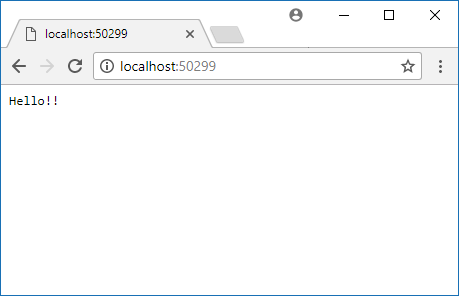
public void ConfigureServices(IServiceCollection services)

{

**services.AddScoped<IGreeter, Greeter>();**

}

خروجی:



طول عمر سرویس‌ها

**AddSingleton** – تنها به یک وهله از سرویس در طول کل حیات اپلیکیشن نیاز خواهیم داشت.

**AddTransient** – هر کسی به وهله‌ایی از سرویس نیاز داشت یک وهله جدید برای او ایجاد کن.

**AddScoped** – به ازای هر درخواست Http یک وهله از سرویس ایجاد کن.

# Startup and Middleware

## Introduction

با استفاده از middleware در ASP.NET می‌توانیم نحوه‌ی پاسخ‌گویی درخواست‌های HTTP را کنترل کنیم.

## How Middleware Works

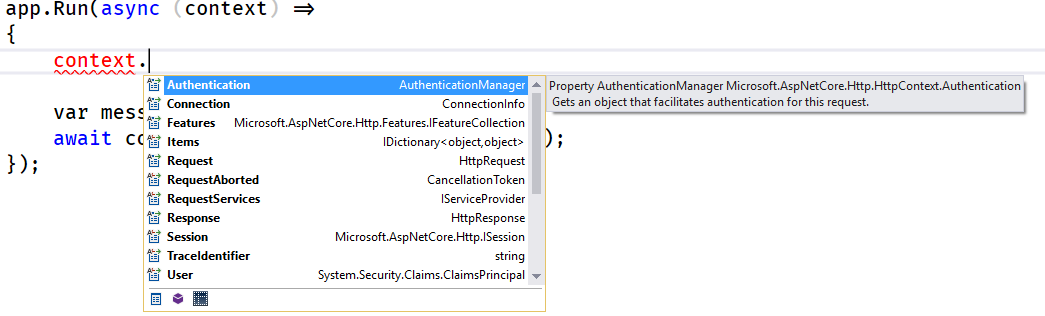
وقتی یک درخواست HTTP به سمت سرور می‌رسد، مثلاً فرض کنید یک درخواست از نوع POST داریم. نرم‌افزار ما باید به نوعی به این درخواست پاسخ دهد. در ASP.NET Core این middleware است که تعیین می‌کند این درخواست چگونه باید پردازش شود. فرض کنید برای هر درخواست می‌خواهیم اطلاعات آن را لاگ کنیم؛ در نتیجه به یک کامپوننت logging نیاز خواهیم داشت؛ این کامپوننت بعد از انجام وظیفه‌ی خودش، درخواست را به middleware بعدی هدایت می‌کند. ممکن است middleware بعدی یک authorizer باشد که دنبال یک کوکی یا اکسس توکن خاص درون درخواست جستجو خواهد کرد؛ اگر این authorizer توکن و یا کوکی را پیدا کند به درخواست اجازه خواهد داد تا به middleware بعدی هدایت شود. Middleware بعدی ممکن است یک روتر باشد.



## Using IApplicationBuilder

برای راه‌اندازی middleware در یک اپلیکیشن ASP.NET Core نیاز به تغییر کدهای درون متد Configure است.

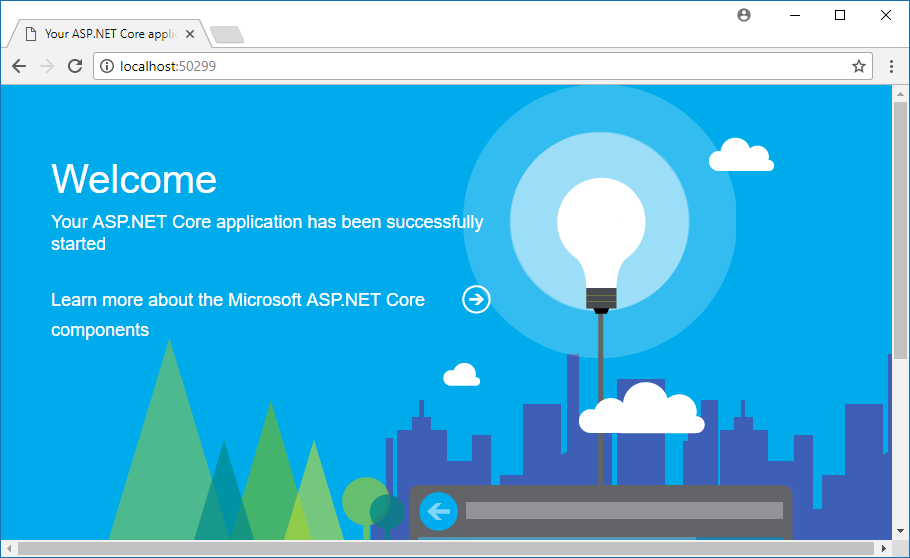
به عنوان مثال توسط متد Run امکان اجرای middlewareهای low-level را خواهیم داشت. درون این متد با استفاده از شیء context به تمامی قسمت‌های یک درخواست دسترسی خواهیم داشت:



بیشتر middlewareها به صورت extension method برای اینترفیس IApplicationBuilder هستند و معمولاً توسط عبارت Use شروع می‌شوند

app.UseWelcomePage();

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



نکته‌ایی که باید به آن دقت داشته باشید؛ اولویت در نصب middlewareها می‌باشد. به عنوان مثال middleware فوق به هر درخواستی با هر آدرسی پاسخ خواهد داد و صفحه فوق را نمایش می‌دهد؛ در اینحالت middlewareهای بعد از آن شانس نمایش داده شدن را نخواهد داشت. به هر حال می‌توانید با پاس دادن یکسری آپشن، مسیر نمایش صفحه فوق را تغییر دهید:

app.UseWelcomePage(**new WelcomePageOptions**

**{**

**Path = "/wp"**

});

اکنون middleware فوق زمانی نمایش داده خواهد شد که آدرس /wp را وارد نمائیم.

توسط app.Run می‌توانیم middlewareهای low-level بنویسیم؛ این متد یک تابع از ورودی دریافت خواهد کرد که ورودی و خروجی این تابع یک RequestDelegate می‌باشد؛ یک RequestDelegate نیز تابعی است که یک HttpContext از ورودی دریافت کرده و در نهایت یک تسک به عنوان خروجی خواهد داشت:

app.Use(next =>

{

return async context =>

{

logger.LogInformation("Request incoming");

if (context.Request.Path.StartsWithSegments("/mym"))

{

await context.Response.WriteAsync("Hit!!");

}

else

{

await next(context);

logger.LogInformation("Response outgoing");

}

};

});

## Showing Exception Details

در ادامه می‌خواهیم میان‌افزار UseDeveloperExceptionPage کامنت کنیم؛ و به صورت عمد استثناء زیر را صادر نمائیم:

app.Run(async (context) =>

{

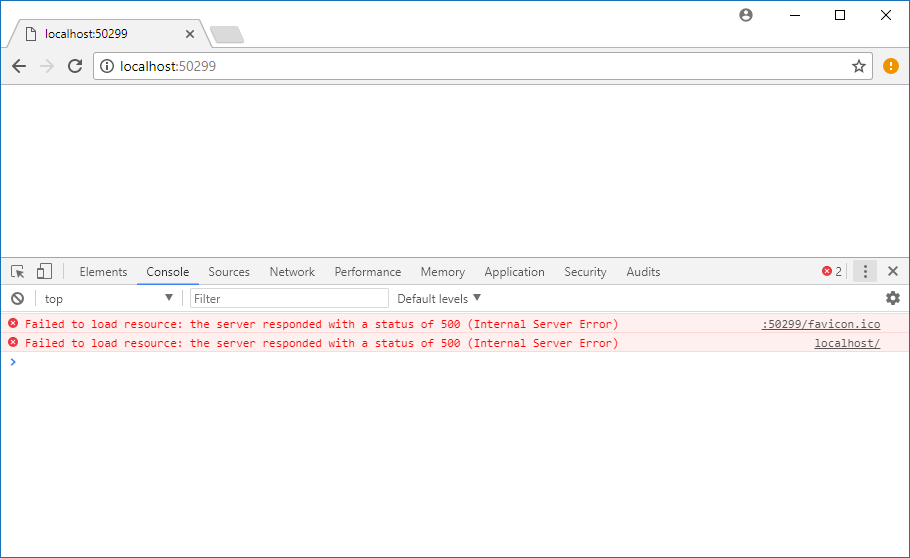
**throw new Exception("error!");**

var greeting = greeter.GetMessageOfTheDay();

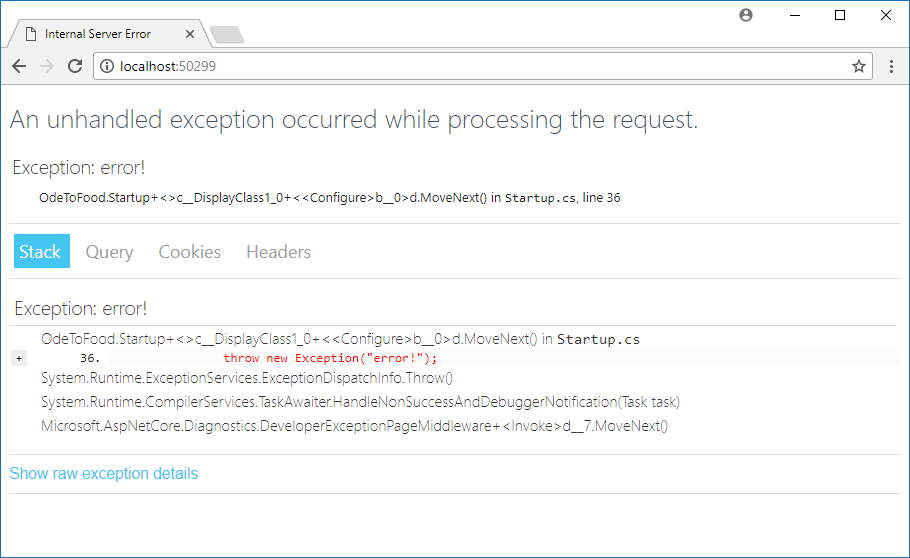
await context.Response.WriteAsync(greeting);

});

به محض بروز یک استثناء در pipeline این خروجی را خواهیم داشت:



اما اگر **UseDeveloperExceptionPage** مجدداً نصب کنیم (کامنت را برداریم) این خروجی را خواهیم داشت:



این middleware اجازه می‌دهد درخواست به middlewareهای بعدی هدایت شود. اگر در جایی دیگر در pipeline خطایی رخ دهد، یک UI همراه با اطلاعات کاملی از وضعیت خطا به کاربر نمایش داده خواهد شد (شکل فوق)

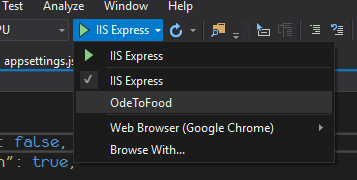
## Middleware to Match the Environment

اینترفیس IHostingEnvironment اطلاعات کاملی را درباره hosting environment در اختیارمان قرار خواهد داد. از طریق این اینترفیس است که نوع environment را تعیین خواهیم کرد.

زمانیکه یک پروژه از نوع ASP.NET Core را ایجاد خواهیم کرد، Visual Studio تعدادی پراپرتی را به پروژه جهت پیکربندی اپلیکیشن برای اجرا در محیط development اضافه خواهد کرد؛ برای ست کردن این پراپرتی‌ها می‌توانیم با مراجعه فایل launchSettings.json اینکار را انجام دهیم:



دیگر ابزارها نیز مانند dotnet CLI تنظیمات درون این فایل را در نظر خواهند گرفت. همانطور که مشاهده می‌کنید درون فایل فوق قسمتی تحت عنوان profiles داریم؛ اگر از ویژوال‌استدیو استفاده شود برای اجرای پروژه پروفایل IIS Express در نظر گرفته خواهد شد. این همان نامی است که در قسمت toolbar موقع اجرای پروژه وجود دارد:



همچنین امکان انتخاب پروفایل OdeToFood نیز وجود دارد؛ این پروفایل نیز همچنین درون فایل launchSettings.json وجود دارد:

"profiles": {

"IIS Express": {

"commandName": "IISExpress",

"launchBrowser": true,

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

**"OdeToFood": {**

**"commandName": "Project",**

**"launchBrowser": true,**

**"environmentVariables": {**

**"ASPNETCORE\_ENVIRONMENT": "Development"**

**},**

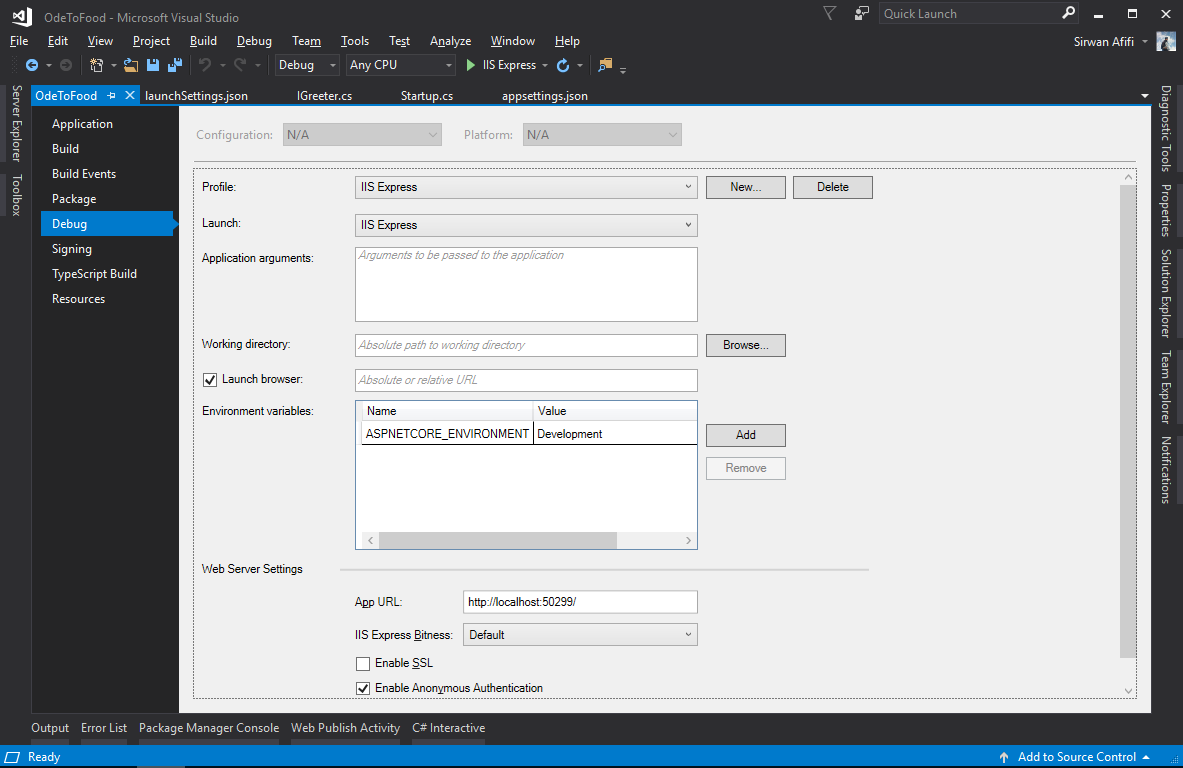
**"applicationUrl": "http://localhost:50300/"**

**}**

}

اما این پروفایل از commandی با نام Project برای اجرای پروژه استفاده خواهد کرد. منظور این است که در پشت‌صحنه از دستور dotnet run برای اجرای پروژه استفاده شود. همانطور که در کد فوق مشاهده می‌کنید برای هر پروفایل‌ امکان تعریف یکسری متغیر محیطی را داریم.

یک روش friendlyتر نیز استفاده از UI زیر جهت تغییر متغیرهای محیطی و همچنین پروفایل جاری پروژه می‌باشد:

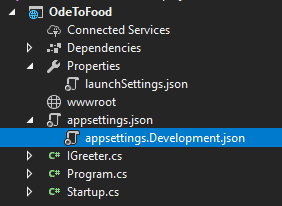


ASPNETCORE\_ENVIRONMENT محیطی است که ASP.NET به صورت پیش‌فرض برای آن جستجو خواهد کرد. اگر مقدار آن برابر با Development باشد، فلاگ IsDevelopment برابر با true خواهد شد. اگر این پراپرتی وجود نداشته باشد ASP.NET فرض را بر این خواهد گذاشت که در محیط Production هستیم.

لازم به ذکر است که امکان داشتن چندین فایل appsettings را درون پروژه داریم؛ در حالت پیش‌فرض اپلیکیشن از فایل appsettings.json استفاده می‌کند؛ اما ASP.NET Core فایل دیگری با این الگوی نامگذاری را جستجو خواهد کرد:

**appsettings.EnvironmentName.json**

با ایجاد فایل این الگو ساختار نمایش آن اینچنین خواهد بود:



مزیت این فایل به عنوان مثال امکان داشتن connection string متفاوت برای محیط‌های متفاوتی می‌باشد.

## Serving Files

یکی از ویژگی‌های مهم هر وب‌اپلیکیشن داشتن قابلیت serve کردن فایل از فایل‌سیستم است.

اگر به پروژه‌ی مثال قبل یک فایل HTML اضافه کنید:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

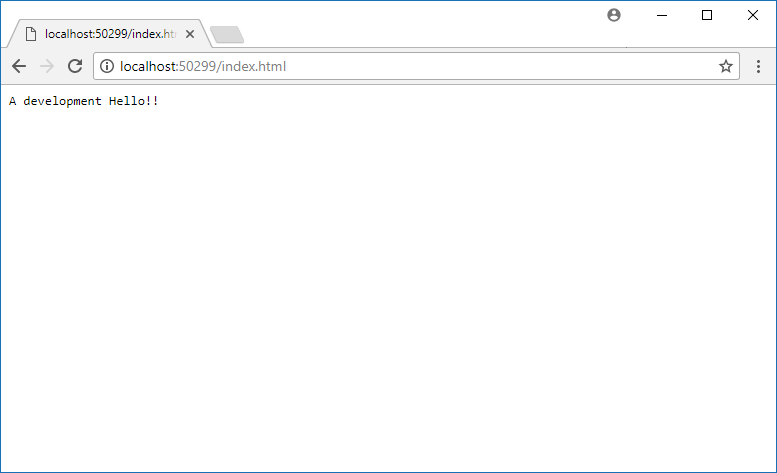
<body>

<h1>This is index.html!</h1>

</body>

</html>

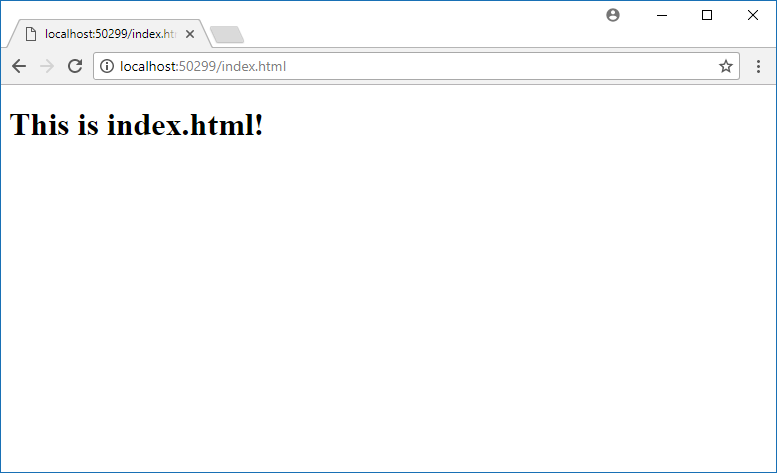
اکنون اگر سعی کنید از طریق مرورگر به آن دسترسی داشته باشید، این امکان را نخواهید داشت. **(لازم به ذکر است که فایل‌های عمومی باید درون پوشه‌ی wwwroot قرار گیرند.)**



دلیل این است middlewareی جهت رسیدگی به درخواست index.html درون pipelineمان هنوز ثبت نشده است. برای نصب این middleware باید پکیج زیر را نصب کنید:

**app.UseStaticFiles();**

خروجی:



در حالت پیش‌فرض StaticFiles به دنبال فایل‌های استاتیک درون پوشه‌ی wwwroot جستجو خواهد کرد. اگر بخواهیم index.html به عنوان پیش‌فرض در هنگام اجرای پروژه‌ اجرا شود باید یک middleware دیگر با نام UseDefaultFiles را نصب کنیم:

**app.UseDefaultFiles();**

**app.UseStaticFiles();**

ترتیب قرار دادن middlewareهای فوق خیلی مهم است، اگر UseDefaultFiles را بعد از UseStaticFiles قرار دهیم، نتیجه مورد انتظار را دریافت نخواهیم کرد زیرا UseDefaultFiles در واقع کار serve کردن فایل‌های استاتیک را انجام نمی‌دهد؛ یک middleware دیگر با نام UseFileServer وجود دارد که کار هر دوی این middlewareها را انجام می‌دهد:

**app.UseFileServer();**

حتی می‌توانیم برای این middleware نیز یک آپشن ارسال کنیم؛ توسط این آپشن می‌توانیم کارهایی مانند فعال‌سازی Directory Browsing را انجام دهیم.

## Setting up ASP.NET MVC Middleware

در نهایت می‌خواهیم یک اپلیکیشن بر فراز ASP.NET ایجاد کنیم؛ به طور دقیق‌تر ایجاد یک اپلیکیشن برفزار ASP.NET MVC؛ در واقع می‌توانیم یک اپلیکیشن را به صورت کامل با استفاده از middlewareها ایجاد کنیم؛ MVC این قابلیت را در اختیارمان خواهد گذاشت تا به سادگی صفحات HTML و همچنین HTTP APIs را ایجاد کنیم. اینکار توسط MVC آسان است زیرا می‌توانیم درخواست‌های وارده را به متدهایی درون کلاس نگاشت کنیم؛ درون این متدها کارهایی مانند کوئری زدن به دیتابیس، خواندن فایل و ... آسان خواهد بود.

در این قسمت می‌خواهیم MVC را نصب کنیم؛ نصب MVC در یک پروژه خالی با سه قدم انجام خواهد شد:



1. نصب پکیج ASP.NET MVC
2. ریجستر کردن سرویس‌های MVC
3. افزودن middleware مربوط به MVC

**قدم اول: نصب پکیج MVC:**

وقتی یک پروژه خالی ASP.NET Core را آغاز کنید، پکیجی تحت عنوان متاپکیج به صورت پیش‌فرض برای پروژه نصب خواهد شد:

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp2.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="**Microsoft.AspNetCore.All**" Version="2.0.3" />

</ItemGroup>

</Project>

همانطور که از نام آن پیداست این پکیج تمامی پکیج‌های ASP.NET Core را تحت یک پکیج با نام **Microsoft.AspNetCore.All** نصب خواهد کرد؛ این پکیج حاوی MVC نیز می‌باشد.

**قدم دوم: ریجستر کردن سرویس‌های MVC:**

app.UseStaticFiles();

**app.UseMvcWithDefaultRoute();**

نکته: اینبار از UseStaticFiles استفاده کرده‌ایم زیرا نمی‌خواهیم نام پیش‌فرضی برای فایل‌های استاتیک در نظر گرفته شود (UseDefaultFiles)؛ می‌خواهیم کاربر خودش فایل موردنظر را درخواست کند. در ادامه کار هندل کردن درخواست را به میان‌افزار UseMvcWithDefaultRoute واگذار کرده‌ایم.

**قدم سوم: افزودن middleware مربوط به MVC:**

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<IGreeter, Greeter>();

**services.AddMvc();**

}

اکنون قبل از اجرای پروژه یک کنترلر ساده به صورت زیر ایجاد کنید:

namespace OdeToFood.Controllers

{

public class HomeController

{

public string Index()

{

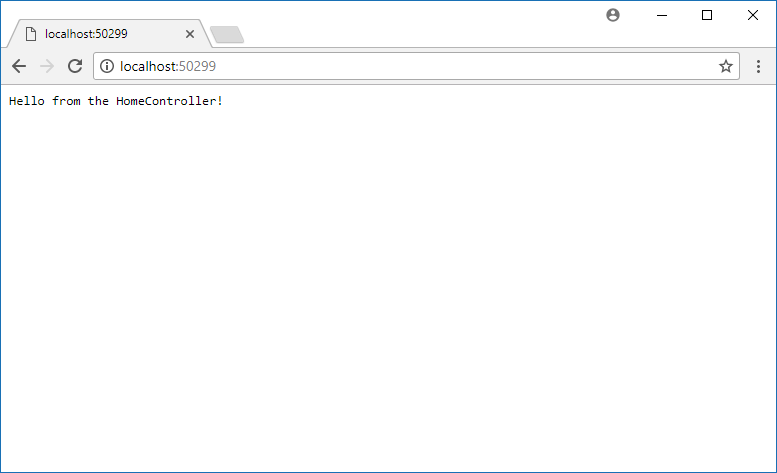
return "Hello from the HomeController!";

}

}

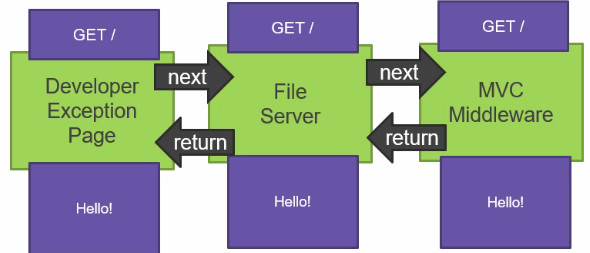
}

خروجی:



## Summary

در این ماژول بیشتر درباره middlewareها و همچنین نحوه‌ی پیکربندی middlewareها درون متد Configure را بررسی کردیم. اکنون ما یک processing pipeline داریم؛ در این حالت یک درخواست از تعدادی middleware عبور خواهد کرد.

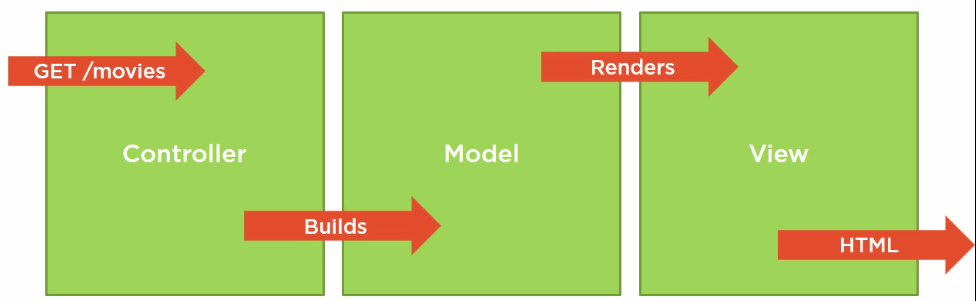


# Controllers in the MVC Framework

## The Model View Controller Design Pattern

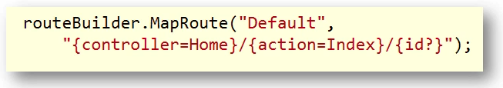
نام فریم‌ورک MVC برگرفته از الگوی طراحی MVC است؛ این الگو یکی از الگوهای محبوب در طراحی UI Layer است. در اپلیکیشن‌هایی با مقیاس بزرگتر، معمولاً MVC را با دیگر الگوهای طراحی ترکیب خواهیم کرد؛ الگوهایی مانند Data Access, Messaging.

در MVC کنترلر یک درخواست HTTP را دریافت خواهد کرد، سپس کنترلر تصمیم خواهد گرفت که چگونه به این درخواست پاسخ دهد؛ اینکار را با populate کردن یک مدل انجام خواهد داد سپس این مدل را به ویو جهت نمایش ارسال خواهد کرد؛ سپس ویو اطلاعات را گرفته و سپس از آنها جهت ساختن یک صفحه‌ HTML استفاده خواهد کرد سپس این HTML به عنوان یک HTTP response به کلاینت ارسال خواهد شد:



## Routing

ASP.NET middlewareی که در ماژول قبلی نصب کردیم، به روشی جهت تعیین اینکه یک درخواست HTTP باید به یک کنترلر جهت پردازش ارسال شود یا نه، نیاز دارد. MVC middleware این تصمیم را براساس URL و یکسری تنظیمات (configuration information) اتخاذ خواهد کرد؛ یک روش برای تعیین این تنظیمات، تعریف routes برای کنترلرها درون فایل **Startup.cs** است؛ به این رهیافت convention based routing گفته می‌شود:



**با این روش به MVC یک تمپلیت را معرفی خواهد کرد که براساس آن URL را بررسی کند.**

روش دیگر استفاده از attribute based routing است:



## Conventional Routes

در قسمت قبل یک کنترلر ایجاد کردیم:

namespace OdeToFood.Controllers

{

public class HomeController

{

public string Index()

{

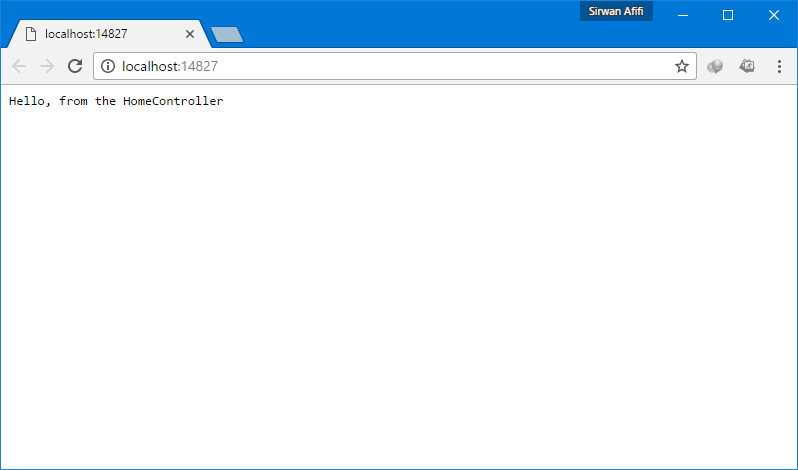
return "Hello from the HomeController!";

}

}

}

همانطور که مشاهده می‌کنید این کنترلر نیازی به ارث‌بری از کلاس پایه‌ی Controller ندارد. در عوض یک کلاس ساده‌ی C# همراه با یک متد درون آن داریم. در اینحالت خروجی زیر را خواهیم داشت:



دلیل این است که مسیریابی پیش‌فرض برای MVC فعال شده است:

app.UseMvcWithDefaultRoute();

برای داشتن مسیریابی سفارشی باید از متد الحاقی **UseMvc** به اینصورت استفاده کنیم:

app.UseMvc(ConfigureRoute);

private void ConfigureRoute(IRouteBuilder routeBuilder)

{

// /Home/Index/4

routeBuilder.MapRoute("Default",

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

}

نکته: در حالت استفاده از متد Run باید نوع MIME Type خروجی را نیز تعیین کنیم. به عنوان مثال در کد زیر تنها یک رشته ساده را به خروجی فرستاده‌ایم:

app.UseMvc();

app.Run(async (context) =>

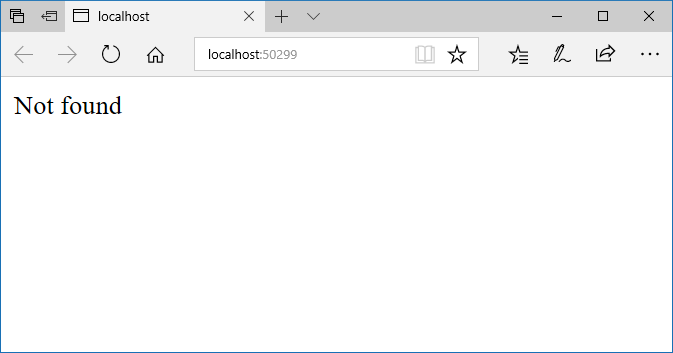
{

var greeting = greeter.GetMessageOfTheDay();

await context.Response.WriteAsync**("Not found"**);

});

در اینحالت مرورگر وب تلاش خواهد کرد نوع محتوایی دریافتی برای مشاهده تشخیص دهد؛ بنابراین چون درون Response Header نمی‌تواند اطلاعاتی پیدا کند آن را به صورت زیر نمایش خواهد داد:



اما اگر MIME Type مناسب را برای خروجی تعیین کنیم؛ مرورگر به راحتی می‌تواند محتوای دریافتی را تشخیص دهد:

app.Run(async (context) =>

{

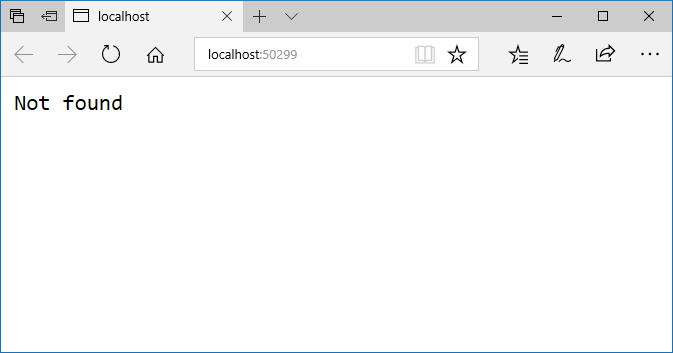
var greeting = greeter.GetMessageOfTheDay();

**context.Response.ContentType = "text/plain";**

await context.Response.WriteAsync("Not found");

});

خروجی:



## Attribute Routes

روش دیگر تعریف route در MVC استفاده از attribute routing است:

using Microsoft.AspNetCore.Mvc;

namespace OdeToFood.Controllers

{

//about

[Route("about")]

public class AboutController

{

[Route("")]

public string Phone()

{

return "09189701078";

}

[Route("address")]

public string Address()

{

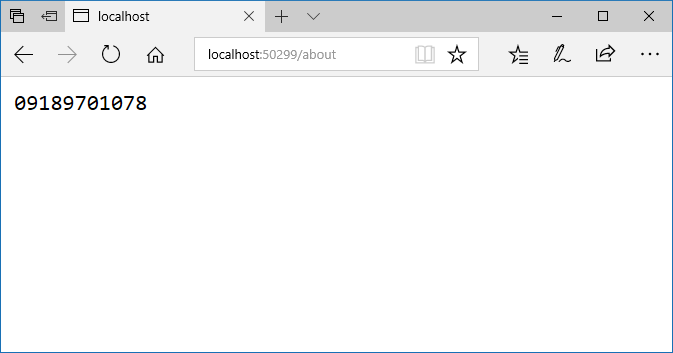
return "Krdistan, Sanandaj";

}

}

}

عدم تعیین template برای Route به معنای دیفالت بودن آن است (همانند Index) - در حالت فوق Phone حالت پیش‌فرض است:



از توکن‌ها نیز می‌توانید استفاده کنید، در اینحالت دیگر نیازی نیست نام کنترلر یا اکشن متد را بنویسید:

using Microsoft.AspNetCore.Mvc;

namespace OdeToFood.Controllers

{

//about

[Route("company/[controller]/[action]")]

public class AboutController

{

public string Phone()

{

return "09189701078";

}

public string Address()

{

return "Krdistan, Sanandaj";

}

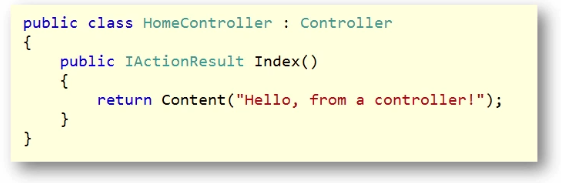
}

}

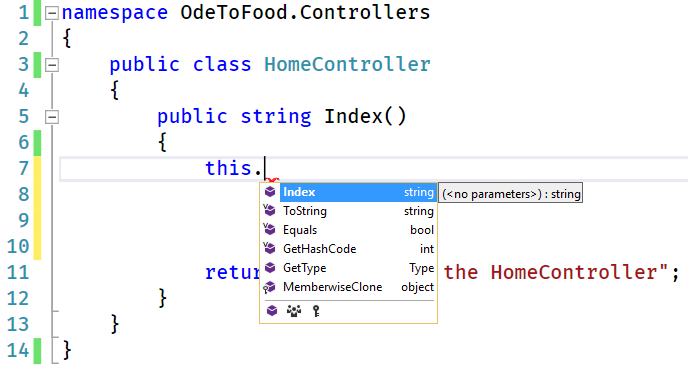
مزیت توکن این است که اگر بعداً نام کنترلر را تغییر دهیم؛ خود فریم‌ورک آن را تشخیص خواهد داد زیرا به صورت صریح نام about را ننوشته‌ایم.

## Action Results

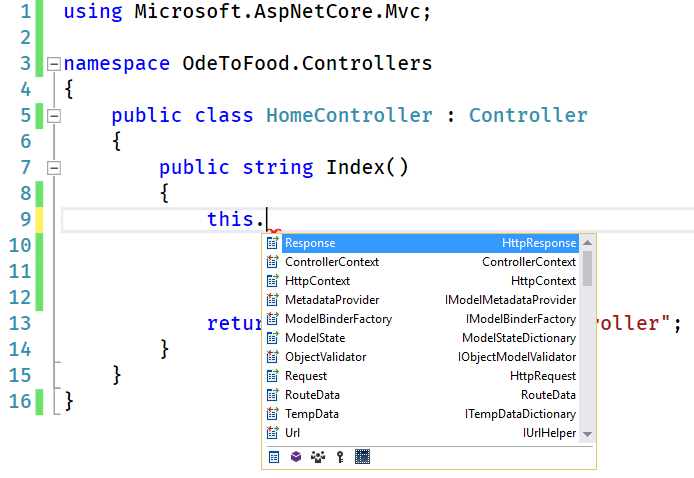
تا اینجا از Plain simple C# classes به عنوان کنترلر استفاده کردیم؛ در MVC مرسوم است که از کلاس‌ پایه‌ی Controller ارث‌بری کنیم همچنین خروجی اکشن متدها از نوع IActionResult باشند:



ارث‌بری از کلاس Controller قابلیت‌های زیادی را درون کنترلر در اختیارمان قرار خواهد داد؛ در حالت عدم ارث‌بری از این کلاس درون یک کنترلر خواهیم داشت:



اما اگر از Controller ارث‌بری کنیم:



همانطور که مشاهده می‌کنید در اینحالت به contextual propertyهای زیادی دسترسی خواهیم داشت؛ به عنوان مثال:

public JsonResult Index()

{

var headers = this.HttpContext.Request.Headers;

return Json(headers);

}

**نکته: بهتر است از استفاده مستقیم از HttpContext خودداری کنید، زیرا تست کردن را سخت خواهد کرد. در ادامه از یک مکانیزم friendlier توکار در MVC استفاده خواهیم کرد.**

اجازه دهید یک Model به پروژه اضافه کنیم:

namespace OdeToFood.Models

{

public class Restaurant

{

public int Id { get; set; }

public string Name { get; set; }

}

}

اکنون می‌خواهیم اطلاعات رستوران را به عنوان خروجی اکشن‌متد قبل تعیین کنیم:

public class HomeController : Controller

{

public IActionResult Index()

{

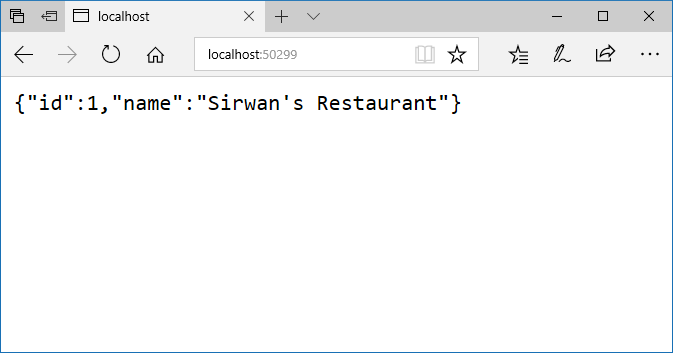
var model = new Restaurant { Id = 1, Name = "Sirwan's Restaurant" };

return new ObjectResult(model);

}

}

تا اینجا کنترلر دیدی ندارد که خروجی این ObjectResult دقیقاً چه چیزی خواهد بود. تنها چیزی که می‌داند این است که اطلاعاتی که می‌خواهیم در خروجی ظاهر شوند، اطلاعات یک رستوران است. مولفه‌ی دیگر در pipeline قرار است نوع این ObjectResult را تشخیص دهد. اگر پروژه را اجرا کنید خواهید دید که خروجی به صورت JSON می‌باشد:



نکته: کار کنترلر: درخواست به اکشن‌متد وارد خواهد شد، اطلاعات از منبع داده (دیتابیس، وب‌سرویس و...) دریافت خواهند شد، از این اطلاعات جهت ایجاد مدل استفاده خواهد شد، در نهایت خروجی را توسط یکی از IActionResultها مشخص خواهیم کرد.

## Rendering Views

در MVC یکی از رایج‌ترین روش‌های تولید HTML استفاده از razor view engine است. برای استفاده از این قابلیت اکشن متد باید یک شیء **ViewResult** (این شیء اینترفیس **IActionResult** را پیاده‌سازی کرده است) تولید کند. ViewResult نیز نام یک ویوی خاص را به همراه دارد؛ این ویو بر روی دیسک ذخیره شده است؛ همچنین یک مدل را برای استفاده در ویو نیز منتقل خواهد کرد.



به عنوان مثال با داشتن اکشن متد زیر:

public class HomeController : Controller

{

public IActionResult Index()

{

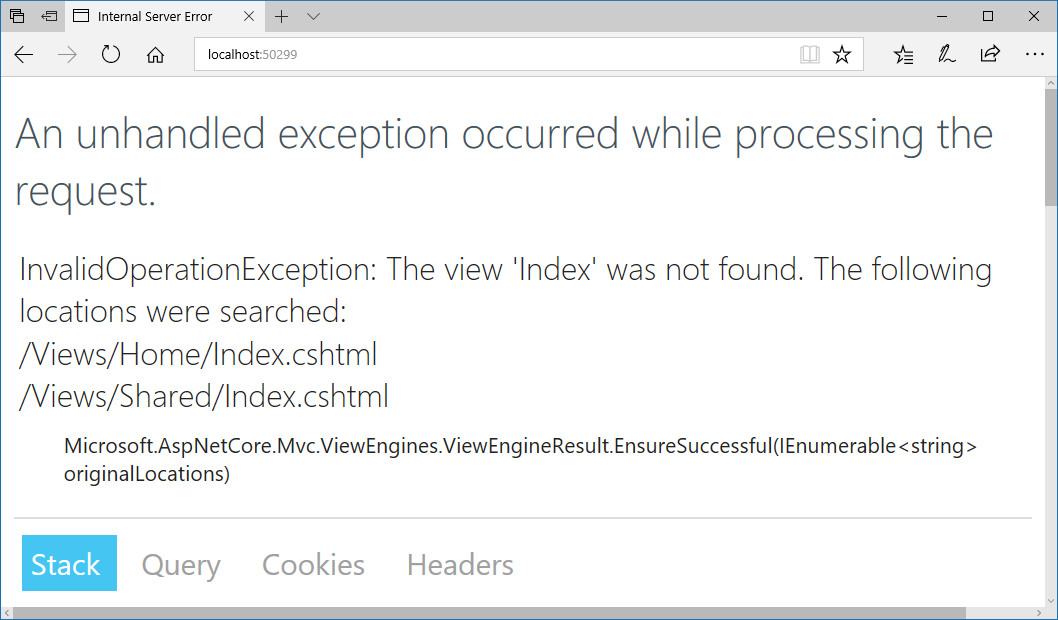
var model = new Restaurant { Id = 1, Name = "Sirwan's Restaurant" };

**return View(model);**

}

}

خطای زیر را دریافت خواهیم کرد؛ زیرا Viewایی با نام Index برای اکشن متد فوق وجود ندارد:



برای حل این مشکل باید ویوی موردنظر را در یکی از مکان‌های اشاره شده در خطای فوق ایجاد کنیم.

ویوی Index:

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title></title>

</head>

<body>

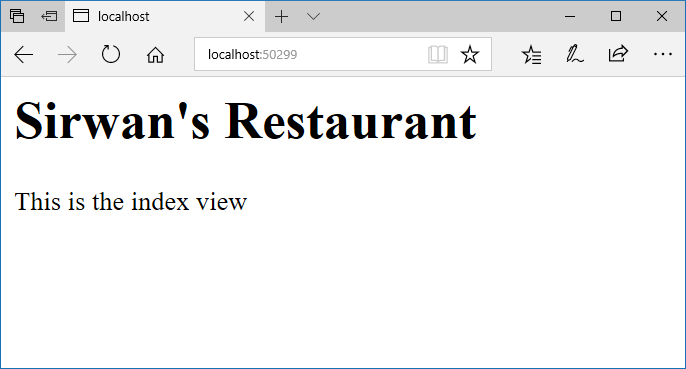
<h1>@Model.Name</h1>

<div>This is the index view</div>

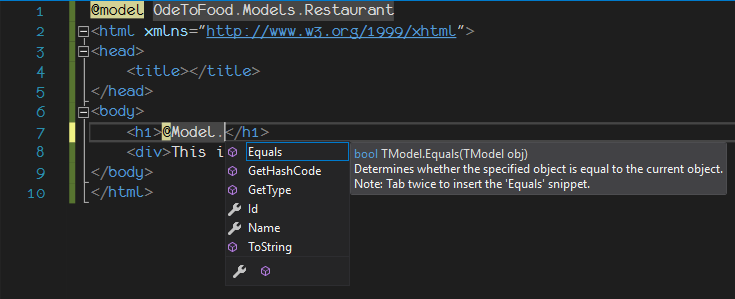
</body>

</html>

خروجی:



نکته: درون ویو اگر دقت کرده باشید Intellisense برای نمایش پراپرتی‌های مدل وجود ندارد. برای داشتن این قابلیت باید از دایرکتیو **@model** و همچنین تعیین نوع مدل درون ویو استفاده کنیم:



## A Table Full of Restaurants

می‌خواهیم درون ویوی Index لیستی از رستوران‌ها را نمایش دهیم. برای اینکار نیاز به یک سرویس داریم؛ این سرویس می‌تواند دیتابیس باشد اما برای سادگی می‌خواهیم از یک لیست درون حافظه‌ایی برای اینکار استفاده کنیم:

namespace OdeToFood.Services

{

public interface IRestaurantData

{

IEnumerable<Restaurant> GetAll();

}

public class InMemoryRestaurantData : IRestaurantData

{

public InMemoryRestaurantData()

{

\_restaurants = new List<Restaurant>

{

new Restaurant { Id = 1, Name = "Sirwan's Pizza Place" },

new Restaurant { Id = 2, Name = "Tersiguels" },

new Restaurant { Id = 3, Name = "King's Contrivance" }

};

}

List<Restaurant> \_restaurants;

public IEnumerable<Restaurant> GetAll()

{

return \_restaurants.OrderBy(x => x.Name);

}

}

}

**نکته: هر زمانیکه از لیست استفاده می‌کنید دقت داشته باشید که List<T> به صورت thread-safe نیست؛ بنابراین باید در حین استفاده از آن در یک اپلیکیشن وب مواظب باشید؛ به خصوص زمانیکه می‌خواهیم این سرویس را در بین چندین درخواست به اشتراک بگذارید.**

اکنون از سرویس فوق می‌توانیم درون کنترلرهایمان استفاده کنیم اما نمی‌خواهیم این سرویس را به صورت مستقیم وهله‌سازی کنیم. یعنی نمی‌خواهیم کنترلرمان اطلاعی از وجود concrete class؛ یعنی کلاس InMemoryRestaurant داشته باشد. در عوض می‌خواهیم کنترلرمان با اینترفیس IRestaurant در ارتباط باشد (programming to interfaces). برای اینکار، کار وهله‌سازی سرویس را به عهده IoC خواهیم گذاشت:

**services.AddScoped<IRestaurantData, InMemoryRestaurantData>();**

اکنون درون کنترلر خواهیم داشت:

public class HomeController : Controller

{

**private IRestaurantData \_restaurantData;**

public HomeController(**IRestaurantData restaurantData**)

{

**\_restaurantData = restaurantData;**

}

public IActionResult Index()

{

var model = \_restaurantData.GetAll();

return View(model);

}

}

در اینحالت HomeController هیچ اطلاعی از concrete serviceی که اینترفیس IRestaurantData را پیاده‌سازی کرده است، ندارد.

ویو:

@model **IEnumerable<OdeToFood.Models.Restaurant>**

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title></title>

</head>

<body>

<table>

**@foreach (var restaurant in Model)**

**{**

**<tr>**

**<td>@restaurant.Id</td>**

**<td>@restaurant.Name</td>**

**</tr>**

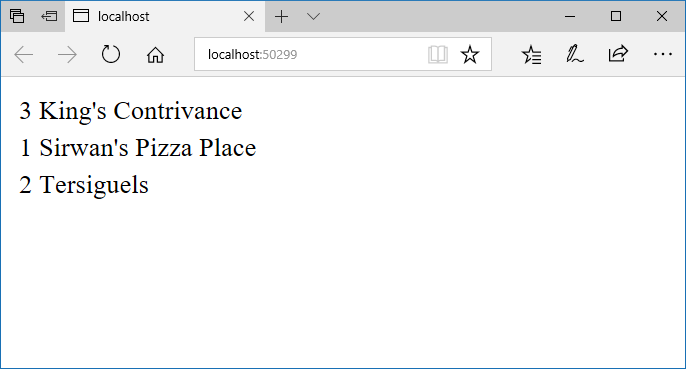
**}**

</table>

</body>

</html>

خروجی:



**زیبایی این مثال این است که اگر بعداً به استفاده از SQL Server سوئیچ کنیم، نیازی به تغییر هیچکدام از ویوها و کنترلرهایمان نخواهیم داشت.**

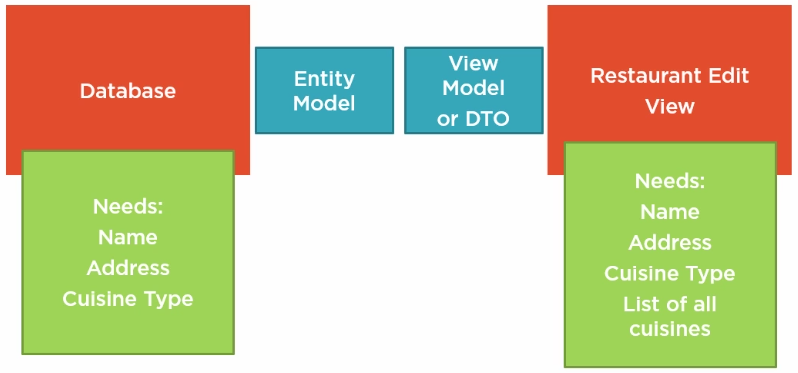
# Models in the MVC Framework

## Models and View Models

فرض کنید یک اپلیکیشن دارید که قرار است یکسری اطلاعات را ذخیره کند. وقتی چیزی مانند یک رستوران را درون دیتابیس ذخیره می‌کنید، دیتابیس نیاز به یکسری اطلاعات خاص در مورد رستوران دارد؛ نام – آدرس – نوع غذا و...

در جایی درون اپلیکیشن‌تان نیز نیاز خواهید داشت که اطلاعات یک رستوران خاص را ویرایش کنید. ویوی edit نیز نیاز به یکسری اطلاعات دارد؛ نام – آدرس – نوع غذا – لیست تمامی غذاها و...

همانطور که مشاهده می‌کنید ویوی edit نیاز به اطلاعات بیشتری نیست به فیلدهای درون دیتابیس دارد.



برای مثال در ویوی edit ممکن است یک لیست برای انتخاب نوع غذا داشته باشیم؛ بنابراین در حین کوئری گرفتن از دیتابیس برای ویرایش یک رستوران، اطلاعات یک رستوران همراه با نوع غذای فعلی دریافت خواهیم کرد نه تمام غذاهای احتمالی. در این شرایط از یکی از دو model objects زیر می‌توانیم استفاده کنیم:

* **Entity Model**: یک موجودیت یا اینتینی در واقع شیءی است که درون دیتابیس ذخیره خواهد شد. این مدل معمولاً شبیه به اسکیمای دیتابیس است.
* **View Model or DTO**: شیء است که اطلاعات را از کنترلر به ویو منتقل خواهد کرد. این شیء نه تنها حاوی اطلاعات اینتیتی است بلکه حاوی اطلاعات موردنیاز ویو نیز خواهد بود. به این دلیل است که معمولاً به ویو مدل DTO (Data Transfer Object) گفته می‌شود. DTO درون دیتابیس ذخیره نخواهد شد، بلکه کار انتقال اطلاعات را از ویو به مدل و یا برعکس انجام خواهد داد.

مثال: فرض کنید برای مثال قبل، صفحه اصلی علاوه بر لیست رستوران‌ها می‌خواهیم پیام امروز را نیز نمایش دهیم؛ که در واقع از دیتاسورس دیگری خواهد آمد؛ در حال حاضر صفحه Index تنها تعدادی رستوران را به ویو انتقال می‌دهد اما اکنون می‌خواهیم همراه با این لیست یک مقدار دیگر را نیز به ویو جهت نمایش ارسال کنیم. در اینحالت به یک special purpose view model class نیاز خواهیم داشت؛ کلاسی که به صورت اختصاصی برای اکشن Index استفاده خواهد شد و تمامی اطلاعاتی که ویوی آن نیاز خواهد داشت را با خود ارسال می‌کند.

قدم اول: ایجاد ویو مدل:

using OdeToFood.Models;

using System.Collections.Generic;

namespace OdeToFood.ViewModels

{

public class HomeIndexViewModel

{

public IEnumerable<Restaurant> Restaurants { get; set; }

public string CurrentMessage { get; set; }

}

}

قدم دوم: استفاده از ویو مدل فوق درون اکشن متد:

using Microsoft.AspNetCore.Mvc;

using OdeToFood.Services;

using OdeToFood.ViewModels;

namespace OdeToFood.Controllers

{

public class HomeController : Controller

{

private IRestaurantData \_restaurantData;

private IGreeter \_greeter;

public HomeController(IRestaurantData restaurantData

, IGreeter greeter)

{

\_restaurantData = restaurantData;

\_greeter = greeter;

}

public IActionResult Index()

{

var model = new HomeIndexViewModel

{

Restaurants = \_restaurantData.GetAll(),

CurrentMessage = \_greeter.GetMessageOfTheDay()

};

return View(model);

}

}

}

قدم دوم: استفاده از ویو مدل فوق درون ویو:

@model OdeToFood.ViewModels.HomeIndexViewModel

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title></title>

</head>

<body>

<h1>@Model.CurrentMessage</h1>

<table>

@foreach (var restaurant in Model.Restaurants)

{

<tr>

<td>@restaurant.Id</td>

<td>@restaurant.Name</td>

</tr>

}

</table>

</body>

</html>

خروجی:



## Detail a Restaurant

قدم اول: تعریف اکشن متد:

public IActionResult Details(int id)

{

var model = \_restaurantData.Get(id);

return View(model);

}

تعریف متد Get:

public interface IRestaurantData

{

IEnumerable<Restaurant> GetAll();

Restaurant Get(int id);

}

public class InMemoryRestaurantData : IRestaurantData

{

// as before

public Restaurant Get(int id)

{

return \_restaurants.FirstOrDefault(r => r.Id == id);

}

}

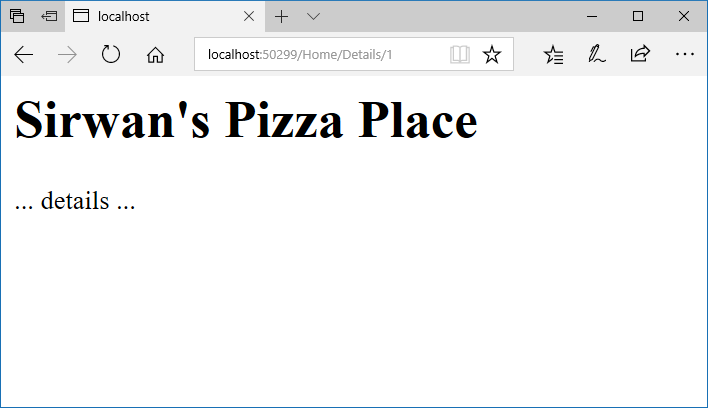
قدم سوم: تعریف ویوی Details:

@model OdeToFood.Models.Restaurant

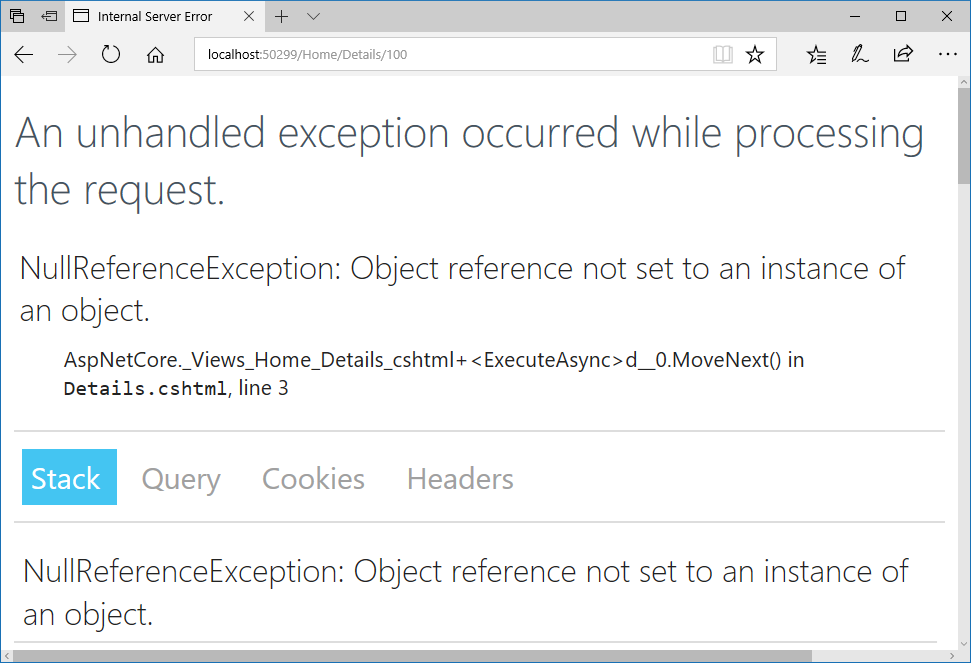
<h1>@Model.Name</h1>

<div> ... details ... </div>

خروجی:



اگر یک آی‌دی را وارد کنیم که وجود نداشته باشد خطای زیر را دریافت خواهیم کرد:



زیرا خروجی **FirstOrDefault** در صورتیکه آی‌دی موردنظر را پیدا نکند null خواهد بود. برای حل این مشکل چندین استراتژی وجود دارد:

* بررسی null بودن Model درون ویو:
* @model OdeToFood.Models.Restaurant
* @if (Model == null)
* {
* <div>Not found</div>
* }
* else
* {
* <h1>@Model.Name</h1>
* <div> ... details ... </div>
* }
* بررسی null بودن درون اکشن متد:
* public IActionResult Details(int id)
* {
* var model = \_restaurantData.Get(id);
* if (model == null)
* {
* return RedirectToAction("Index");
* }
* return View(model);
* }

برای دسترسی سریع‌تر به صفحه‌ی Details می‌توانیم لینک آن را درون صفحه‌ی اصلی قرار دهیم؛ به چند روش می‌توانیم اینکار را انجام دهیم:

* HTML ساده:
* @foreach (var restaurant in Model.Restaurants)
* {
* <tr>
* <td>@restaurant.Id</td>
* <td>@restaurant.Name</td>
* <td>
* **<a href="/home/details/@restaurant.Id"></a>**
* </td>
* </tr>
* }
* استفاده از Razor:
* @foreach (var restaurant in Model.Restaurants)
* {
* <tr>
* <td>@restaurant.Id</td>
* <td>@restaurant.Name</td>
* <td>
* **@Html.ActionLink("Details", "Details", new { Id = restaurant.Id })**
* </td>
* </tr>
* }

استفاده از ActionLink کمی جالب نیست زیرا خوانایی آن ساده نیست؛ پس یک روش سوم را معرفی خواهیم کرد؛ اما برای استفاده از این روش باید ابتدا فایلی تحت عنوان \_ViewImports.cshtml را به پوشه Views اضافه کنیم:

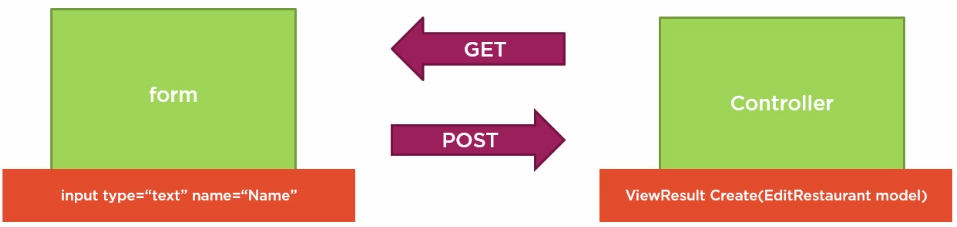
**@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers**

با اینکار TagHelper به تمامی ویوهایمان اضافه خواهد شد و می‌توانیم از آن استفاده کنیم.

* استفاده از TagHelper
* @foreach (var restaurant in Model.Restaurants)
* {
* <tr>
* <td>@restaurant.Id</td>
* <td>@restaurant.Name</td>
* <td>
* **<a asp-action="Details" asp-route-id="@restaurant.Id">Details</a>**
* </td>
* </tr>
* }

## Create a Restaurant

در ادامه نحوه‌ی ایجاد یک رستوران را بررسی خواهیم کرد.



قدم اول: تعریف اکشن متد:

public IActionResult Create()

{

return View();

}

قدم دوم: تعریف ویو Create:

@using OdeToFood.Models

@model OdeToFood.Models.Restaurant

<h1>Create</h1>

<form method="post">

<input asp-for="Name" />

<select asp-for="Cuisine"

asp-items="@Html.GetEnumSelectList<Cuisine>()">

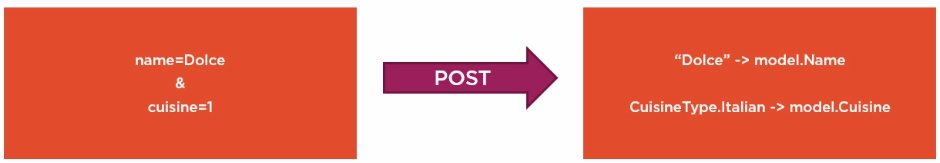
</select>

<input type="submit" name="save" value="Save" />

</form>

## Accepting Form Input

پراپرتی‌ها براساس ویژگی Name مپ خواهند شد:



دریافت داده‌های ارسالی از فرم:

قدم اول: تعریف اکشن متد:

[HttpPost]

public IActionResult Create(RestaurantEditModel model)

{

var newRestaurant = new Restaurant();

newRestaurant.Name = model.Name;

newRestaurant.Cuisine = model.Cuisine;

newRestaurant = \_restaurantData.Add(newRestaurant);

return View("Details", newRestaurant);

}

قدم دوم: افزودن متد Add:

**public Restaurant Add(Restaurant restaurant)**

**{**

**restaurant.Id = \_restaurants.Max(r => r.Id) + 1;**

**\_restaurants.Add(restaurant);**

**return restaurant;**

**}**

نکته: در سناریوی فوق کاربر با ایجاد یک آیتم جدید و کلیک بر روی کلید Create باید آیتم جدید را درون صفحه Details مشاهده کند؛ اما اگر پروژه را اجرا کنید خواهید دید که بعد از ایجاد آیتم جدید و مراجعه به لیست آیتم، آیتم جدیدی وجود ندارد. دلیل آن نیز این است که برای سرویس IRestaurantData از حالت Scoped استفاده کرده‌ایم؛ اینحالت یعنی یک وهله از InMemoryRestaurantData را برای هر درخواست HTTP ایجاد کن؛ در طول درخواست از این وهله استفاده خواهد شد اما با پایان درخواست HTTP این وهله از حافظه پاک خواهد شد. برای حل این مشکل باید طول عمر را به Singleton تغییر دهیم:

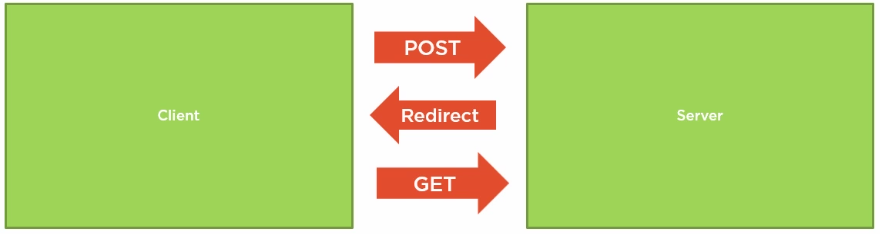
services.**AddSingleton**<IRestaurantData, InMemoryRestaurantData>();

## POST Redirect GET Pattern

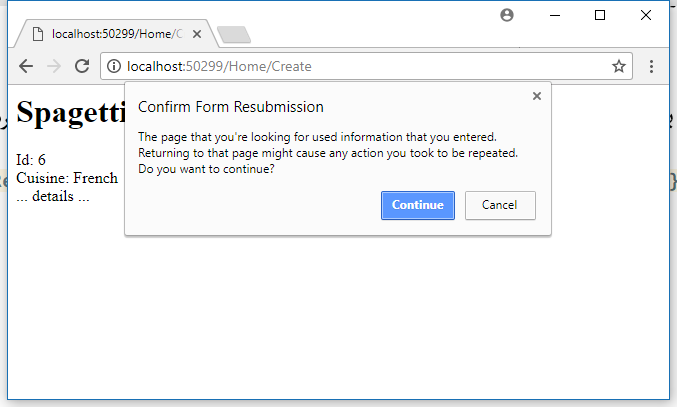
تا اینجا کلاینت؛ یعنی مرورگر ما توسط یک فرم با اکشن post اطلاعات را (یک رستوران) به سمت سرور ارسال خواهد کرد؛ در نهایت سرور بلافاصله اطلاعات رستوران جدید را در یک صفحه دیگر نمایش خواهد داد. اما پاسخ دادن به یک عملیات POST درون یک صفحه‌ی HTML می‌تواند باعث بروز مشکلاتی در اپلیکیشن شود؛ زیرا اگر کاربر تصمیم بگیرد مرورگر را ریفرش کند، مرورگر سعی خواهد کرد یک درخواست دیگر به سرور ارسال کند؛ که اینکار باعث ثبت یک رستوران جدید در دیتابیس خواهد شد.

به عنوان یک best practice در وب اپلیکیشن‌ها بهتر است بعد از یک عملیات POST به این عملیات با یک http status redirect code پاسخ دهید؛ یعنی به مرورگر بگوئید که یک GET request را برای خواندن داده‌ها ارسال کند. بخاطر داشته باشید که POST برای عملیات نوشتن است و GET برای عملیات خواندن است.

الگوی **POST Redirect GET** در نهایت یک صفحه را در اختیار کاربر قرار خواهد داد که توسط کاربر خوانده شود؛ یعنی صفحه‌ایی که قابلیت ریفرش شدن و بوکمارک کردن را داشته باشد؛



همانطور که در مثال قبل دیدید بعد از ثبت رستوران جدید ویوی Details را در خروجی رندر خواهیم کرد؛ در این صفحه اگر ریفرش کنیم با پیغام زیر مواجه خواهیم شد که با کلیک بر روی Continue اطلاعات تکراری درون دیتابیس ذخیره خواهد شد:

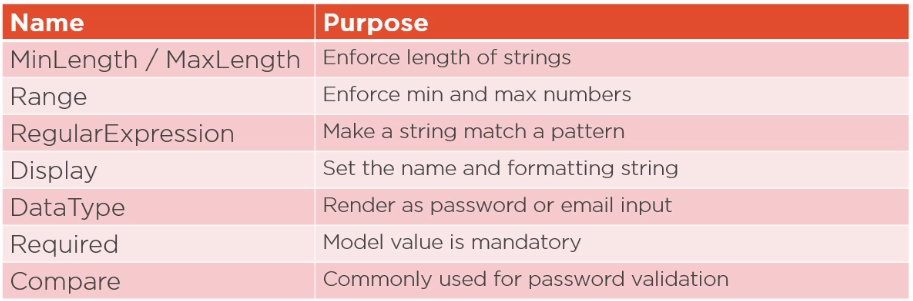


برای حل این مشکل به جای return کردن یک ویو، کاربر را به یک اکشن متد دگر هدایت خواهیم کرد:

**return RedirectToAction("Details", new { id = newRestaurant.Id });**

## Model Validation with Data Annotations

تعدادی از Data Annotations موجود در .NET



افزودن Data Annotation به مدل و مجبور کردن Model Validation:

برای اینکه دیگر annotations به خوبی کار کنند، بهتر است در ویو از هلپر EditorFor که یک نسخه‌ی جنریک‌تر از TextBoxFor است، استفاده کنیم. این هلپر براساس annotations یک تگ مناسب را تولید خواهد کرد.

public class Restaurant

{

public int Id { get; set; }

**[Display(Name = "Restaurant Name")]**

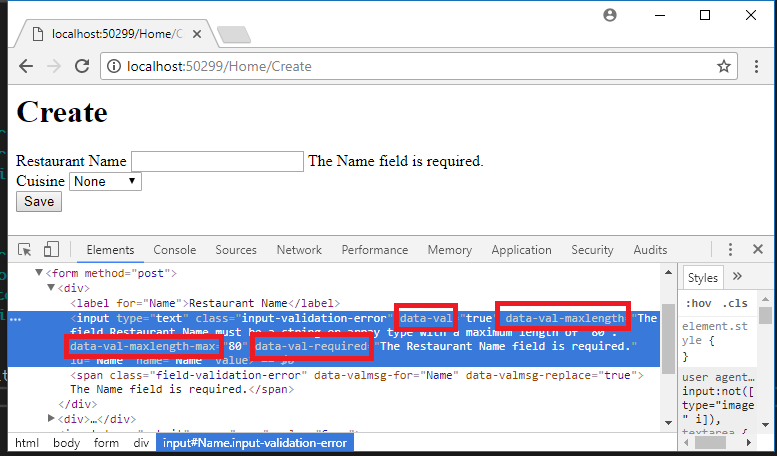
**[Required, MaxLength(80)]**

public string Name { get; set; }

public Cuisine Cuisine { get; set; }

}

خروجی:



این اتریبیوت‌های خاص برای اعمال client side validation مورد استفاده قرار می‌گیرند؛ چیزی که مورد استفاده ما است server side validation می‌باشد زیرا **ما نباید هیچ وقت به کلاینت اعتماد کنیم.**

MVC وقتی ویومدل‌مان را با داده‌های ارسالی توسط فرم populate کرد، validation rules را بر روی ویومدل اعمال خواهد کرد و بررسی خواهد کرد که آیا مدل دریافتی معتبر است یا خیر. وضعیت مدل را می‌توانیم توسط پراپرتی ModelState.IsValid بررسی کنیم:

[HttpPost]

public IActionResult Create(RestaurantEditModel model)

{

**if (ModelState.IsValid)**

**{**

var newRestaurant = new Restaurant();

newRestaurant.Name = model.Name;

newRestaurant.Cuisine = model.Cuisine;

newRestaurant = \_restaurantData.Add(newRestaurant);

return RedirectToAction("Details", new { id = newRestaurant.Id });

**}**

return View();

}

نمایش خطاها در ویو:

@using OdeToFood.Models

@model OdeToFood.Models.Restaurant

<h1>Create</h1>

<form method="post">

<div>

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

<input asp-for="Name" />

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

</div>

<div>

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

<select asp-for="Cuisine"

asp-items="@Html.GetEnumSelectList<Cuisine>()"></select>

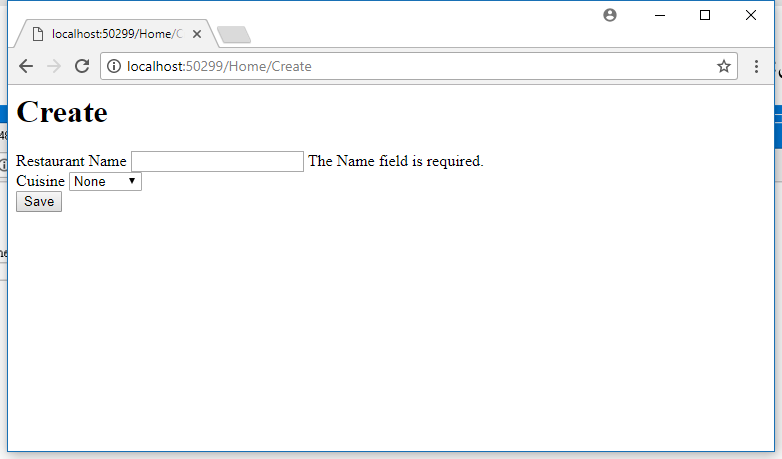
**<span asp-validation-for="Cuisine"></span>**

</div>

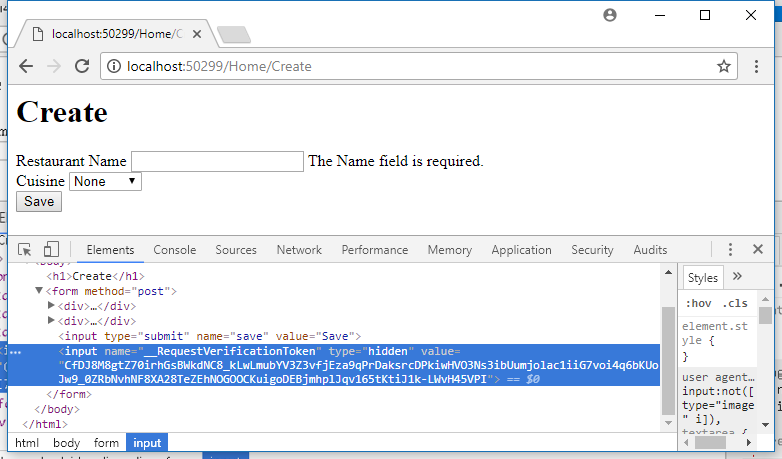
<input type="submit" name="save" value="Save" />

</form>

خروجی:



وظیفه‌ی فیلد مخفی درون فرم با نام **RequestVerificationToken** چیست؟

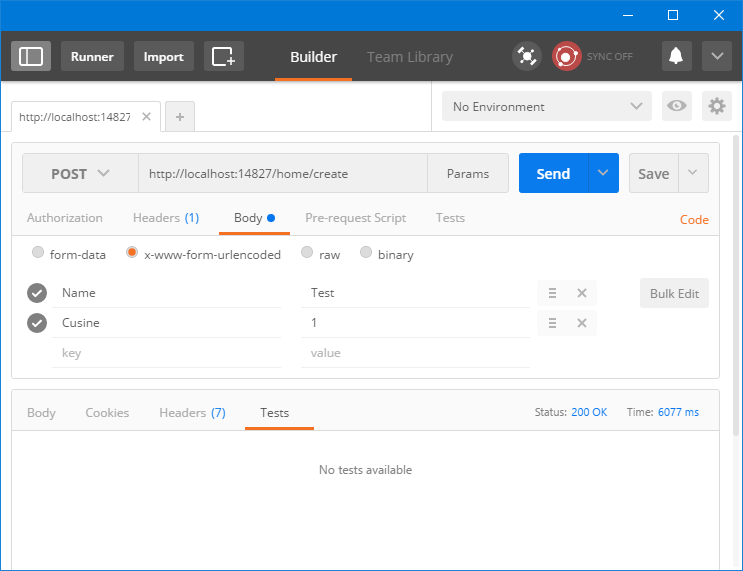


**مقدار این فیلد مخفی تضمین خواهد کرد که اطلاعاتی که کاربر برایمان ارسال کرده است از طریق فرمی است که در اختیار کاربر قرار داده‌ایم نه از جایی دیگر یعنی از وقوع حمله‌ی Cross Site Request Forgery (CSRF) جلوگیری خواهد کرد.**

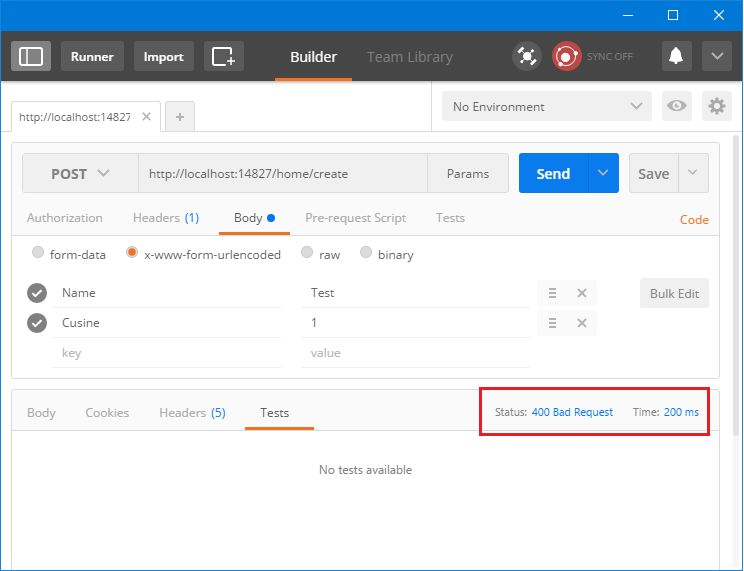
برای استفاده از این قابلیت باید فرم‌های POSTتان را با اتریبیوت زیر مزین نمائید:

**[ValidateAntiForgeryToken]**

مثال: اگر از ویژگی فوق استفاده نکنید به راحتی توسط هر ابزاری می‌توانید یک فرم را به سرور ارسال کنید:



اما با استفاده از ویژگی **ValidateAntiForgeryToken** هیچ کسی جز فرمی که در اختیار کاربر قرار داده‌ایم، قادر به ارسال اطلاعات به سرور نخواهد بود؛ به نوعی ارسال اطلاعات تنها توسط فرم خودمان قابل انجام است:



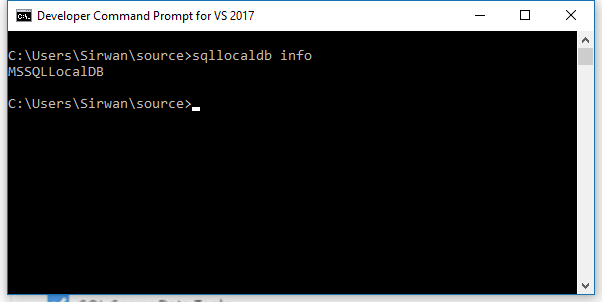
# Using the Entity Framework

## Introduction

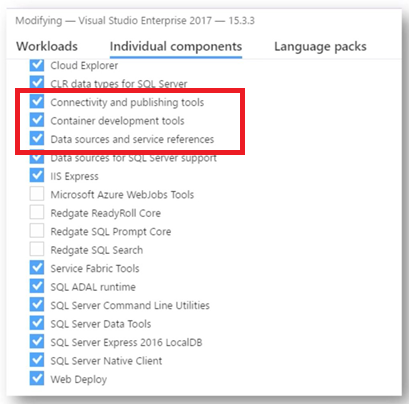
در این ماژول می‌خواهیم اپلیکیشن‌مان اطلاعات را درون دیتابیس درج و بازیابی نمائید.

## SQL Server LocalDB

LocalDB یک نسخه از SQL Server است که برای توسعه‌دهندگان بهینه شده است. این نسخه از SQL Server همراه با Visual Studio 2017 نصب خواهد شد. برای اینکه مطمئن شوید نصب شده است می‌توانید از دستور زیر استفاده کنید:



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



کانشکن استرینگ:

Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=master;

Integrated Security=True;Connect Timeout=30;

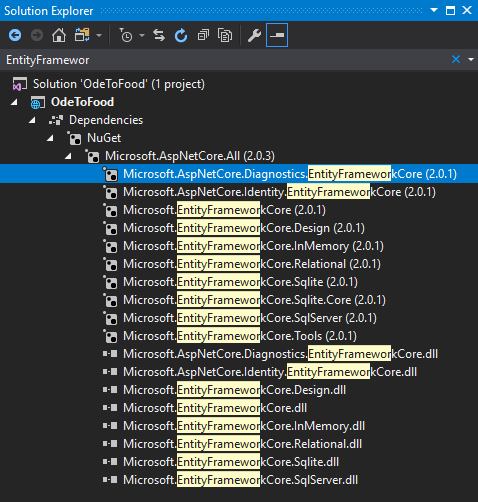
Encrypt=False;TrustServerCertificate=True;

ApplicationIntent=ReadWrite;MultiSubnetFailover=False

## Installing the Entity Framework

نصب EF:

از آنجائیکه پروژه ما از متاپکیج **Microsoft.AspNetCore.All** استفاده می‌کند، در نتیجه EF Core نیز درون این پکیج قرار دارد و نیازی به نصب مجدد آن نیست:



برای فعال‌سازی فرمان dotnet ef باید package reference زیر را نیز به پروژه اضافه کنیم:

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>netcoreapp2.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />

</ItemGroup>

**<ItemGroup>**

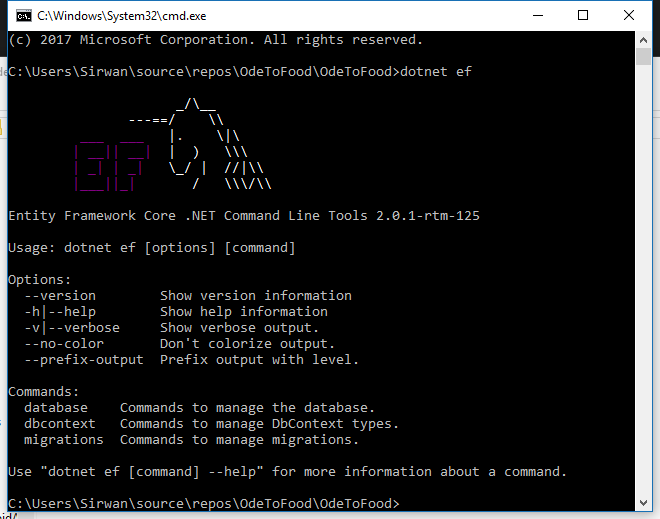
**<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1">**

**</DotNetCliToolReference>**

**</ItemGroup>**

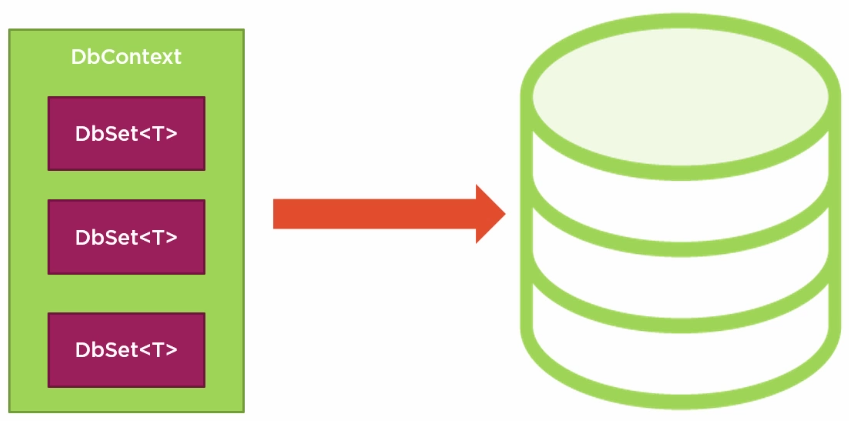
</Project>

اکنون فرمان dotnet ef از طریق خط فرمان برای پروژه در دسترس می‌باشد:



## Implementing a DbContext

برای استفاده از EF باید یک کلاس ایجاد کنیم که از کلاس **DbContext** ارث‌بری کند. هر DbContext که ایجاد می‌کنیم اجازه‌ی دسترسی به **یک** دیتابیس را به ما می‌دهد. با تغییر connection string کلاس context مان می‌تواند به دیتابیس‌های فیزیکی مختلفی دسترسی داشته باشد.



درون کلاس context مان می‌توانیم یکسری **DbSet<T>** تعریف کنیم. Generic type parameter T در واقع نوع entity مانند Restaurant و... است. هر DbSet درون کانتکست به یک جدول در دیتابیس نگاشت داده خواهد شد:



قدم اول: ایجاد کلاس کانتکست:

using Microsoft.EntityFrameworkCore;

using OdeToFood.Models;

namespace OdeToFood.Data

{

public class OdeToFoodDbContext : DbContext

{

public OdeToFoodDbContext(DbContextOptions options)

: base(options)

{

}

public DbSet<Restaurant> Restaurants { get; set; }

}

}

قدم دوم: ایجاد یک پیاده‌سازی دیگر برای **IRestaurantData**:

public class SqlRestaurantData : IRestaurantData

{

private OdeToFoodDbContext \_context;

public SqlRestaurantData(OdeToFoodDbContext context)

{

\_context = context;

}

public Restaurant Add(Restaurant restaurant)

{

\_context.Restaurants.Add(restaurant);

\_context.SaveChanges();

return restaurant;

}

public Restaurant Get(int id)

{

return \_context.Restaurants.FirstOrDefault(x => x.Id == id);

}

public IEnumerable<Restaurant> GetAll()

{

return \_context.Restaurants.OrderBy(x => x.Name);

}

}

**نکته: فراخوانی متد SaveChanges() نباید درون متد Add صورت بگیرد؛ در شرایطی ممکن است چندین دستور Add, Update, Delete را برای اشیاء دیگر طی یک درخواست نیاز داشته باشید؛ در این حالت بهتر است بعد از اتمام تمامی عملیات بر روی اشیاء متد SaveChanges را فراخوانی کنید. (**[**مطالعه بیشتر – الگوی واحد کار**](http://www.dotnettips.info/post/2509)**)**

## Configuring the Entity Framework Services

برای داشتن یک DbContext با قابلیت استفاده مجدد، باید مقداری Configuration اپلیکیشن را تغییر دهیم؛ باید به فریم‌ورک بگوئیم که از چه connection string, server قرار است استفاده کنیم؛ از آنجائیکه connection string همیشه تغییر می‌کند آن را درون appsettings.json ذخیره خواهیم کرد:

{

"Greeting": "Hello!!",

"ConnectionStrings": {

**"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=OdeToFood;Trusted\_Connection=True;MultipleActiveResultSets=true"**

**}**

}

در ادامه باید به جای سرویس InMemoryRestaurantData از سرویس SqlRestaurantDat استفاده کنیم:

services.**AddScoped**<IRestaurantData, **SqlRestaurantData**>();

همانطور که ملاحظه می‌کنید برای این سرویس از AddScoped استفاده کرده‌ایم؛ زیرا DbContext در واقع به صورت thread-safe نیست؛ در اینحالت مطمئن خواهیم شد که DbContext تنها درون یک ترد در دسترس قرار خواهد گرفت.

در قدم بعدی یاد سرویس‌های موردنیاز EF را ریجستر کنیم:

**private IConfiguration \_configuration;**

**public Startup(IConfiguration configuration)**

**{**

**\_configuration = configuration;**

**}**

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<IGreeter, Greeter>();

**services.AddDbContext<OdeToFoodDbContext>(options =>**

**{**

**options.UseSqlServer(\_configuration.GetConnectionString("DefaultConnection"));**

**});**

services.AddScoped<IRestaurantData, SqlRestaurantData>();

services.AddMvc();

}

## Entity Framework Migrations

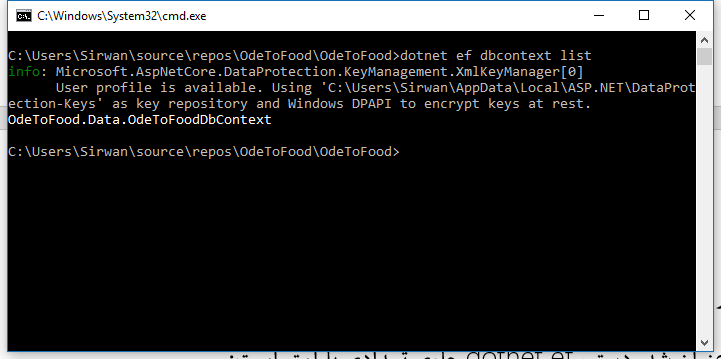
یک روش برای ایجاد دیتابیس، وادار کردن EF به ساخت دیتابیس است؛ این فرآیند با دو مرحله انجام خواهد شد:

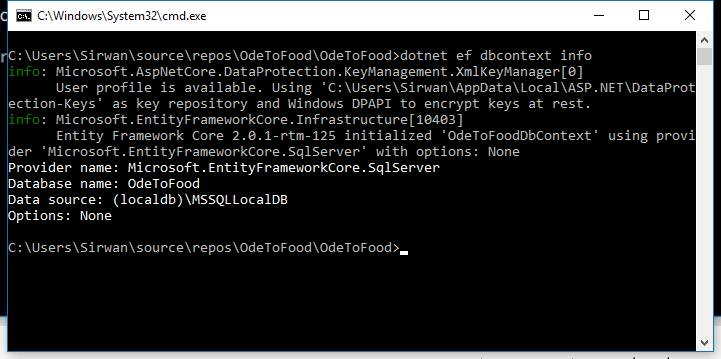


* اضافه کردن کدهای مهاجرت به پروژه
  + در واقع منظور از کدهای مهاجرت اجرای یکسری کد سی‌شارپ جهت ایجاد دیتابیس است. EF می‌تواند کدهای مهاجرت را برایمان تولید کند؛ ابتدا به دیتابیس مراجعه کرده و در صورت وجود آن را با اینتینی مطابقت خواهد داد و تغییرات اسکیما را از روی مدل به دیتابیس اعمال خواهد کرد. برای اضافه کرده migration باید از دستور **dotnet ef migrations add** استفاده کنیم.
* اعمال دستور **dotnet ef database update** جهت اعمال تغییرات و بروزرسانی دیتابیس

همانطور که قبلاً نیز عنوان شد، دستور dotnet ef حاوی تعدادی پارامتر است:

* لیست contextهای برنامه:



* اطلاعاتی در مورد contextهای پروژه جاری:
* 

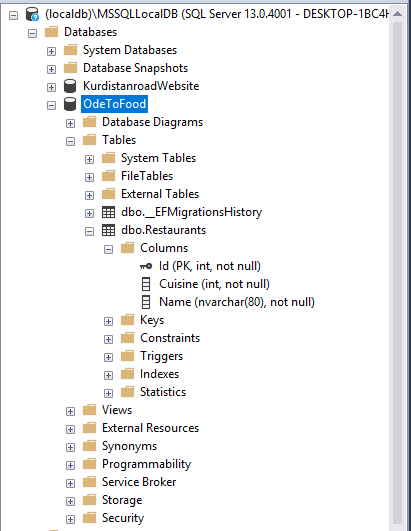
قدم اول: اضافه کردن کدهای مهاجرت:

dotnet ef migrations add IntialCreate -v

قدم دوم: اعمال تغییرات به دیتابیس:

dotnet ef database update

خروجی:



# Razor Views

## Layout Views

لی‌اوت باید در مسیر زیر ایجاد شود:



محتویات لی‌اوت:



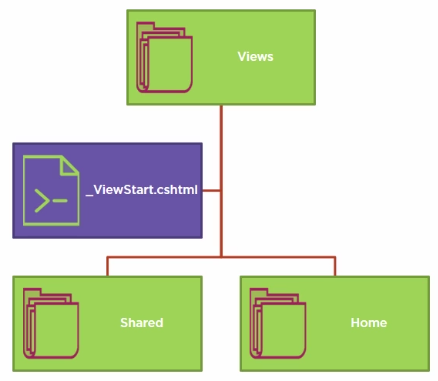
برای ایجاد section سفارشی نیز می‌توانیم از هلپر RenderSection استفاده کنیم.

اکنون درون ویوها دیگر نیاز به نوشتن کامل ساختار HTML نیست. در قدم بعدی باید دو کار انجام دهیم:

* استفاده از لی‌اوت فوق برای ویوها
* ست کردن ViewBag مربوط به عنوان صفحه
* **@{**
* **Layout = "\_Layout";**
* **ViewBag.Title = "Home Page";**
* **}**

## \_ViewStart

فریم‌ورک MVC قبل از اجرای کدهای ویوها ابتدا محتویات ویوی‌ایی با نام \_ViewStart.cshtml را اجرا خواهد کرد:



محتویات و محل قرار دادن فایل \_ViewStart.cshtml

**@{**

**Layout = "\_Layout";**

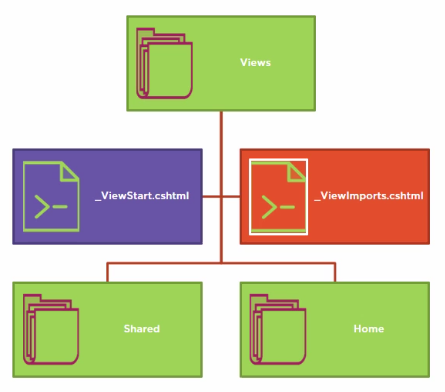
**}**

اکنون دیگر درون ویوها نیازی به تعیین لی‌اوت نیست.

**نکته: اگر فایل \_ViewStart.cshtml را درون پوشه‌ی Home قرار دهیم؛ لی‌اوت تنها برای ویوهای Home در نظر گرفته خواهد شد. اما با قرار دادن این فایل درون پوشه‌ی Views لی‌اوت برای تمامی ویوها تنظیم خواهد شد.**

## \_ViewImports

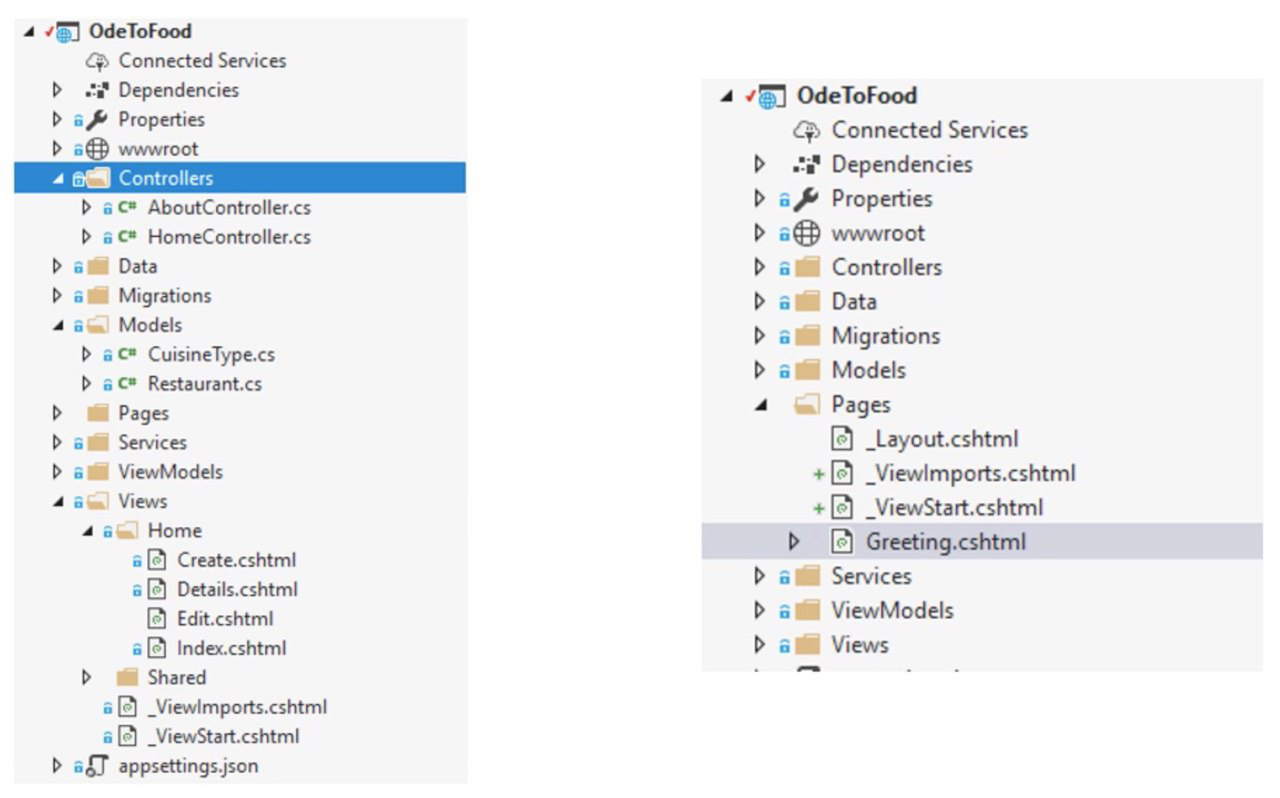
علاوه بر قرار دادن فایل \_ViewStart.cshtml می‌توانیم از فایل \_ViewImports.cshtml نیز جهت افزودن namespaceهای موردنیاز برای ویوها نیز استفاده کنیم:



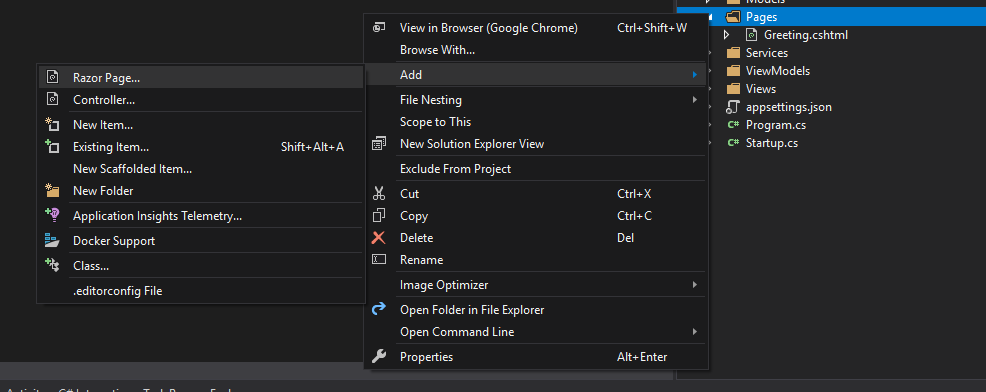
در ورژن‌های قبلی MVC از یک فایل XML برای اینکار استفاده می‌شد.

## Razor Pages

Razor Pages یک قابلیت جدید است که به نسخه ASP.NET Core 2.0 اضافه شده است. اگر هدف‌تان ایجاد اپلیکیشنی است که عموماً کارش تولید HTML است؛ می‌توانید از این قابلیت استفاده کنید. این قابلیت یک روش جایگزین برای الگوی MVC جهت تولید HTML است. در MVC این کنترلر است که با دریافت درخواست یک مدل ایجاد کرده و آن را به ویو جهت رندر شدن ارائه می‌دهد. اما Razor Pages مقداری ساده‌تر است؛ در واقع یک درخواست مستقیماً به یک صفحه Razor که درون پوشه Pages واقع در ریشه پروژه، هدایت خواهد شد؛ این صفحه نیز می‌تواند مدل موردنظر را تولید و خروجی مناسب برای کاربر را تولید کند. بنابراین اگر می‌خواهید وب‌اپلیکیشن‌های مبتنی بر سرویس یا سرویس‌های مبتنی بر HTTP یا Rest ایجاد کنید از الگوی MVC استفاده کنید، اما هدف از Razor Pages ایجاد صفحات HTML است.



قدم اول ایجاد یک Razor Page:



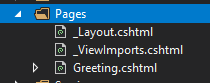
محتویات فایل Greeting.cshtml:

@page

@{

}

اکنون هر درخواستی به /greeting بیاید، به صورت خودکار ویو فوق به عنوان خروجی نمایش داده می‌شود. در واقع دایرکتیو @page است که اینکار را انجام خواهد داد. همچنین لازم به ذکر است که Razor Pages نیز می‌توانند دارای فایل‌های لی‌اوت و \_ViewImports باشند:



همچنین می‌شود درون Razor Pages از سرویس‌هایمان نیز استفاده کرد:

@page

@inject IGreeter Greeter

<div>

@Greeter.GetMessageOfTheDay()

</div>

برای Razor Pages امکان تعریف مدل را نیز خواهیم داشت؛ تعریف مدل را می‌توانیم درون کد code-behind مربوط به Razor Page انجام دهیم:

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace OdeToFood.Pages

{

public class GreetingModel : PageModel

{

public void OnGet()

{

}

}

}

بنابراین کلاس GreetingModel به کلاسی وهله‌سازی شده‌ایی تبدیل خواهد شد که درون Razor Pageمان در دسترس خواهد بود.

مثال:

using Microsoft.AspNetCore.Mvc.RazorPages;

using OdeToFood.Services;

namespace OdeToFood.Pages

{

public class GreetingModel : PageModel

{

**private IGreeter \_greeter;**

**public string CurrentGreeting { get; set; }**

public GreetingModel(**IGreeter greeter**)

{

**\_greeter = greeter;**

}

**public void OnGet()**

**{**

**CurrentGreeting = \_greeter.GetMessageOfTheDay();**

**}**

}

}

نکته: کار متد **OnGet** چیست؟

وقتی Razor Pageمان یک درخواست HTTP GET را دریافت کند، فریم‌ورک MVC مدل آن را (در اینجا GreetingModel) را وهله‌سازی کرده و تمامی سرویس‌های موردنیاز را تزریق خواهد کرد، سپس متد OnGet را فراخوانی خواهد کرد. بنابراین درون این متد می‌توانید تمامی اطلاعات موردنیاز مدل را populate کنیم.

استفاده از مدل فوق درون Razor Page:

@page

@model GreetingModel

<div>@Model.CurrentGreeting</div>

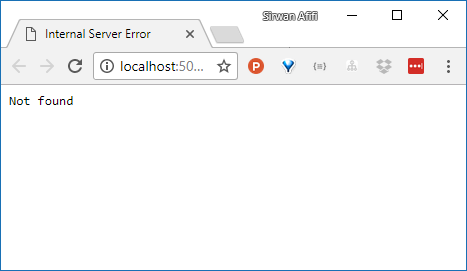
توسط Razor Pages می‌توانیم named parameter را نیز برای درخواست‌های GET تعیین کنیم:

@page "{name}"

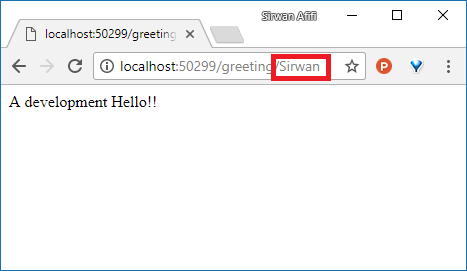
@model GreetingModel

<div>@Model.CurrentGreeting</div>

بنابراین اکنون با مراجعه به page فوق خواهیم داشت:



در واقع اکنون گفته‌ایم که این page باید دارای یک پارامتر name باشد:



اکنون می‌توانیم از این پارامتر درون متد OnGet استفاده کنیم:

public void OnGet(**string name**)

{

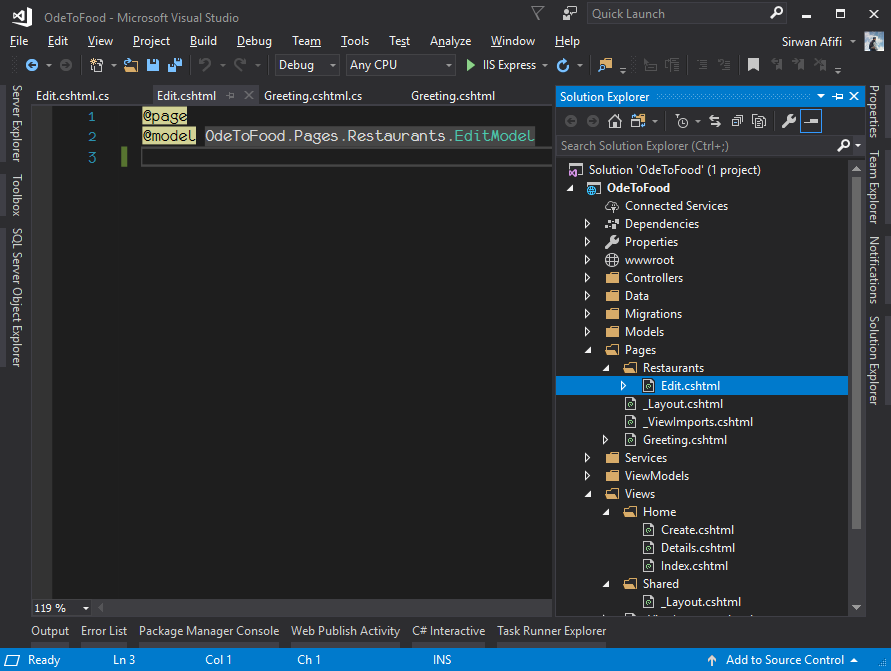
CurrentGreeting = $"{name} {\_greeter.GetMessageOfTheDay()}";

}

## An Edit Form

اضافه کردن قابلیت ویرایش رستوران توسط Razor Pages:

**قدم اول:** افزودن پوشه Restaurants درون پوشه Pages



**قدم دوم:** تعریف پارامتر id برای page ایجاد شده:

**@page "{id}"**

@model OdeToFood.Pages.Restaurants.EditModel

**قدم سوم:** افزودن لینک Edit:

<a **asp-page**="/Restaurants/Edit" asp-route-id="@restaurant.Id">Edit</a>

**قدم چهارم:** درون متد OnGet باید آیتم موردنظر را از دیتابیس استخراج کنیم و درون صفحه نمایش دهیم:

using Microsoft.AspNetCore.Mvc.RazorPages;

using OdeToFood.Models;

using OdeToFood.Services;

namespace OdeToFood.Pages.Restaurants

{

public class EditModel : PageModel

{

**private IRestaurantData \_restaurantData;**

**public Restaurant Restaurant { get; set; }**

public EditModel(**IRestaurantData restaurantData**)

{

\_restaurantData = restaurantData;

}

public void OnGet(int id)

{

Restaurant = \_restaurantData.Get(id);

}

}

}

**اگر شیء Restaurant نال بود چکار باید بکنیم؟ در واقع خروجی متد OnGet یک IActionResult است؛ در نتیجه می‌توانیم یک redirect action استفاده کنیم:**

public IActionResult OnGet(int id)

{

Restaurant = \_restaurantData.Get(id);

**if (Restaurant == null)**

**{**

**return RedirectToAction("Index", "Home");**

**}**

**return Page();**

}

**قدم پنجم:** نمایش آیتم انتخاب شده درون پیج Edit:

@page "{id}"

@model OdeToFood.Pages.Restaurants.EditModel

<h1>@Model.Restaurant.Name</h1>

<form method="post">

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

<div>

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

<input asp-for="Restaurant.Name" />

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

</div>

<div>

<label asp-for="Restaurant.Cuisine"></label>

<select asp-for="Restaurant.Cuisine"

asp-items="@Html.GetEnumSelectList<Cuisine>()"></select>

<span asp-validation-for="Restaurant.Cuisine"></span>

</div>

<input type="submit" name="save" value="Save" />

</form>

**همانطور که مشاهده می‌کنید برای سورس asp-items از HTML Helpers استفاده کرده‌ایم؛ در واقع در بیشتر اوقات از Tag Helper به همراه HTML Helpers استفاده می‌کنیم.**

**قدم ششم:** نسخه‌ی Post اکشن متد Edit:

public IActionResult OnPost()

{

if (ModelState.IsValid)

{

\_restaurantData.Update(Restaurant);

return RedirectToAction("Details", "Home", new { id = Restaurant.Id });

}

return Page();

}

**نکته: همانطور که مشاهده می‌کنید برخلاف حالت اکشن‌متد در MVC، در اینجا شیء موردنظر را به متد OnPost ارسال نکرده‌ایم، زیرا این شیء را در حال‌حاضر در اختیار داریم (پراپرتی پابلیک Restaurant) اما در اینحالت باید اتریبیوت BindProperty را به این پراپرتی اضافه کنیم؛ با کمک این اتربیوت گفته‌ایم که اطلاعات وارد شده از طریق درخواست POST را دریافت کرده و آن را به پراپرتی موردنظر بایند کن:**

**[BindProperty]**

public Restaurant Restaurant { get; set; }

همانطور که مشاهده می‌کنید برای ثبت تغییرات یک متد با نام Update را به اینترفیس IRestaurantData اضافه کرده‌ایم:

Restaurant Update(Restaurant restaurant);

پیاده‌سازی متد فوق:

public Restaurant Update(Restaurant restaurant)

{

\_context.Attach(restaurant).State =

Microsoft.EntityFrameworkCore.EntityState.Modified;

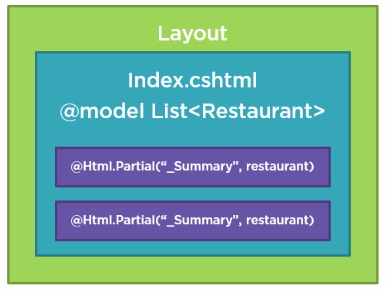
\_context.SaveChanges();

return restaurant;

}

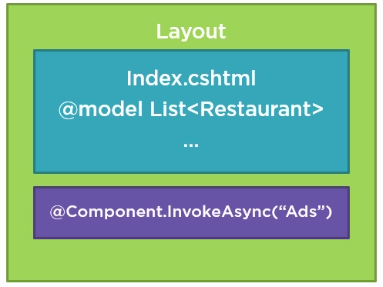
## Partial Views

دو حالت برای استفاده از Partial Views وجود دارد؛ فرض کنید یک Layout View دارید؛ این ویو توسط ویوی Index جهت رندر شدن انتخاب خواهد شد؛ ویوی Index نیز یک مدل را از کنترلر دریافت خواهد کرد؛ مدل نیز به Partial View جهت رندر شدن ارسال خواهد شد:



وقتی از Html.Partial جهت رندر کردن یک partial view استفاده می‌کنید، این partial view برای رندر کردن ویو به مدل parent نیاز خواهد داشت؛ Partial View نمی‌تواند به صورت مستقل اطلاعات را دریافت کند.

اگر نیاز داشته باشید مدلی دیگری را جهت رندر شدن به partial view ارسال کنید می‌توانید از View Components استفاده کنید:



از **@Component.InvokeAsync()** در هر جایی می‌توانیم استفاده کنیم. برخلاف Partial Views، ویو کامپوننت‌ها متکی بر parent view نیستند.

ایجاد یک Partial Views – درون صفحه Index می‌خواهیم لیست رستوران‌ها توسط Partial View رندر کنیم:

@using OdeToFood.Models

@model Restaurant

<section>

<h3>@Model.Name</h3>

<div>

Cuisine: @Model.Cuisine

</div>

<div>

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

<a asp-page="/Restaurants/Edit" asp-route-id="@Model.Id">Edit</a>

</div>

</section>

نحوه‌ی استفاده:

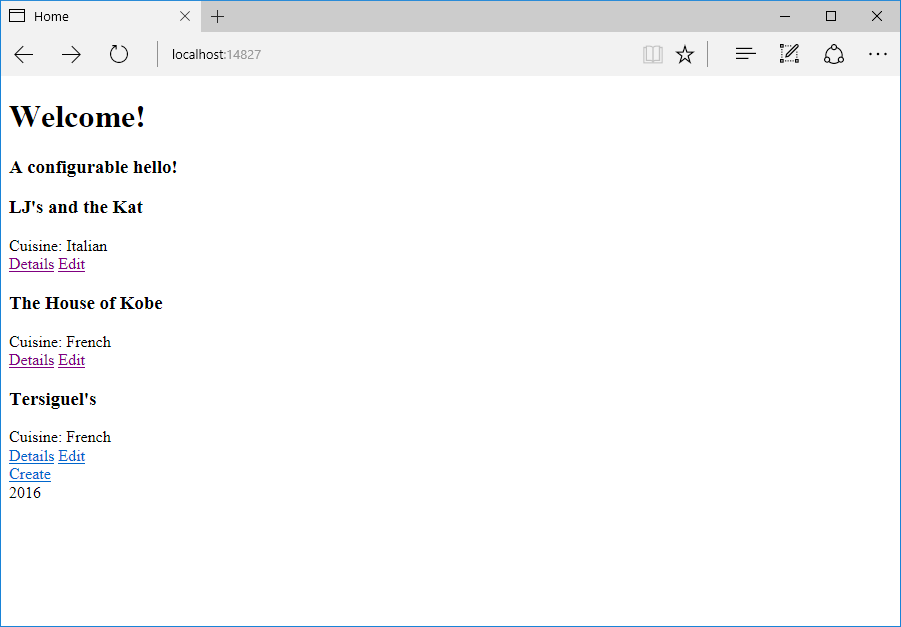
@foreach (var restaurant in Model.Restaurants)

{

@Html.Partial("\_Summary", restaurant)

}

خروجی:



**نکته: همانطور که عنوان شد، Partial Views همیشه به یک مدل نیاز خواهند داشت، اما View Components نیازی به مدل ندارند.**

## View Components

ویو کامپوننت به صورت کاملاً مستقل از کنترلر و ویو عمل می‌کند. به عنوان مثال درون \_Layout می‌توانیم اینگونه یک ویو کامپوننت را فراخوانی کنیم:

<footer>

**@await Component.InvokeAsync("Greeter")**

</footer>

نکته: می‌توانیم همچنین از سینتکس زیر برای نمایش ویوکامپوننت استفاده کنیم:

**<vc:greeter></vc:greeter>**

برای اینکه حالت فوق کار کند باید یک تگ‌هلپر دیگر را به فایل \_ViewImports اضافه کنیم:

**@addTagHelper \*, OdeToFood**

اینکار باعث خواهد شد که تمامی المنت‌های سفارشی پروژه‌ی OdeToFood برای تمامی ویوها در دسترس باشند.

**نکته: قبلاً از @Html.Action برای نمایش خروجی یک اکشن متد استفاده می‌کردیم؛ اما در این نسخه از فریم‌ورک این متد وجود ندارد و بهتر است از View Components به جای آن استفاده شود.**

تعریف ویو کامپوننت Greeting:

using Microsoft.AspNetCore.Mvc;

using OdeToFood.Services;

namespace OdeToFood.ViewComponents

{

public class GreeterViewComponent : ViewComponent

{

private IGreeter \_greeter;

public GreeterViewComponent(IGreeter greeter)

{

\_greeter = greeter;

}

public IViewComponentResult Invoke()

{

var model = \_greeter.GetMessageOfTheDay();

return View("Default", model);

}

}

}

**نکته: اگر ورودی تابع View یک رشته باشد، MVC فرض خواهد کرد که ما نام View را ارسال کرده‌ایم؛ بنابراین برای حل این مشکل باید بگوئیم که منظورمان از رشته مدل‌مان است؛ بنابراین پارامتر اول را نام ویو قرار خواهیم داد و پارامتر دوم را برابر با مدل قرار خواهیم داد.**

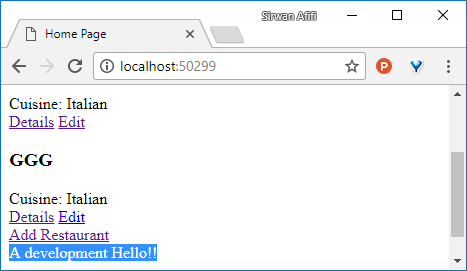
طبق convention محل ویو فوق باید در مسیر **/Shared/Components/Greeting/Default.cshtml** است.

محتویات ویو کامپوننت Greeting:

@model string

<div>@Model</div>

خروجی:



# ASP.NET Identity

## Authentication Services and Middlewares

**قدم اول:** نصب Middleware – این میان‌افزار را بعد از StaticFiles و قبل از Mvc اضافه کنید:

app.UseStaticFiles();

**app.UseAuthentication();**

app.UseMvc(ConfigureRoute);

**قدم دوم:** افزودن سرویس‌های Identity:

services.AddAuthentication(options =>

{

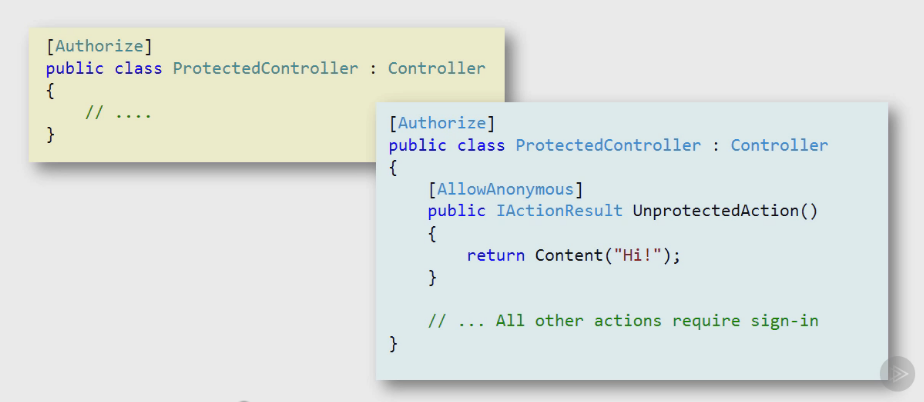
options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;

})

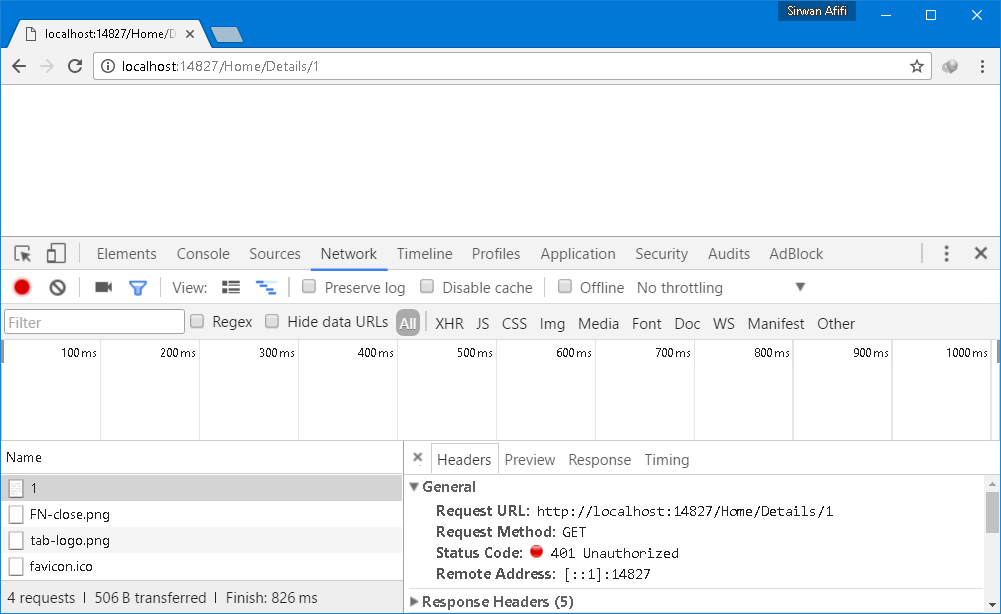
.AddCookie();

نکته: توسط **DefaultScheme** نحوه‌ی authenticate کردن کاربران را تعیین کرده‌ایم؛ این نوع می‌تواند کوکی، توکن و... باشد. در اینجا ما از کوکی استفاده کرده‌ایم.

## Using the Authorize Attribute



در این مرحله اگر به اکشن‌متدی که با ویژگی Authorize مزین شده باشد مراجعه کنیم، خروجی زیر را دریافت خواهیم کرد:

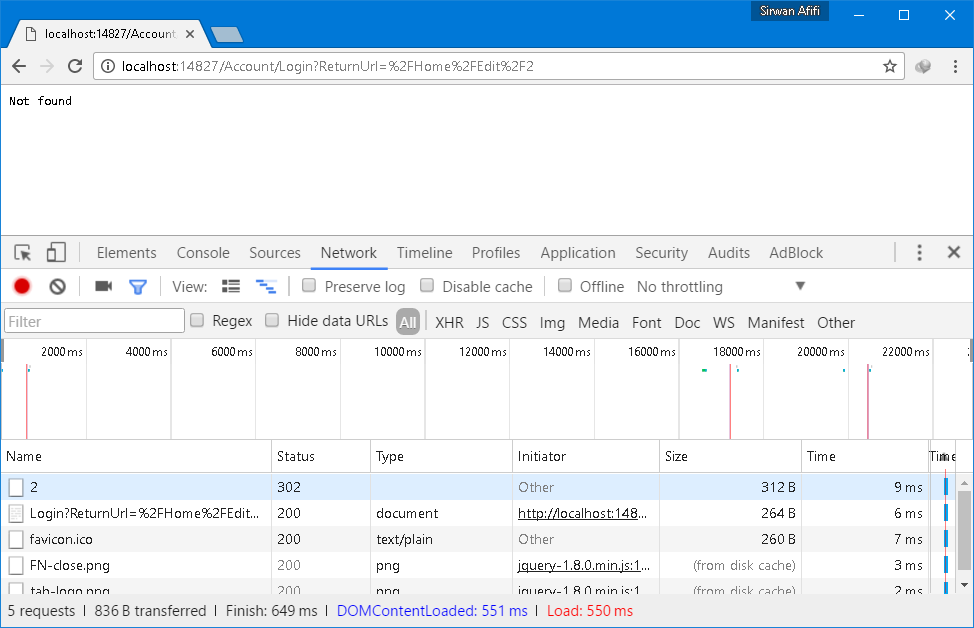


در اینحالت است که Identity Framework به ما کمک خواهد کرد؛ این فریم‌ورک خروجی 401 را در اپلیکیشن شناسایی کرده و آن را تبدیل به یک صفحه لاگین خواهد کرد؛ سپس کاربر می‌تواند از طریق این صفحه لاگین از مانع فوق (خطای 401) عبور کند.

برای اتریبیوت Authorize می‌توانیم پراپرتی‌ایی با نام Policy را نیز ست کنیم، توسط این پراپرتی می‌توانیم یک named policy تعریف کنیم، فرض کنید یک پالیسی با نام IsAdmin تعریف کرده‌ایم، این پالسی می‌تواند حاوی تمامی چک‌های موردنیاز برای کاربر جاری باشد. نه تنها می‌توان بررسی کرد که کاربر authenticate شده است بلکه می‌توانیم اطلاعاتی دیگری را نیز چک کنیم.

## User Registration

اکنون که اسکیمای دیتابیس‌مان آپدیت شده است، با مراجعه به صفحه‌ی Authorize شده، خروجی not found را مشاهده خواهیم کرد؛ همچنین آدرس صفحه نیز تغییر خواهد کرد:



همانطور که مشاهده می‌کنید اینبار به جای status code 404 اینبار 302 status code را دریافت کرده‌ایم؛ یعنی صفحه ریدایرکت شده است. همچنین آدرس پیش‌فرض برای لاگین نیز /Account/Login است که می‌توانیم آن را تغییر دهیم. از آنجائیکه این کنترلر و اکشن متد را نداریم، این درخواست توسط MVC هندل نخواهد شد؛ در نتیجه به آخرین middleware ثبت شده درون pipeline که خروجی not found را نمایش می‌دهد رسیده‌ایم.

**ایجاد کنترلر Account**



**ایجاد ویو مدل ثبت‌نام کاربران**



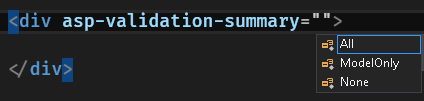
**ایجاد ویو ثبت‌نام**



نکته: همانطور که مشاهده می‌کنید در ویو فوق از عبارت using برای ایمپورت کردن namespace حاوی ویو مدل استفاده نکرده‌ایم؛ زیرا اینکار را یکبار برای تمامی ویوها درون فایل \_ViewImports.cshtml انجام داده‌ایم:



نکته: asp-validation-summary یکی از سه نوع enum زیر را می‌پذیرد:



* All: تمامی خطاهای مدل نمایش داده شوند.
* ModelOnly: خطاهای مربوط به یک پراپرتی خاص را نمایش نمی‌دهد.
* None: هیچ خطایی نمایش داده نشود.

**ایجاد نمونه‌ Post اکشن متد ثبت‌نام**



در قسمت بعد به نحوه‌ی پیاده‌سازی اکشن متد فوق خواهیم پرداخت.

## Creating a User

**اکشن متد Register**

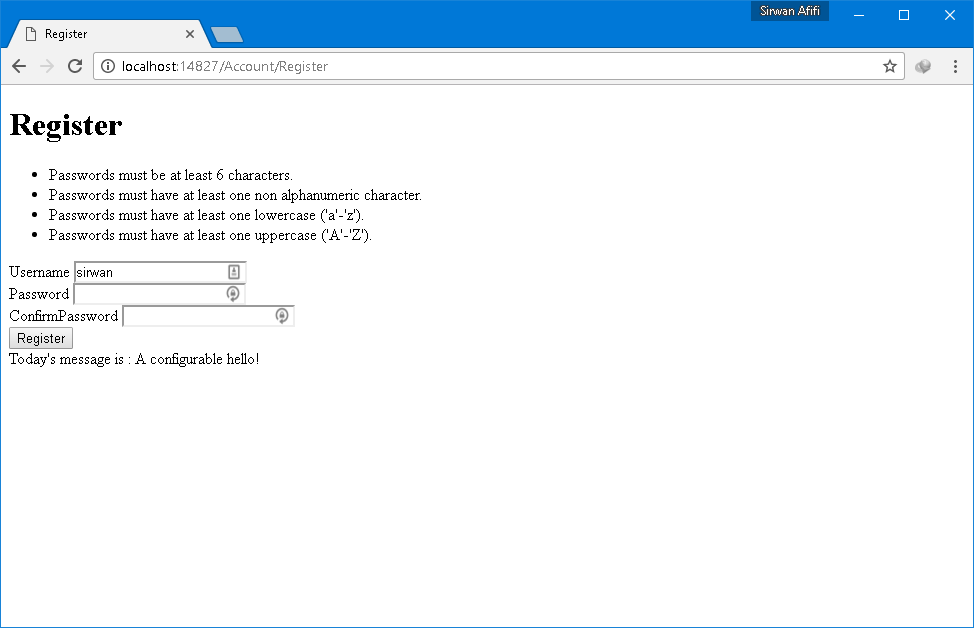


نکته: اگر نتیجه‌ی یک کاربر false بود؛ یعنی Succeed برابر با false بود؛ تمامی خطاهای درون پراپرتی Errors را به ModelState اضافه کرده‌ایم. پراپرتی اول متد AddModelError نام پراپرتی‌ایی است که می‌توانیم خطا را به آن مرتبط (associate) کنیم:

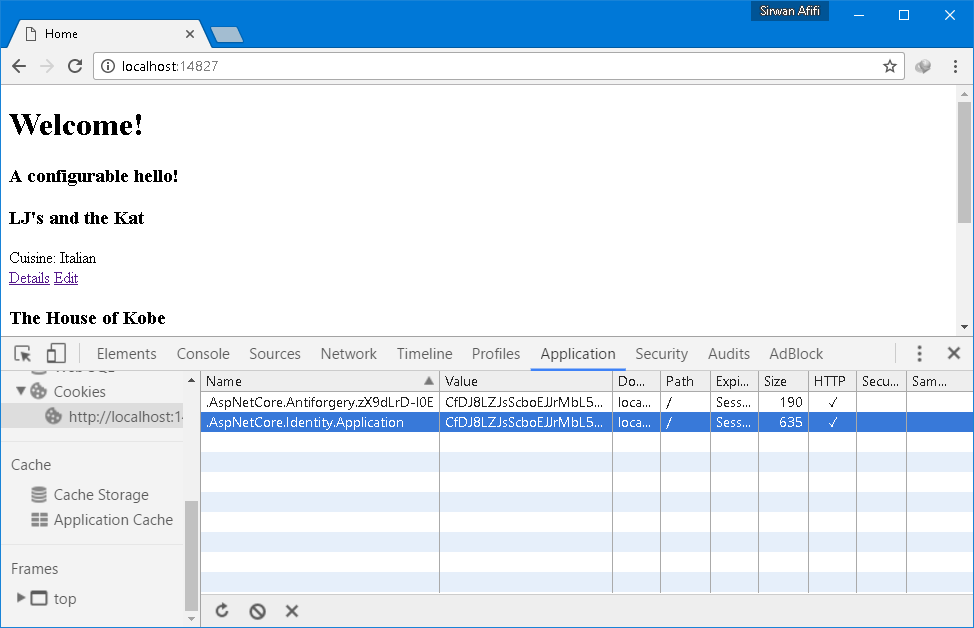


اما چون نمی‌خواهیم اینکار را انجام دهیم؛ آن را خالی رها کرده‌ایم؛ در اینحالت خطا مربوط به مدل است نه یک پراپرتی خاص از مدل:





با وارد کردن یک پسورد معتبر کاربر ایجاد شده و کوکی موردنظر نیز ذخیره خواهد شد:



همانطور که مشاهده می‌کنید authentication ticket با موفقیت ایجاد شده است؛ مرورگر نیز این تیکت را به همراه تمامی درخواست‌ها ارسال خواهد کرد؛ بنابراین اینگونه مطمئن خواهد شد که به سایت لاگین کرده‌ایم.

## Log in and Log Out

درون فایل \_Layout.cshtml به کامپوننتی جهت لاگین و لاگ‌اوت نیاز داریم:



**تعریف کامپوننت LoginLogout**



**ویو کامپوننت فوق**



**اکشن متد Logout**



**ایجاد نسخه‌ی GET اکشن متد Login**



**ایجاد View Model لاگین**

****

**ویوی اکشن متد Login**



**ایجاد نسخه‌ی POST اکشن متد لاگین**



نکته: دلیل استفاده از متد IsLocalUrl جلوگیری از آسیب‌پذیری [open redirect](http://whatis.techtarget.com/definition/open-redirect) است.

# Identities and Claims

پراپرتی User علاوه بر کنترلر درون ویوها نیز قابل دسترسی است، فرض کنید می‌خواهیم تمامی claimهای کاربر را نمایش دهیم؛ منظور از claims اطلاعات اضافه‌تری از کاربر است:

@if (User.Identity.IsAuthenticated)

{

foreach (var identity in User.Identities)

{

<h3>@identity.Name</h3>

<ul>

@foreach (var claim in identity.Claims)

{

<li>@claim.Type - @claim.Value</li>

}

</ul>

}

}

else

{

<div>You are anonymouse</div>

}

# Front End Frameworks and Tools

## Introduction

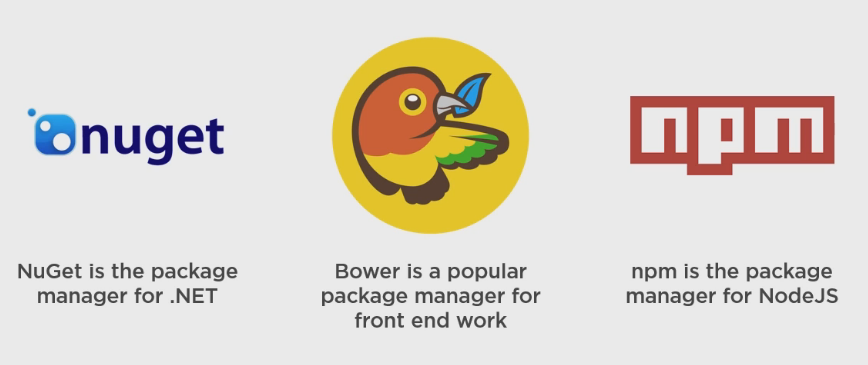
در این ماژول به بررسی کتابخانه‌های CSS and JavaScript خواهیم پرداخت؛ در واقع به بررسی ابزارهایی که برای نصب این کتابخانه‌ها استفاده می‌شوند خواهیم پرداخت.

## Front End Tools

در سالیان اخیر نیوگت یک پکیج منیجر برای دات‌نت بوده است. برای مقطع خاصی نیوگت در ASP.NET Core تنها برای مدیریت پکیج‌های دات‌نتی استفاده می‌شد اما تیم وعده داده است که Content Packages را نیز پشتیبانی کند.

برای نصب پکیج‌های فرنت‌اند bower گزینه‌ی خوبی است اما به دلیلی که در ادامه توضیح خواهیم داد نمی‌خواهیم از آن استفاده کنیم.

**Because one tool with one configuration file is easier than using two tools with two configuration files.**



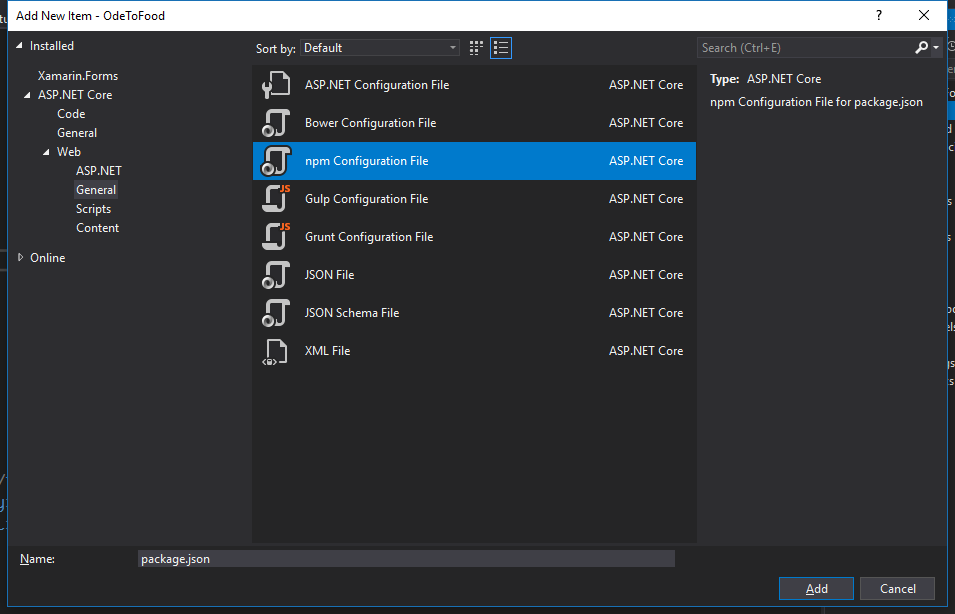
## Command Line vs. Visual Studio

در یک پروژه‌ی ASP.NET دو روش برای کار با NPM وجود دارد.

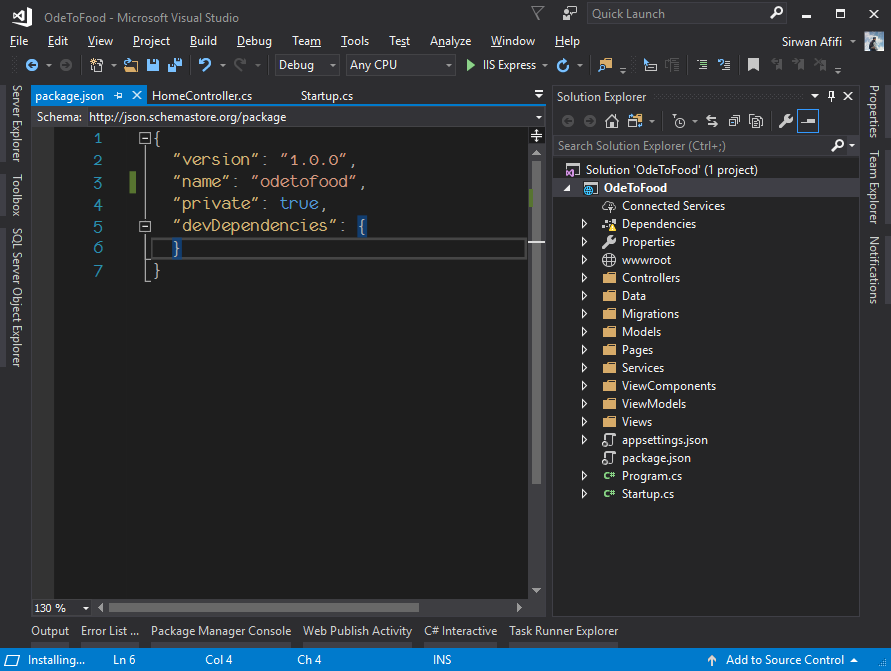
* Use tooling and support provided by Visual Studio
* Use from the Command Line

## Setting up npm

درن ویژوال استدیو کار با NPM به سادگی کار با project.json است؛ project.json کار مدیریت وابستگی‌های سمت سرور را برایمان انجام می‌دهد. فایل package.json نیز کار مدیریت پکیج‌های سمت کلاینت توسط NPM را برایمان انجام خواهد داد:



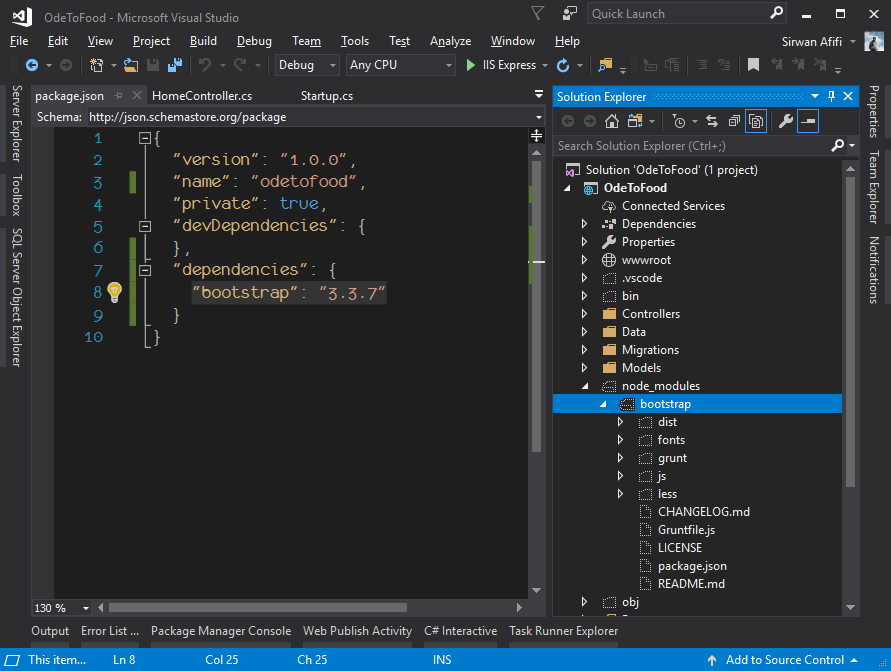
با تغییر این فایل و ذخیره آن، پوشه‌ی npm به قسمت Dependencies اضافه خواهد شد؛ ویژال استدیو تشخیص داده است که از npm می‌خواهیم استفاده کنیم:



برای نصب پکیج‌های مورد نظر می‌توانیم همانند فایل‌های csproj عمل کنیم:



اما سوال اینجاست که فایل bootstrap در کجا ذخیره شده است؟



در قسمت بعد نحوه‌ی serve کردن دایرکتیوری node\_modules را بررسی خواهیم کرد.

## Serving File from node\_modules

تا اینجا تنها محتویات درون دایرکتوری wwwroot توسط مرورگر serve خواهند شد؛ برای serve کردن محتویات داخل دایرکتوری node\_modules باید به صورت زیر عمل کنیم.

اگر به خاطر داشته باشید توسط middleware زیر توانستیم یک File Server را جهت serve کردن محتویات داخل دایرکتوری wwwroot ایجاد کنیم:

app.UseFileServer();

میان‌افزار فوق در پشت صحنه از StaticFiles استفاده خواهد کرد. در ادامه‌ی کد فوق StaticFiles را برای serve کردن دایرکتوری wwwroot پیکربندی خواهیم کرد؛ برای جلوگیری از شلوغ شدن (cluttering up) فایل Startup کلاس زیر را درون یک دایرکتوری جدید با نام Middleware در ریشه‌ی پروژه قرار خواهیم داد:

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.FileProviders;

using System.IO;

namespace Microsoft.AspNetCore.Builder

{

public static class ApplicationBuilderExtension

{

public static IApplicationBuilder UseNodeModules(

this IApplicationBuilder app, string root)

{

var path = Path.Combine(root, "node\_modules");

var provider = new PhysicalFileProvider(path);

var options = new StaticFileOptions();

options.RequestPath = "/node\_modules";

options.FileProvider = provider;

app.UseStaticFiles(options);

return app;

}

}

}

همانطور که مشاهده می‌کنید فضای نام را به **Microsoft.AspNetCore.Builder** تغییر داده‌ایم؛ در واقع مرسوم است زمانیکه یک متد الحاقی برای IApplicationBuilder طراحی می‌کنید، بهتر است آن را در همان فضای نامی قرار دهید که این اینترفیس در آن واقع شده است.

برای استفاده از متد الحاقی فوق:

**app.UseNodeModules(env.ContentRootPath);**

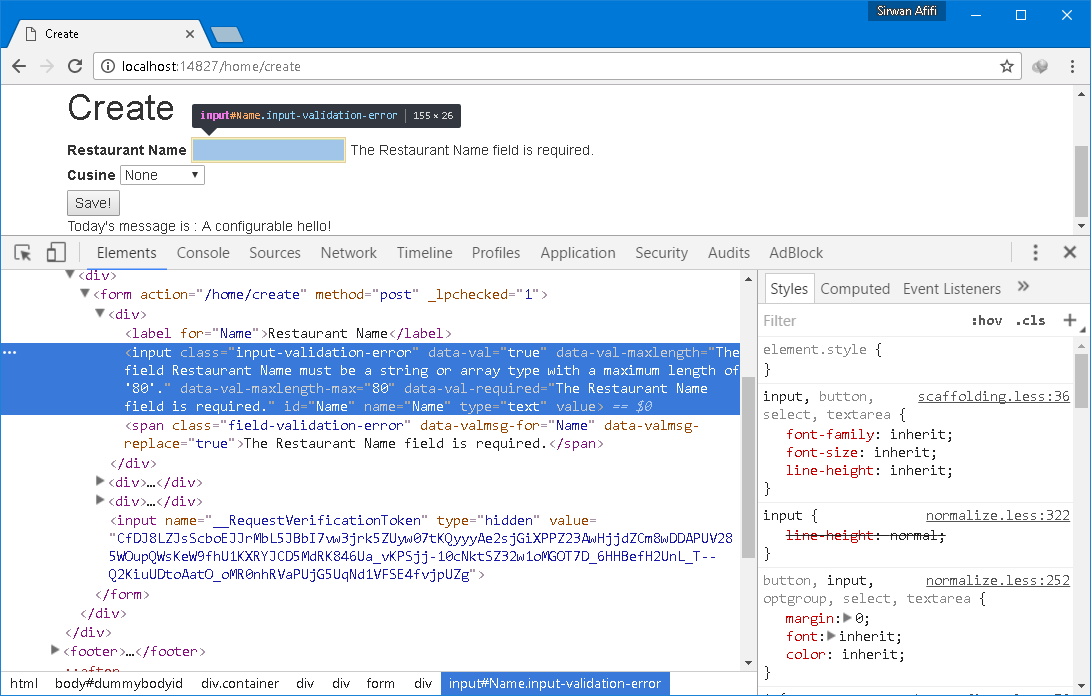
البته بهتر است از یک task runner مانند gulp یا grunt به عنوان یک build system جهت انتقال فایل‌ها از node\_modules به wwwroot استفاده کنید.

اکنون می‌توانیم از فایل‌های داخل دایرکتوری node\_modules استفاده کنیم:

<link href="/node\_modules/bootstrap/dist/css/bootstrap.css" rel="stylesheet"/>

## Enabling Client-side Validation

درون فرم Create در حالت وجود خطا برای مدل اگر صفحه را inspect کنید خواهید دید که تعدادی اتربیوت سفارشی به inputها اضافه شده است:



اینها در واقع توسط فریم‌ورک MVC براساس Data Annotationsهایی که برای مدل تعیین کردیم، تولید شده‌اند. مرورگر نمی‌داند با این اتریبیوت‌های تولید شده چه کاری انجام دهد، اما ایده اصلی این است که این اتریبیوت‌ها توسط جاوا اسکریپت مورداستفاده قرار گیرند. اینکار را می‌توانیم توسط کتابخانه‌ایی با نام jQuery Validation انجام دهیم.

پیش‌نیاز لازم برای استفاده از jQuery Validation

**قدم اول: نصب وابستگی‌های زیر از طریق NPM(package.json)**



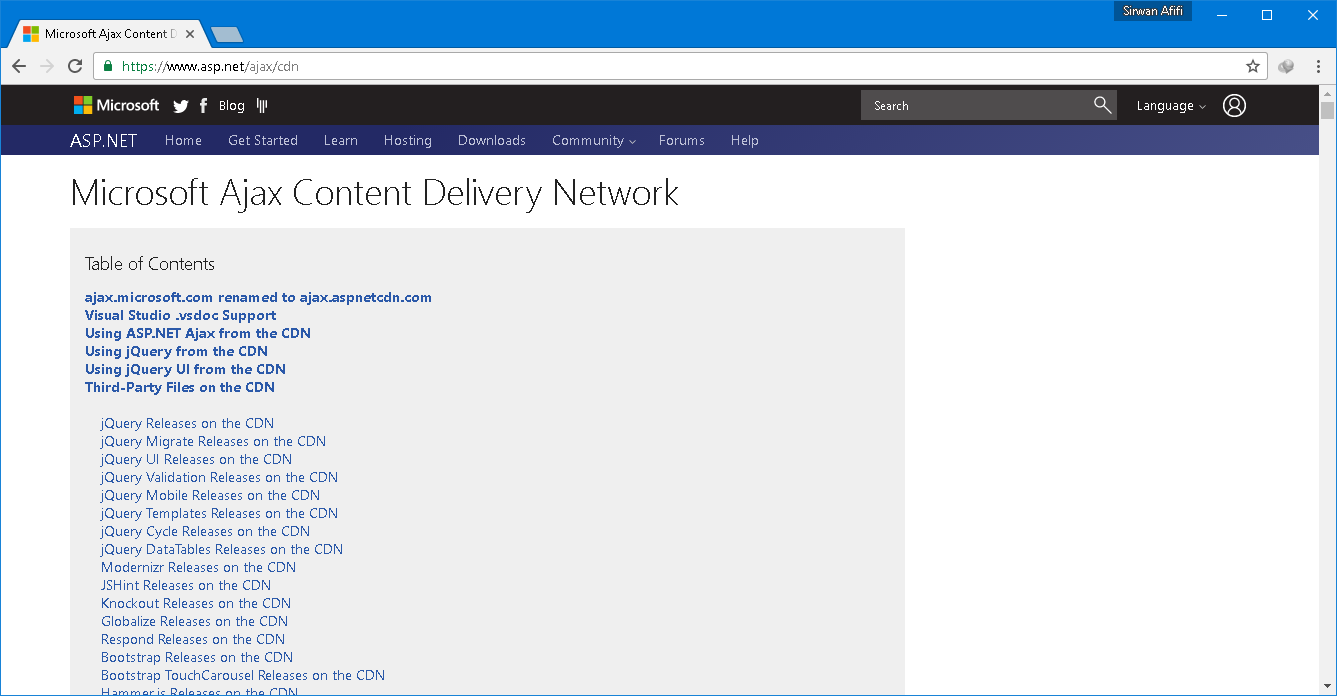
**ارجاع به وابستگی‌های فوق درون فایل لی‌اوت**



اکنون با کلیک بر روی دکمه Save جهت نمایش خطاهای مدل، نیاز به post back نیست؛ بلکه اعتبارسنجی در سمت کلاینت انجام خواهد شد.

## Using CDNs and Fallbacks

برای اپلیکیشن‌هایی که حجم زیادی از جاوا اسکریپت استفاده می‌کنند، داشتن یک build process برای جاوا اسکریپت است؛ یعنی استفاده از یک task runner مانند gulp و... به طوریکه فایل‌های جاوا اسکریپت را به صورت minify شده تحویل کلاینت دهیم. روش دیگر قرار دادن اسکریپت‌ها در CDNs است.



اما در بیشتر اوقات از اسکریپت‌های لوکال برای development و در حالت production از اسکریپت‌های موجود برروی CDN استفاده می‌شود. اینکار را می‌توانیم توسط Tag Helper زیر مدیریت کنیم:

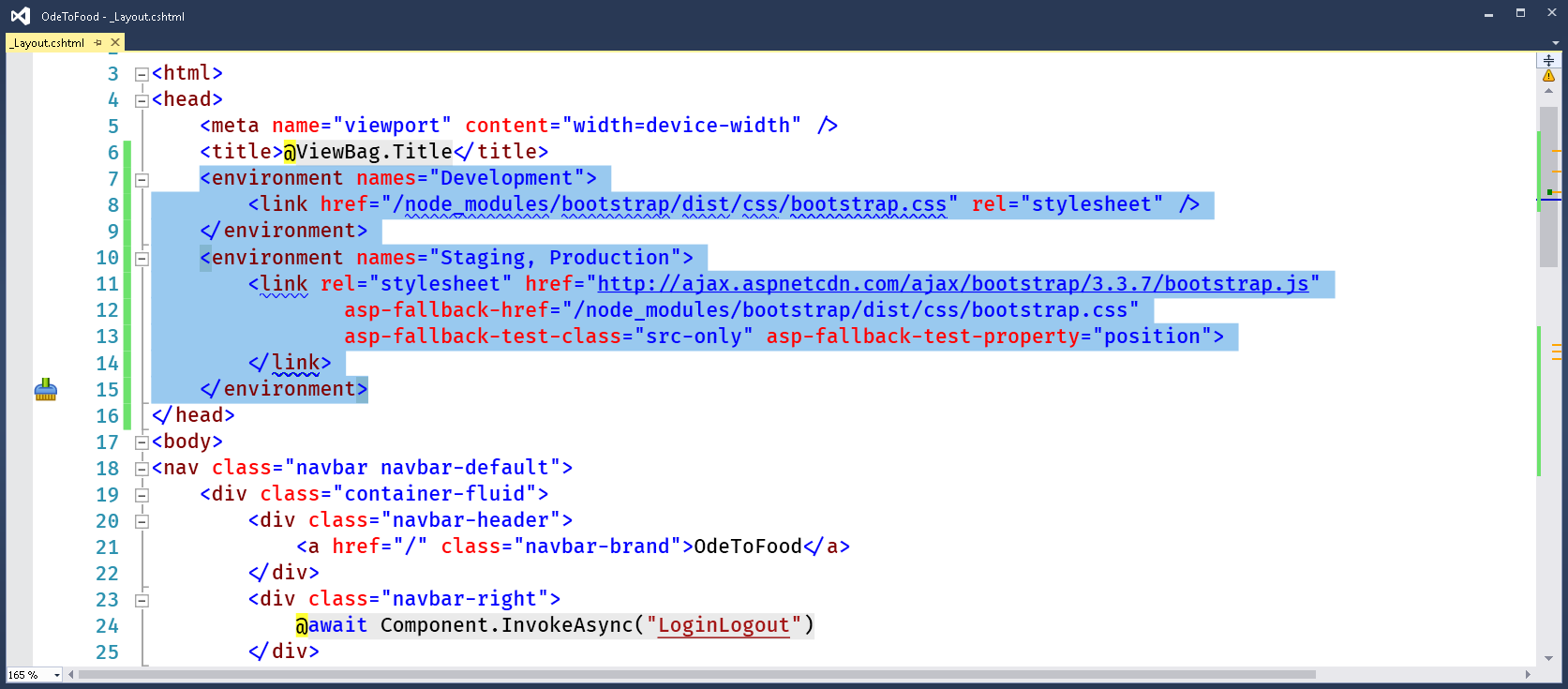


همانطور که مشاهده می‌کنید در حالت Staging and Production and Development اسکریپت‌های لازم را بارگذاری کرده‌ایم. در حالت‌های Staging and Production دو اتریبیوت داریم:

* asp-fallback-src
* asp-fallback-test

اگر تحت شرایطی CDN داون شود؛ اسکریپت موردنظر از آدرس تعیین شده در asp-fallback-src بارگذاری خواهد شد. توسط asp-fallback-test نیز شروط لازم از اینکه اسکریپت موردنظر با موفقیت بارگذاری شده است، انجام شده است. یعنی اگر شروط تعیین شده fail شود، اسکریپت از آدرس تعیین شده در asp-fallback-src بارگذاری خواهد شد.

لازم به ذکر است که برای استایل‌ها نیز می‌توانیم همینکار را انجام دهیم:



در اینحالت اپلیکیشن‌مان باید سریع‌تر از حالت قبل باشد.