

Table of Contents

[介绍](#)

[入门](#)

[创建 Web 应用](#)

[创建 Web API](#)

[教程](#)

[创建 Razor 页面 Web 应用](#)

[Razor 页面入门](#)

[添加模型](#)

[已搭建基架的 Razor 页面](#)

[SQL Server LocalDB](#)

[更新页面](#)

[添加搜索](#)

[添加新字段](#)

[添加验证](#)

[上载文件](#)

[创建 MVC Web 应用](#)

[入门](#)

[添加控制器](#)

[添加视图](#)

[添加模型](#)

[使用 SQL Server LocalDB](#)

[控制器方法和视图](#)

[添加搜索](#)

[添加新字段](#)

[添加验证](#)

[检查 Details 和 Delete 方法](#)

[生成 Web API](#)

[在 Visual Studio Code 中创建 Web API](#)

[在 Visual Studio for Mac 中创建 Web API](#)

[在 Visual Studio for Windows 中创建 Web API](#)

[为本机移动应用创建后端服务](#)

[使用 Swagger 的帮助页](#)

[数据访问 - 使用 EF Core](#)

[数据访问 - 使用 Razor 页面和 EF Core](#)

[数据访问 - MVC 与 EF Core](#)

[跨平台教程](#)

[macOS 上的 Razor 页面 Web 应用](#)

[使用 VS 代码的 Razor 页面 Web 应用](#)

[使用 Visual Studio for Mac 的 MVC Web 应用](#)

[在 macOS 或 Linux 上使用 Visual Studio Code 创建 MVC Web 应用](#)

[使用 Visual Studio for Mac 创建 Web API](#)

[使用 Visual Studio Code 创建 Web API](#)

[为移动应用创建后端服务](#)

[基础知识](#)

[应用程序启动](#)

[依赖关系注入\(服务\)](#)

[中间件](#)

[中间件](#)

[基于工厂的中间件](#)

[Factory-based middleware with third-party container\(第三方容器中基于工厂的中间件\)](#)

[静态文件](#)

[路由](#)

[URL 重写中间件](#)

[使用多个环境](#)

[配置和选项](#)

[配置](#)

[选项](#)

[从外部程序集增强应用](#)

[日志记录](#)

[日志记录与 LoggerMessage](#)

[处理错误](#)

[文件提供程序](#)

[承载](#)

[会话和应用状态](#)

[服务器](#)

[Kestrel](#)

[ASP.NET Core 模块](#)

[HTTP.sys](#)

[全球化和本地化](#)

[使用 Orchard Core 配置可移植对象本地化](#)

[启动 HTTP 请求](#)

[请求功能](#)

[使用托管服务的后台任务](#)

[基元](#)

[更改令牌](#)

[.NET 的开放 Web 接口 \(OWIN\)](#)

[WebSockets](#)

[Microsoft.AspNetCore.All metapackage](#)

[在 .NET Core 和 .NET Framework 之间进行选择](#)

[在 ASP.NET Core 和 ASP.NET 之间进行选择](#)

[Razor 页面](#)

[Razor 页面的筛选方法](#)

[创建一个 Razor 类库](#)

[路由和应用约定](#)

[Razor SDK](#)

[MVC](#)

[模型绑定](#)

[模型验证](#)

[视图](#)

[Razor 语法](#)

[视图编译](#)

[布局](#)

[标记帮助程序](#)

- [部分视图](#)
- [视图中的依赖关系注入](#)
- [视图组件](#)
- [控制器](#)
 - [路由到控制器操作](#)
 - [文件上传](#)
 - [控制器中的依赖关系注入](#)
 - [测试控制器](#)
 - [高级](#)
 - [使用应用模型](#)
 - [筛选器](#)
 - [区域](#)
 - [应用程序部件](#)
 - [自定义模型绑定](#)
- [Web API](#)
 - [控制器操作返回类型](#)
 - [高级](#)
 - [自定义格式化程序](#)
 - [格式化响应数据](#)
- [测试、调试和疑难解答](#)
 - [单元测试](#)
 - [集成测试](#)
 - [Razor 页面测试](#)
 - [测试控制器](#)
 - [远程调试](#)
 - [快照调试](#)
 - [Visual Studio 中的快照调试](#)
 - [疑难解答](#)
- [使用 EF Core 和 Azure 的数据访问](#)
 - [通过 Visual Studio 开始使用 Razor 页面和 EF Core](#)
 - [通过 Visual Studio 开始使用 ASP.NET Core 和 EF Core](#)
 - [ASP.NET Core 和 EF Core - 新数据库](#)

[ASP.NET Core 和 EF Core - 现有数据库](#)

[开始使用 ASP.NET Core 和 Entity Framework 6](#)

[Azure 存储](#)

[使用 Visual Studio 连接服务添加 Azure 存储](#)

[开始使用 Blob 存储和 Visual Studio 连接服务](#)

[开始使用队列存储和 Visual Studio 连接服务](#)

[开始使用表存储和 Visual Studio 连接服务](#)

[客户端开发](#)

[使用 Gulp](#)

[使用 Grunt](#)

[使用 Bower 管理客户端包](#)

[使用 Bootstrap 构建响应式站点](#)

[使用 LESS、Sass 和 Font Awesome 为应用设置样式](#)

[捆绑和缩小](#)

[使用浏览器链接](#)

[对 SPA 使用 JavaScriptServices](#)

[使用 SPA 项目模板](#)

[Angular 项目模板](#)

[React 项目模板](#)

[带 Redux 的 React 项目模板](#)

[SignalR](#)

[介绍](#)

[入门](#)

[中心](#)

[JavaScript 客户端](#)

[发布到 Azure](#)

[支持的平台](#)

[移动](#)

[为本机移动应用创建后端服务](#)

[托管和部署](#)

[托管 Azure 应用服务](#)

[使用 Visual Studio 发布到 Azure](#)

[使用 CLI 工具发布到 Azure](#)
[使用 Visual Studio 和 Git 持续部署到 Azure](#)
[使用 VSTS 持续部署到 Azure](#)
[对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)
[使用 IIS 在 Windows 上进行托管](#)
[对 IIS 上的 ASP.NET Core 进行故障排除](#)
[ASP.NET Core 模块配置参考](#)
[Visual Studio 中针对 ASP.NET Core 的开发时 IIS 支持](#)
[IIS Modules 与 ASP.NET Core](#)
[在 Windows 服务中进行托管](#)
[在 Linux 上使用 Nginx 进行托管](#)
[在 Linux 上使用 Apache 进行托管](#)
[在 Docker 中进行托管](#)
[生成 Docker 映像](#)
[Visual Studio Tools for Docker](#)
[发布到 Docker 映像](#)
[代理和负载均衡器配置](#)
[Visual Studio 发布配置文件](#)
[目录结构](#)
[Azure 应用服务和 IIS 的常见错误参考](#)

安全性

[身份验证](#)
[社区 OSS 身份验证选项](#)
[标识简介](#)
[配置标识](#)
[配置 Windows 身份验证](#)
[配置标识的主键类型](#)
[自动以标识的存储提供程序](#)
[启用使用 Facebook、Google 和其他外部提供程序的身份验证](#)
[WS 联合身份验证](#)
[帐户确认和密码恢复](#)
[在标识中启用 QR 代码生成](#)

[使用 SMS 设置双因素身份验证](#)

[在没有标识的情况下使用 cookie 身份验证](#)

[Azure Active Directory](#)

[使用 IdentityServer4 保护 ASP.NET Core 应用](#)

[使用 Azure App Service 身份验证保护 ASP.NET Core 应用\(简易身份验证\)](#)

[各个用户帐户](#)

[授权](#)

[介绍](#)

[通过授权保护的用户数据创建应用](#)

[Razor 页面授权](#)

[简单授权](#)

[基于角色的授权](#)

[基于声明的授权](#)

[基于策略的授权](#)

[要求处理程序中的依赖关系注入](#)

[基于资源的授权](#)

[基于视图的授权](#)

[使用方案限制标识](#)

[数据保护](#)

[数据保护简介](#)

[数据保护 API 入门](#)

[使用者 API](#)

[配置](#)

[扩展性 API](#)

[实现](#)

[兼容性](#)

[Enforce HTTPS](#)

[在开发期间安全存储应用机密](#)

[Azure Key Vault 配置提供程序](#)

[反请求伪造](#)

[阻止打开重定向攻击](#)

[阻止跨站点脚本编写](#)

[启用跨域请求 \(CORS\)](#)

[在应用之间共享 Cookie](#)

[性能](#)

[缓存响应](#)

[内存中缓存](#)

[使用分布式缓存](#)

[响应缓存](#)

[响应缓存中间件](#)

[响应压缩中间件](#)

[迁移](#)

[ASP.NET 到 ASP.NET Core](#)

[MVC](#)

[Web API](#)

[配置](#)

[身份验证和标识](#)

[ClaimsPrincipal.Current](#)

[共成员身份到标识](#)

[HTTP 模块到中间件](#)

[ASP.NET Core 1.x 到 2.0](#)

[身份验证和标识](#)

[API 参考](#)

[2.0 发行说明](#)

[1.1 发行说明](#)

[早期发行说明](#)

[VS 2015/project.json 文档](#)

[参与](#)

ASP.NET Core 简介

2018/4/10 • 4 min to read • [Edit Online](#)

作者: Daniel Roth、Rick Anderson 和 Shaun Luttin

ASP.NET Core 是一个跨平台的高性能开源框架，用于生成基于云且连接 Internet 的新式应用程序。使用 ASP.NET Core，您可以：

- 建置 Web 应用程式和服务、IoT 应用和移动后端。
- 在 Windows、macOS 和 Linux 上使用喜爱的开发工具。
- 部署到云或本地。
- 在 .NET Core 或 .NET Framework 上运行。

为何使用 ASP.NET Core？

数百万开发人员使用过(并将继续使用)ASP.NET 4.x 创建 Web 应用。ASP.NET Core 是重新设计的 ASP.NET 4.x，更改了体系结构，形成了更精简的模块化框架。

ASP.NET Core 具有如下优点：

- 生成 Web UI 和 Web API 的统一场景。
- 集成新式客户端框架和开发工作流。
- 基于环境的云就绪配置系统。
- 内置依赖项注入。
- 轻型的高性能模块化 HTTP 请求管道。
- 能够在 IIS、Nginx、Apache、Docker 上进行托管或在自己的进程中进行自托管。
- 定目标到 .NET Core 时，可以使用并行应用版本控制。
- 简化新式 Web 开发的工具。
- 能够在 Windows、macOS 和 Linux 进行生成和运行。
- 开放源代码和以社区为中心。

ASP.NET Core 完全作为 NuGet 包的一部分提供。借助 NuGet 包，可以将应用优化为只包含必需的依赖项。实际上，定目标到 .NET Core 的 ASP.NET Core 2.x 应用只需要使用一个 NuGet 包。较小的应用图面区域的优势包括：提升安全性、减少维护和提高性能。

使用 ASP.NET Core MVC 生成 Web API 和 Web UI

ASP.NET Core MVC 提供生成 Web API 和 Web 应用所需的功能：

- Model-View-Controller (MVC) 模式 使 Web API 和 Web 应用可测试。
- ASP.NET Core 2.0 中新增的 Razor 页面是基于页面的编程模型，可简化 Web UI 生成并提高工作效率。
- Razor 标记提供了适用于 Razor 页面和 MVC 视图的高效语法。
- 标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。
- 内置的多数据格式和内容协商支持使 Web API 可访问多种客户端，包括浏览器和移动设备。
- 模型绑定 自动将 HTTP 请求中的数据映射到操作方法参数。
- 模型验证 自动执行客户端和服务器端验证。

客户端开发

ASP.NET Core 与常用客户端框架和库(包括 Angular、React 和 Bootstrap)无缝集成。有关详细信息, 请参阅[客户端开发](#)。

面向 .NET Framework 的 ASP.NET Core

ASP.NET Core 可以面向 .NET Core 或 .NET Framework。面向 .NET Framework 的 ASP.NET Core 应用无法跨平台, 它们仅在 Windows 上运行。没有计划删除 ASP.NET Core 中对面向 .NET Framework 的支持。通常, ASP.NET Core 由 [.NET Standard](#) 库组成。使用 .NET Standard 2.0 编写的应用可在 .NET Standard 2.0 支持的任何位置运行。

面向 .NET Core 有以下几个优势, 并且这些优势会随着每次发布增加。与 .NET Framework 相比, .NET Core 的部分优势包括:

- 跨平台。在 macOS、Linux 和 Windows 上运行。
- 提高的性能
- 并行版本控制
- 新 API
- 打开源

我们正努力缩小 .NET Framework 与 .NET Core 的 API 差距。[Windows 兼容性包](#)使数千个仅 Windows API 可在 .NET Core 中使用。这些 API 在 .NET Core 1.x 中不可用。

后续步骤

有关更多信息, 请参见以下资源:

- [Razor 页面入门](#)
- [ASP.NET Core 教程](#)
- [ASP.NET Core 基础知识](#)
- [每周 ASP.NET Community Standup](#) 介绍了团队的工作进度和计划。它以新博客和第三方软件为重点。

ASP.NET Core 入门

2018/5/14 • 1 min to read • [Edit Online](#)

注意

这些说明仅适用于 ASP.NET Core 的最新版本。有关本文档的 1.1 版本，请参阅 [ASP.NET Core 1.1 入门](#)。

1. 安装 .NET Core SDK 2.0 or later。
2. 创建新的 .NET Core 项目。

在 macOS 和 Linux 上，打开终端窗口。在 Windows 上，打开命令提示符。输入以下命令：

```
dotnet new razor -o aspnetcoreapp
```

3. 运行应用。

使用以下命令运行应用：

```
cd aspnetcoreapp  
dotnet run
```

4. 浏览到 <http://localhost:5000>
5. 打开 Pages/About.cshtml 并将页面修改为显示消息“Hello, world! The time on the server is @DateTime.Now”：

```
@page  
@model AboutModel  
{  
    ViewData["Title"] = "About";  
}  
<h2>@ViewData["Title"]</h2>  
<h3>@Model.Message</h3>  
  
<p>Hello, world! The time on the server is @DateTime.Now</p>
```

6. 浏览到 <http://localhost:5000/About> 并验证更改。

后续步骤

有关入门教程，请参阅 [ASP.NET Core 教程](#)

有关 ASP.NET Core 概念和体系结构的简介，请参阅 [ASP.NET Core 简介](#) 和 [ASP.NET Core 基础知识](#)。

ASP.NET Core 应用可使用 .NET Core 或 .NET Framework 基类库和运行时。有关详细信息，请参阅 [在 .NET Core 和 .NET Framework 之间进行选择](#)。

ASP.NET Core 中的 Razor 页面介绍

2018/5/17 • 22 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Ryan Nowak](#)

Razor 页面是 ASP.NET Core MVC 的一个新特性, 它可以使基于页面的编码方式更简单高效。

若要查找使用模型视图控制器方法的教程, 请参阅 [ASP.NET Core MVC 入门](#)。

本文档介绍 Razor 页面。它并不是分步教程。如果认为某些部分过于复杂, 请参阅 [Razor 页面入门](#)。有关 ASP.NET Core 的概述, 请参阅 [ASP.NET Core 简介](#)。

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

创建 Razor 页面项目

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)

请参阅 [Razor 页面入门](#), 获取关于如何使用 Visual Studio 创建 Razor 页面项目的详细说明。

Razor 页面

Startup.cs 中已启用 Razor 页面:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

请考虑一个基本页面:

```

@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>

```

上述代码看上去类似于一个 Razor 视图文件。不同之处在于 `@page` 指令。`@page` 使文件转换为一个 MVC 操作，这意味着它将直接处理请求，而无需通过控制器处理。`@page` 必须是页面上的第一个 Razor 指令。`@page` 将影响其他 Razor 构造的行为。

将在以下两个文件中显示使用 `PageModel` 类的类似页面。Pages/Index2.cshtml 文件：

```

@page
@using RazorPagesIntro.Pages
@model IndexModel2

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>

```

Pages/Index2.cshtml.cs 页面模型：

```

using Microsoft.AspNetCore.Mvc.RazorPages;
using System;

namespace RazorPagesIntro.Pages
{
    public class IndexModel2 : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}

```

按照惯例，`PageModel` 类文件的名称与追加 `.cs` 的 Razor 页面文件名称相同。例如，前面的 Razor 页面的名称为 Pages/Index2.cshtml。包含 `PageModel` 类的文件的名称为 Pages/Index2.cshtml.cs。

页面的 URL 路径的关联由页面在文件系统中的位置决定。下表显示了 Razor 页面路径及匹配的 URL：

文件名和路径	匹配的 URL
/Pages/Index.cshtml	/ 或 /Index
/Pages/Contact.cshtml	/Contact
/Pages/Store/Contact.cshtml	/Store/Contact
/Pages/Store/Index.cshtml	/Store 或 /Store/Index

注意：

- 默认情况下，运行时在“Pages”文件夹中查找 Razor 页面文件。
- URL 未包含页面时，`Index` 为默认页面。

编写基本窗体

由于 Razor 页面的设计，在构建应用时可轻松实施用于 Web 浏览器的常用模式。[模型绑定、标记帮助程序](#) 和 [HTML 帮助程序](#) 均只可用于 Razor 页面类中定义的属性。请参考为 `Contact` 模型实现基本的“联系我们”窗体的页面：

在本文档中的示例中，`DbContext` 在 `Startup.cs` 文件中进行初始化。

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

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

数据模型：

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

数据库上下文：

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

Pages/Create.cshtml 视图文件:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

Pages/Create.cshtml.cs 页面模型:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public CreateModel(ApplicationDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}

```

按照惯例，`PageModel` 类命名为 `<PageName>Model` 并且它与页面位于同一个命名空间中。

使用 `PageModel` 类，可以将页面的逻辑与其展示分离开来。它定义了页面处理器，用于处理发送到页面的请求和用于呈现页面的数据。借助这种分离，可以通过[依赖关系注入](#)管理页面依赖关系，并对页面执行[单元测试](#)。

页面包含 `OnPostAsync` 处理程序方法，它在 `POST` 请求上运行（当用户发布窗体时）。可以为任何 HTTP 谓词添加处理程序方法。最常见的处理程序是：

- `OnGet`，用于初始化页面所需的状态。[OnGet 示例](#)。
- `OnPost`，用于处理窗体提交。

`Async` 命名后缀为可选，但是按照惯例通常会将它用于异步函数。前面示例中的 `OnPostAsync` 代码看上去与通常在控制器中编写的内容相似。前面的代码通常用于 Razor 页面。多数 MVC 基元（例如[模型绑定](#)、[验证](#)和操作结果）都是共享的。

之前的 `OnPostAsync` 方法：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

OnPostAsync 的基本流：

检查验证错误。

- 如果没有错误，则保存数据并重定向。
- 如果有错误，则再次显示页面并附带验证消息。客户端验证与传统的 ASP.NET Core MVC 应用程序相同。很多情况下，都会在客户端上检测到验证错误，并且从不将它们提交到服务器。

成功输入数据后，`OnPostAsync` 处理程序方法调用 `RedirectToPage` 帮助程序方法来返回 `RedirectToPageResult` 的实例。`RedirectToPage` 是新的操作结果，类似于 `RedirectToAction` 或 `RedirectToRoute`，但是已针对页面进行自定义。在前面的示例中，它将重定向到根索引页（/Index）。[页面 URL 生成](#)部分中详细介绍了 `RedirectToPage`。

提交的窗体存在（已传递到服务器的）验证错误时，`OnPostAsync` 处理程序方法调用 `Page` 帮助程序方法。`Page` 返回 `PageResult` 的实例。返回 `Page` 的过程与控制器中的操作返回 `View` 的过程相似。`PageResult` 是处理程序方法的默认返回类型。返回 `void` 的处理程序方法将显示页面。

`Customer` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。

```
public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}
```

默认情况下，Razor 页面只绑定带有非 GET 谓词的属性。绑定属性可以减少需要编写的代码量。绑定通过使用相同的属性显示窗体字段（`<input asp-for="Customer.Name" />`）来减少代码，并接受输入。

注意

出于安全原因，必须选择绑定 GET 请求数据以对模型属性进行分页。请在将用户输入映射到属性前对其进行验证。当处理依赖查询字符串或路由值的方案时，选择加入此行为非常有用。

若要将属性绑定在 GET 请求上，请将 `[BindProperty]` 特性的 `SupportsGet` 属性设置为 `true`：
`[BindProperty(SupportsGet = true)]`

主页（Index.cshtml）：

```
@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customers)
            {
                <tr>
                    <td>@contact.Id</td>
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page=".Edit" asp-route-id="@contact.Id">edit</a>
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>

    <a asp-page=".Create">Create</a>
</form>
```

Index.cshtml.cs 隱藏文件：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}

```

Index.cshtml 文件包含以下标记来创建每个联系人项的编辑链接：

```
<a asp-page=".Edit" asp-route-id="@contact.Id">edit</a>
```

[定位点标记帮助程序](#) 使用 `asp-route-{value}` 属性生成“编辑”页面的链接。此链接包含路由数据及联系人 ID。例如 <http://localhost:5000/Edit/1>。

Pages/Edit.cshtml 文件：

```
@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

 @{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

第一行包含 `@page "{id:int}"` 指令。路由约束 `"{id:int}"` 告诉页面接受包含 `int` 路由数据的页面请求。如果页面请求未包含可转换为 `int` 的路由数据，则运行时返回 HTTP 404(未找到)错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

Pages/Edit.cshtml.cs 文件：

```

using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}

```

Index.cshtml 文件还包含用于为每个客户联系人创建删除按钮的标记：

```

<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete</button>

```

删除按钮采用 HTML 呈现，其 `formaction` 包括参数：

- `asp-route-id` 属性指定的客户联系人 ID。

- `asp-page-handler` 属性指定的 `handler`。

下面是呈现的删除按钮的示例，其中客户联系人 ID 为 `1`：

```
<button type="submit" formaction="/?id=1&handler=delete">Delete</button>
```

选中按钮时，向服务器发送窗体 `POST` 请求。按照惯例，根据方案 `OnPost[handler]Async` 基于 `handler` 参数的值来选择处理程序方法的名称。

因为本示例中 `handler` 是 `delete`，因此 `OnPostDeleteAsync` 处理程序方法用于处理 `POST` 请求。如果 `asp-page-handler` 设置为不同值（如 `remove`），则选择名称为 `OnPostRemoveAsync` 的页面处理程序方法。

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

`OnPostDeleteAsync` 方法：

- 接受来自查询字符串的 `id`。
- 使用 `FindAsync` 查询客户联系人的数据库。
- 如果找到客户联系人，则从客户联系人列表将其删除。数据库将更新。
- 调用 `RedirectToPage`，重定向到根索引页（`/Index`）。

标记所需的页属性

`PageModel` 上的属性可通过 [Required](#) 特性进行修饰：

[!code-cs]

有关详细信息，请参阅[模型验证](#)。

使用 `OnGet` 处理程序管理 HEAD 请求

通常，针对 HEAD 请求创建和调用 HEAD 处理程序：

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("HandledBy", "Handled by OnHead!");
}
```

如果未定义 HEAD 处理程序（`OnHead`），Razor 页面会回退以调用 ASP.NET Core 2.1 或更高版本中的 GET 页处理程序（`OnGet`）。使用 ASP.NET Core 2.1 到 2.x 版本 `Startup.Configure` 中的 [SetCompatibilityVersion 方法](#)，选择加入此行为：

```
services.AddMvc()
    .SetCompatibilityVersion(Microsoft.AspNetCore.Mvc.CompatibilityVersion.Version_2_1);
```

`SetCompatibilityVersion` 有效地将 Razor 页面选项 `AllowMappingHeadRequestsToGetHandler` 设置为 `true`。

可以显式地选择使用特定行为，而不是通过 `SetCompatibilityVersion` 选择使用所有 2.1 行为。以下代码选择使用将 HEAD 映射到 GET 处理程序这一行为。

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.AllowMappingHeadRequestsToGetHandler = true;
});
```

XSRF/CSRF 和 Razor 页面

无需为[防伪验证](#)编写任何代码。Razor 页面自动将防伪标记生成过程和验证过程包含在内。

将布局、分区、模板和标记帮助程序用于 Razor 页面

页面可使用 Razor 视图引擎的所有功能。布局、分区、模板、标记帮助程序、`_ViewStart.cshtml` 和 `_ViewImports.cshtml` 的工作方式与它们在传统的 Razor 视图中的工作方式相同。

让我们使用其中的一些功能来整理此页面。

向 `Pages/_Layout.cshtml` 添加[布局页面](#)：

```
<!DOCTYPE html>
<html>
<head>
    <title>Razor Pages Sample</title>
</head>
<body>
    <a asp-page="/Index">Home</a>
    @RenderBody()
    <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>
```

布局：

- 控制每个页面的布局(页面选择退出布局时除外)。
- 导入 HTML 结构, 例如 JavaScript 和样式表。

请参阅[布局页面](#)了解详细信息。

在 `Pages/_ViewStart.cshtml` 中设置 `Layout` 属性：

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

布局位于“页面”文件夹中。页面按层次结构从当前页面的文件夹开始查找其他视图(布局、模板、分区)。可以从“页面”文件夹下的任意 Razor 页面使用“页面”文件夹中的布局。

建议不要将布局文件放在“视图/共享”文件夹中。视图/共享 是一种 MVC 视图模式。Razor 页面旨在依赖文件夹层次结构, 而非路径约定。

Razor 页面中的视图搜索包含“页面”文件夹。用于 MVC 控制器和传统 Razor 视图的布局、模板和分区可直接工作。

添加 `Pages/_ViewImports.cshtml` 文件：

```
@namespace RazorPagesContacts.Pages  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

本教程的后续部分中将介绍 `@namespace`。`@addTagHelper` 指令将[内置标记帮助程序](#)引入“页面”文件夹中的所有页面。

页面上显式使用 `@namespace` 指令后：

```
@page  
@namespace RazorPagesIntro.Pages.Customers  
  
@model NameSpaceModel  
  
<h2>Name space</h2>  
<p>  
    @Model.Message  
</p>
```

此指令将为页面设置命名空间。`@model` 指令无需包含命名空间。

_ViewImports.cshtml 中包含 `@namespace` 指令后，指定的命名空间将为在导入 `@namespace` 指令的页面中生成的命名空间提供前缀。生成的命名空间的剩余部分(后缀部分)是包含 _ViewImports.cshtml 的文件夹与包含页面的文件夹之间以点分隔的相对路径。

例如，代码隐藏文件 Pages/Customers/Edit.cshtml.cs 显式设置命名空间：

```
namespace RazorPagesContacts.Pages  
{  
    public class EditModel : PageModel  
    {  
        private readonly AppDbContext _db;  
  
        public EditModel(AppDbContext db)  
        {  
            _db = db;  
        }  
  
        // Code removed for brevity.  
    }  
}
```

Pages/_ViewImports.cshtml 文件设置以下命名空间：

```
@namespace RazorPagesContacts.Pages  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

为 Pages/Customers/Edit.cshtml Razor 页面生成的命名空间与代码隐藏文件相同。已对 `@namespace` 指令进行设计，因此添加到项目的 C# 类和页面生成的代码可直接工作，而无需添加代码隐藏文件的 `@using` 指令。

`@namespace` 也可用于传统的 Razor 视图。

原始的 Pages/Create.cshtml 视图文件：

```

@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>

```

更新后的 Pages/Create.cshtml 视图文件：

```

@page
@model CreateModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>

```

Razor 页面初学者项目包含 Pages/_ValidationScriptsPartial.cshtml，它与客户端验证联合。

页面的 URL 生成

之前显示的 `Create` 页面使用 `RedirectToPage`：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

应用具有以下文件/文件夹结构：

- /Pages
 - Index.cshtml
 - /Customers
 - Create.cshtml

- Edit.cshtml
- Index.cshtml

成功后，Pages/Customers/Create.cshtml 和 Pages/Customers/Edit.cshtml 页面将重定向到 Pages/Index.cshtml。字符串 /Index 是用于访问上一页的 URI 的组成部分。可以使用字符串 /Index 生成 Pages/Index.cshtml 页面的 URI。例如：

- Url.Page("/Index", ...)
- <a asp-page="/Index">My Index Page
- RedirectToPage("/Index")

页面名称是从根"/Pages"文件夹到页面的路径(包含前导 /，例如 /Index)。与硬编码 URL 相比，前面的 URL 生成示例提供了改进的选项和功能。URL 生成使用[路由](#)，并且可以根据目标路径定义路由的方式生成参数并对参数编码。

页面的 URL 生成支持相对名称。下表显示了 Pages/Customers/Create.cshtml 中不同的 RedirectToPage 参数选择的索引页：

REDIRECTTOPAGE(X)	页
RedirectToPage("/Index")	Pages/Index
RedirectToPage("./Index");	Pages/Customers/Index
RedirectToPage("../Index")	Pages/Index
RedirectToPage("Index")	Pages/Customers/Index

RedirectToPage("Index")、 RedirectToPage("./Index") 和 RedirectToPage("../Index") 是相对名称。结合 RedirectToPage 参数与当前页的路径来计算目标页面的名称。

构建结构复杂的站点时，相对名称链接很有用。如果使用相对名称链接文件夹中的页面，则可以重命名该文件夹。所有链接仍然有效(因为这些链接未包含此文件夹名称)。

ViewData 特性

可以通过 [ViewDataAttribute](#) 将数据传递到页面。控制器或 Razor 页面模型上使用 [ViewData] 修饰的属性将其值存储在 [ViewDataDictionary](#) 中并从此处进行加载。

在下面的示例中，AboutModel 包含使用 [ViewData] 修饰的 Title 属性。Title 属性设置为“关于”页面的标题：

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

在“关于”页面中，以模型属性的形式访问 Title 属性：

```
<h1>@Model.Title</h1>
```

在布局中，从 ViewData 字典读取标题：

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>@ViewData["Title"] - WebApplication</title>
...
```

TempData

ASP.NET 在[控制器](#)上公开了 `TempData` 属性。此属性存储未读取的数据。`Keep` 和 `Peek` 方法可用于检查数据，而不执行删除。多个请求需要数据时， `TempData` 有助于进行重定向。

`[TempData]` 是 ASP.NET Core 2.0 中的新属性，在控制器和页面上受支持。

下面的代码使用 `TempData` 设置 `Message` 的值：

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

Pages/Customers/Index.cshtml 文件中的以下标记使用 `TempData` 显示 `Message` 的值。

```
<h3>Msg: @Model.Message</h3>
```

Pages/Customers/Index.cshtml.cs 页面模型将 `[TempData]` 属性应用到 `Message` 属性。

```
[TempData]
public string Message { get; set; }
```

请参阅 [TempData](#) 了解详细信息。

针对一个页面的多个处理程序

以下页面使用 `asp-page-handler` 标记帮助程序为两个页面处理程序生成标记:

```
@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

前面示例中的窗体包含两个提交按钮，每个提交按钮均使用 `FormActionTagHelper` 提交到不同的 URL。

`asp-page-handler` 是 `asp-page` 的配套属性。`asp-page-handler` 生成提交到页面定义的各个处理程序方法的 URL。未指定 `asp-page`，因为示例已链接到当前页面。

页面模型:

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpper();
            return await OnPostJoinListAsync();
        }
    }
}

```

前面的代码使用已命名处理程序方法。已命名处理程序方法通过采用名称中 `on<HTTP Verb>` 之后及 `Async` 之前的文本(如果有)创建。在前面的示例中，页面方法是 `OnPostJoinListAsync` 和 `OnPostJoinListUCAsync`。删除 `OnPost` 和 `Async` 后，处理程序名称为 `JoinList` 和 `JoinListUC`。

```

<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />

```

使用前面的代码时，提交到 `OnPostJoinListAsync` 的 URL 路径为

`http://localhost:5000/Customers/CreateFATH?handler=JoinList`。提交到 `OnPostJoinListUCAsync` 的 URL 路径为
`http://localhost:5000/Customers/CreateFATH?handler=JoinListUC`。

自定义路由

如果你不喜欢 URL 中的查询字符串 `?handler=JoinList`，可以更改路由，将处理程序名称放在 URL 的路径部分。可以通过在 `@page` 指令后面添加使用双引号括起来的路由模板来自定义路由。

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

前面的路由将处理程序放在了 URL 路径中，而不是查询字符串中。`handler` 前面的 `?` 表示路由参数为可选。

可以使用 `@page` 将其他段和参数添加到页面的路由中。其中的任何内容均会被追加到页面的默认路由中。不支持使用绝对路径或虚拟路径更改页面的路由（如 `"~/Some/Other/Path"`）。

配置和设置

若要配置高级选项，请在 MVC 生成器上使用 `AddRazorPagesOptions` 扩展方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
    {
        options.RootDirectory = "/MyPages";
        options.Conventions.AuthorizeFolder("/MyPages/Admin");
    });
}
```

目前，可以使用 `RazorPagesOptions` 设置页面的根目录，或者为页面添加应用程序模型约定。通过这种方式，我们在将来会实现更多扩展功能。

若要预编译视图，请参阅 [Razor 视图编译](#)。

[下载或查看示例代码](#).

请参阅 [Razor 页面入门](#)，这篇文章以本文为基础编写。

指定 Razor 页面位于内容根目录中

默认情况下，Razor 页面位于 `/Pages` 目录的根位置。向 `AddMvc` 添加 `WithRazorPagesAtContentRoot`，以指定 Razor 页面位于应用的内容根目录 (`ContentRootPath`) 中：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    ...
})
    .WithRazorPagesAtContentRoot();
```

指定 Razor 页面位于自定义根目录中

向 `AddMvc` 添加 `WithRazorPagesRoot`，以指定 Razor 页面位于应用中自定义根目录位置(提供相对路径)：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    ...
})
    .WithRazorPagesRoot("/path/to/razor/pages");
```

请参阅

- [ASP.NET Core 简介](#)
- [Razor 语法](#)
- [Razor 页面入门](#)
- [Razor 页面授权约定](#)
- [Razor 页面自定义路由和页面模型提供程序](#)
- [Razor 页面单位与集成测试](#)

使用 ASP.NET Core 和 Visual Studio for Windows 创建 Web API

2018/5/17 • 17 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Mike Wasson](#)

本教程构建了用于管理“待办事项”列表的 Web API。未创建用户界面 (UI)。

本教程提供 3 个版本：

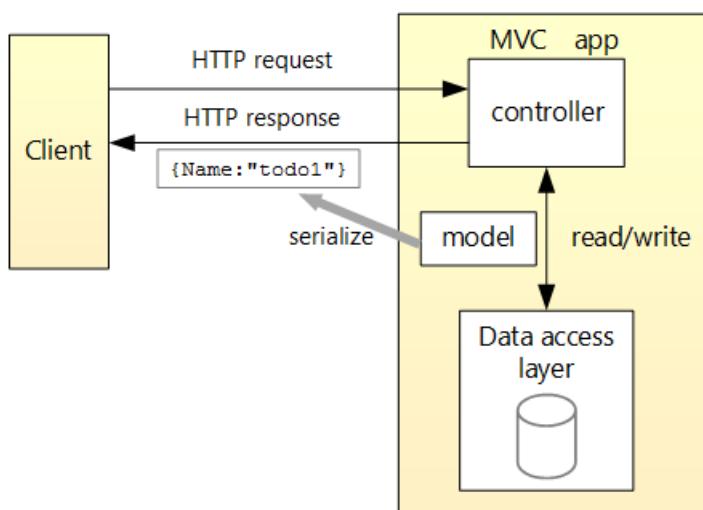
- Windows: 使用 Visual Studio for Windows 创建 Web API (本教程)
- macOS: [使用 Visual Studio for Mac 创建 Web API](#)
- macOS、Linux、Windows: [使用 Visual Studio Code 创建 Web API](#)

概述

本教程将创建以下 API：

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中，唯一的模型是待办事项。模型表示为 C# 类，也称为 Plain Old C# Object (POCO)。

- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

系统必备

Visual Studio for Windows

- **ASP.NET and web development workload**
- **.NET Core cross-platform development workload**
- X.509 security certificate

创建项目

在 Visual Studio 中执行以下步骤：

- 从“文件”菜单中选择“新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”模板。将项目命名为 TodoApi，然后单击“确定”。
- 在“新建 ASP.NET Core Web 应用程序 - TodoApi”对话框中，选择 ASP.NET Core 版本。选择“API”模板，然后单击“确定”。请不要选择“启用 Docker 支持”。

启动应用

在 Visual Studio 中，按 CTRL+F5 启动应用。Visual Studio 启动浏览器并导航到

`http://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。Chrome、Microsoft Edge 和 Firefox 将显示以下输出：

```
["value1","value2"]
```

添加模型类

模型是表示应用中的数据的对象。在此示例中，唯一的模型是待办事项。

在解决方案资源管理器中，右键单击项目。选择“添加” > “新建文件夹”。将文件夹命名为“Models”。

注意

模型类可以出现在项目的任意位置。Models 文件夹按约定用于模型类。

在解决方案资源管理器中右键单击“模型”文件夹，然后选择“添加” > “类”。将类命名为 TodoItem，然后单击“添加”。

使用以下代码更新 `TodoItem` 类：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时，数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调实体框架功能的主类。此类由 `Microsoft.EntityFrameworkCore.DbContext` 类派

生而来。

在解决方案资源管理器中右键单击“模型”文件夹，然后选择“添加”>“类”。将类命名为 TodoContext，然后单击“添加”。

将该类替换为以下代码：

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {

        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 Startup.cs 文件的内容替换为以下代码：

[!code-csharp]

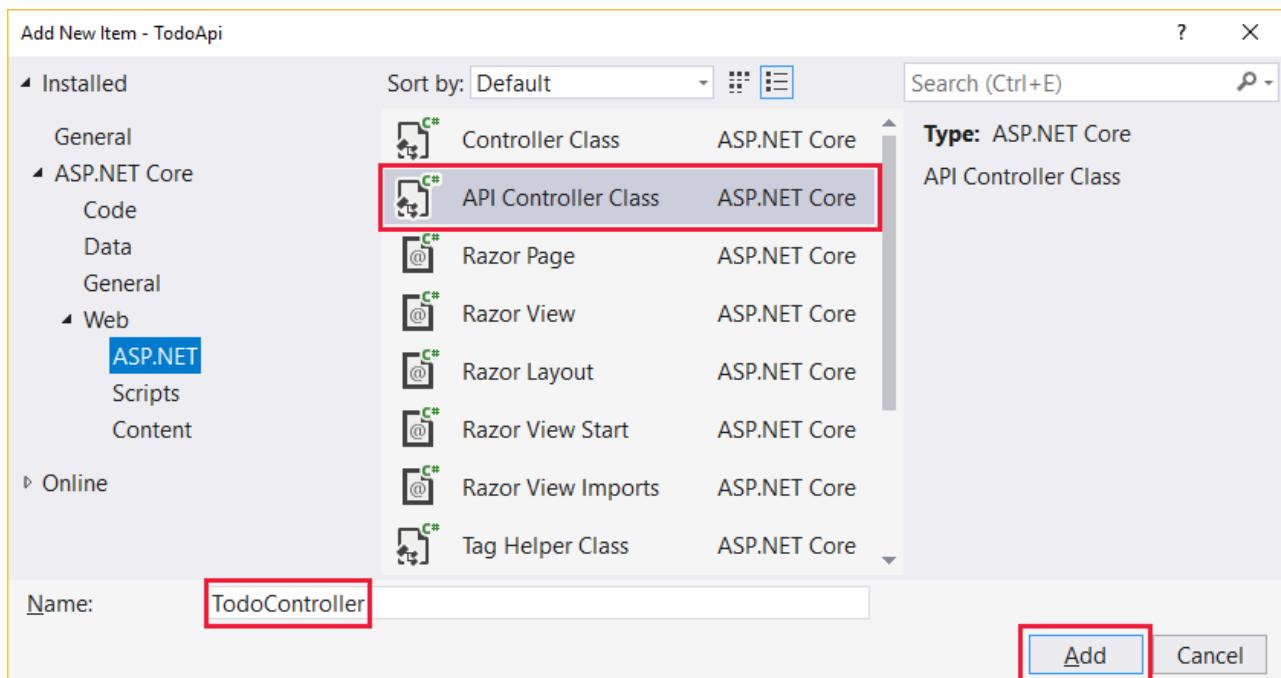
[!code-csharp]

前面的代码：

- 删除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在解决方案资源管理器中，右键单击“控制器”文件夹。选择 **添加 > 新建项**。在“添加新项”对话框中，选择“API 控制器类”模板。将类命名为 TodoController，然后单击“添加”。



将该类替换为以下代码：

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以启用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项(如果不存在)添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

```
[!code-csharp]
```

```
[!code-csharp]
```

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
 {  
   "id": 1,  
   "name": "Item1",  
   "isComplete": false  
 }  
]
```

稍后将在本教程中演示如何使用 `Postman` 或 `curl` 查看 HTTP 响应。

路由和 URL 路径

[HttpGet] 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板（如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

```
[!code-csharp]
```

```
[!code-csharp]
```

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。
- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 `JSON`，并将 `JSON` 写入响应消息的正文。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 `IActionResult` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 `JSON` 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 `ActionResult<T>` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 `JSON` 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 Visual Studio 中，按 `CTRL+F5` 启动应用。Visual Studio 启动浏览器并导航到 `http://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。导航到位子 `http://localhost:<port>/api/todo` 的 `Todo` 控制器。

实现其他的 CRUD 操作

在以下部分中，将 `Create`、`Update` 和 `Delete` 方法添加到控制器。

创建

添加以下 `Create` 方法：

```
[!code-csharp]
```

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

```
[!code-csharp]
```

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。MVC 从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法：

- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 [10.2.2 201 已创建](#)。
- 使用名为 `route` 的“`GetTodo`”来创建 URL。已在 `GetById` 中定义名为 `route` 的“`GetTodo`”：

```
[!code-csharp]
```

```
[!code-csharp]
```

使用 Postman 发送创建请求

- 启动该应用程序。
- 打开 Postman。

The screenshot shows the Postman application interface. The top navigation bar includes File, Edit, View, Help, New, Import, Runner, My Workspace, and IN SYNC status. The main workspace shows a POST request to `https://localhost:44375/api/todo`. The Body tab is selected, showing the raw JSON content:

```
1 [{}  
2   "name": "walk dog",  
3   "isComplete": true  
4 ]
```

The Headers section shows three entries: `Content-Type: application/json`, `Accept: */*`, and `Host: localhost:44375`. The Body content type is set to `raw` and `JSON (application/json)`. The response tab shows a successful `201 Created` status with a response time of `148 ms`. The response body is identical to the request body.

- 更新 `localhost` URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 `JSON (application/json)`
- 输入包含待办事项的请求正文，类似以下 JSON：

```
{  
    "name": "walk dog",  
    "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应，则禁用“SSL 证书验证”选项。在“文件”>“设置”下可以找到该选项。在禁用该设置后，再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡，然后复制位置标头值：

The screenshot shows the Postman interface. In the main area, a POST request is being made to `https://localhost:44375/api/todo`. The Body tab is selected, showing a JSON payload:

```
1 [{}  
2     "name": "walk dog",  
3     "isComplete": true  
4 ]
```

Below the request, the response pane is visible. The **Headers** tab is highlighted with a red box. It contains the following header information:

- `Content-Type` → `application/json; charset=utf-8`
- `Date` → `Fri, 27 Apr 2018 18:32:32 GMT`
- `Location` → `https://localhost/api/Todo/6`

The status bar at the bottom right indicates `Status: 201 Created Time: 148 ms`.

位置标头 URI 可用于访问新项。

更新

添加以下 `Update` 方法：

```
![code-csharp]
```

```
![code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 [204\(无内容\)](#)。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

使用 Postman 将待办事项的名称更新为“带猫出去散步”：

The screenshot shows the Postman interface with a red box highlighting the 'PUT' method in the top left. The URL is set to <https://localhost:44375/api/todo/1>. The 'Body' tab is selected, indicated by a blue dot, and has a red box around it. Below it, the 'raw' radio button is selected, also highlighted with a red box. The JSON payload is defined as:

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```

The response section shows the status as **204 No Content** and time as **70 ms**. The 'Pretty' tab is selected in the preview dropdown.

删除

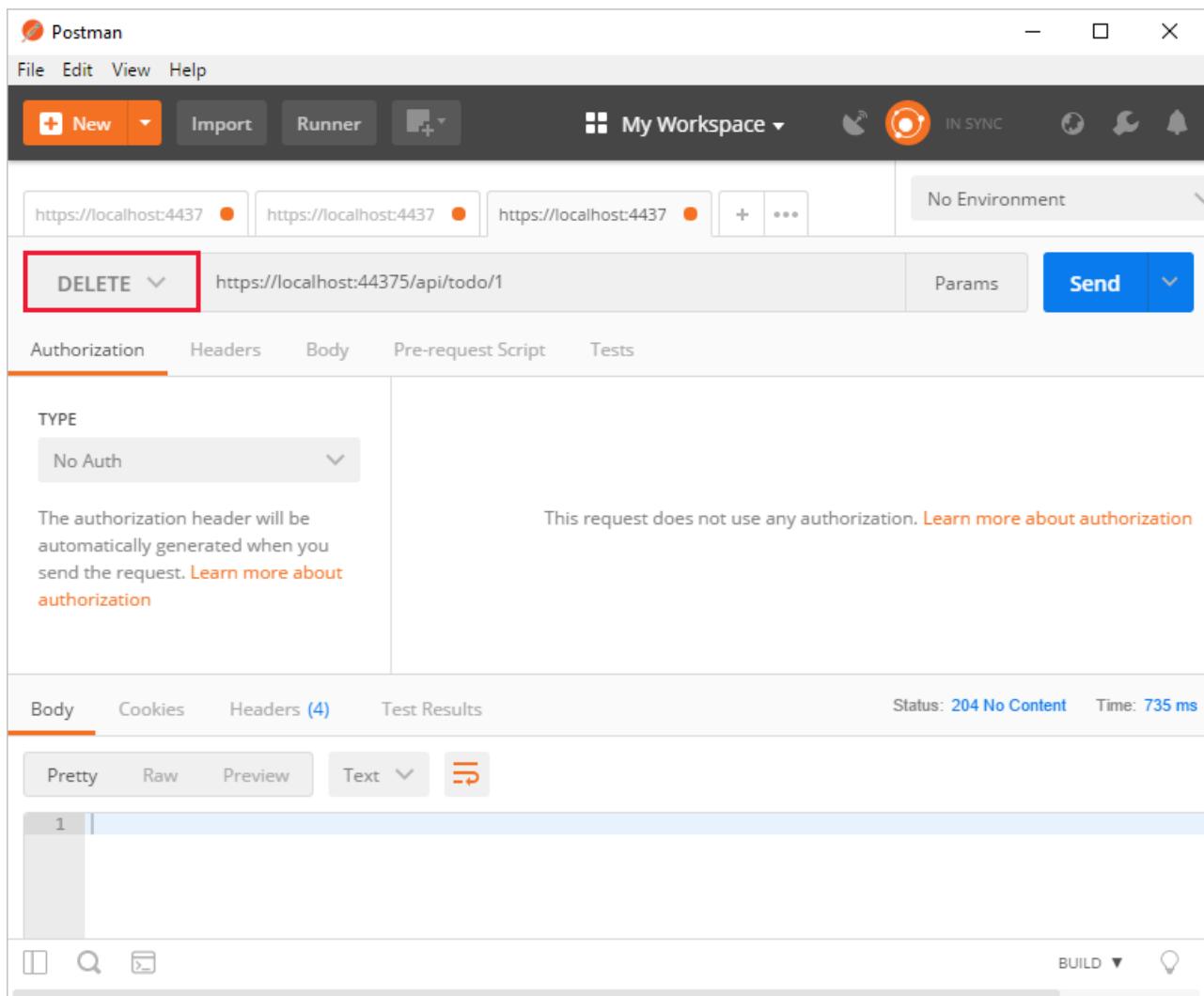
添加以下 **Delete** 方法:

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

Delete 响应是 **204(无内容)**。

使用 Postman 删除待办事项:



使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 `Startup.Configure` 中调用 `UseStaticFiles` 和 `UseDefaultFiles` 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 `index.html` 的 HTML 文件添加至项目的 `wwwroot` 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
```

```

        display: none;
    }

    table {
        font-family: Arial, sans-serif;
        border: 1px solid;
        border-collapse: collapse;
    }

    th {
        background-color: #0066CC;
        color: white;
    }

    td {
        border: 1px solid;
        padding: 5px;
    }

```

</style>

</head>

<body>

<h1>To-do CRUD</h1>

<h3>Add</h3>

<form action="javascript:void(0);" method="POST" onsubmit="addItem()">

<input type="text" id="add-name" placeholder="New to-do">

<input type="submit" value="Add">

</form>

<div id="spoiler">

<h3>Edit</h3>

<form class="my-form">

<input type="hidden" id="edit-id">

<input type="checkbox" id="edit-isComplete">

<input type="text" id="edit-name">

<input type="submit" value="Edit">

×

</form>

</div>

<p id="counter"></p>

<table>

<tr>

<th>Is Complete</th>

<th>Name</th>

<th></th>

<th></th>

</tr>

<tbody id="todos"></tbody>

</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
 integrity="sha256-FgpCb/KJQ1LNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
 crossorigin="anonymous"></script>

<script src="site.js"></script>

</body>

</html>

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
    const el = $('#counter');
    let name = 'to-do';
    if (data) {
        if (data > 1) {
            name = 'to-dos';
        }
        el.text(name + ' ' + data);
    } else {
        el.text('to-do');
    }
}

```

```

        name = todos ;
    }
    el.text(data + ' ' + name);
} else {
    el.html('No ' + name);
}
}

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';
                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };
    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            ...
        }
    });
}

```

```

        $('#edit-name').val(item.name);
        $('#edit-id').val(item.id);
        $('#edit-isComplete').val(item.isComplete);
    }
});

$('#spoiler').css({ 'display': 'block' });

}

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });

    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';

                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
        }
    });
}

```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```
$.ajax({
    url: uri + '/' + $('#edit-id').val(),
    type: 'PUT',
    accepts: 'application/json',
    contentType: 'application/json',
    data: JSON.stringify(item),
    success: function (result) {
        getData();
    }
});
```

删除待办事项

若要删除待办事项, 请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
    url: uri + '/' + id,
    type: 'DELETE',
    success: function (result) {
        getData();
    }
});
```

后续步骤

- 有关使用永久数据库的详细信息, 请参阅:
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)
- 有关部署 API 的信息(包括部署到 Azure 应用服务), 请参阅[托管和部署](#)。
- [查看或下载示例代码](#)。请参阅[如何下载](#)。

ASP.NET Core 教程

2018/4/28 • 2 min to read • [Edit Online](#)

下面的分步指南已可用于开发 ASP.NET Core 应用程序：

生成 Web 应用

[Razor 页面](#) 是使用 ASP.NET Core 2.0 创建新 Web UI 应用时建议使用的方法。

- [ASP.NET Core 中的 Razor 页面介绍](#)
- 使用 ASP.NET Core 创建 Razor 页面 Web 应用
 - 在 Windows 上创建 Razor 页面
 - macOS 上的 Razor 页面
 - 使用 VS Code 创建 Razor 页面
- 创建 ASP.NET Core MVC Web 应用
 - 使用 Visual Studio for Windows 创建 Web 应用
 - 使用 Visual Studio for Mac 创建 Web 应用
 - 在 macOS 或 Linux 上使用 Visual Studio Code 创建 Web 应用
- 通过 Visual Studio 开始使用 ASP.NET Core 和 Entity Framework Core
- 创建标记帮助程序
- 创建简单的视图组件
- 使用 `dotnet watch` 开发 ASP.NET Core 应用

生成 Web API

- 使用 ASP.NET Core 创建 Web API
 - 使用 Visual Studio for Windows 创建 Web API
 - 使用 Visual Studio for Mac 创建 Web API
 - 使用 Visual Studio Code 创建 Web API
- 使用 Swagger 的 ASP.NET Core Web API 帮助页
 - NSwag 入门
 - Swashbuckle 入门
- 为本机移动应用创建后端 Web 服务

数据访问和存储

- 通过 Visual Studio 开始使用 Razor 页面和 EF Core
- 通过 Visual Studio 开始使用 ASP.NET Core MVC 和 EF Core
- ASP.NET Core MVC 和 EF Core - 新数据库
- ASP.NET Core MVC 和 EF Core - 现有数据库

身份验证和授权

- 启用使用 Facebook、Google 和其他外部提供程序的身份验证
- 帐户确认和密码恢复
- 使用 SMS 设置双因素身份验证

客户端开发

- 使用 Gulp
- 使用 Grunt
- 使用 Bower 管理客户端包
- 使用 Bootstrap 构建响应式站点

测试

- 使用 dotnet 测试在 .NET Core 中进行单元测试

托管和部署

- 使用 Visual Studio 将 ASP.NET Core Web 应用部署到 Azure
- 使用命令行将 ASP.NET Core Web 应用部署到 Azure
- 通过持续部署发布到 Azure Web 应用
- 将 ASP.NET 容器部署到远程 Docker 主机
- ASP.NET Core 和 Azure Service Fabric

如何下载示例

1. 下载 ASP.NET 存储库 zip 文件。
2. 解压缩 Docs-master.zip 文件。
3. 使用示例链接中的 URL 帮助你导航到示例目录。

使用 ASP.NET Core 创建 Razor 页面 Web 应用

2018/4/27 • 1 min to read • [Edit Online](#)

此系列介绍了使用 Visual Studio 在 ASP.NET Core 中生成 Razor 页面 Web 应用的基础知识。此系列的其他版本包括 [macOS 版本](#) 和 [Visual Studio Code 版本](#)。

1. [Razor 页面入门](#)
2. [向 Razor 页面应用添加模型](#)
3. [已搭建基架的 Razor 页面](#)
4. [使用 SQL Server LocalDB](#)
5. [更新页面](#)
6. [添加搜索](#)
7. [添加新字段](#)
8. [添加验证](#)
9. [上载文件](#)

在 ASP.NET Core 中开始使用 Razor Pages

2018/5/17 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍构建 ASP.NET Core Razor Pages Web 应用的基础知识。Razor Pages 是在 ASP.NET Core 中为 Web 应用生成 UI 时建议使用的方法。

本教程提供 3 个版本:

- Windows: [本教程](#)
- MacOS: [借助 Visual Studio for Mac 开始使用 Razor Pages](#)
- macOS、Linux 和 Windows: [在 Visual Studio Code 中开始使用 ASP.NET Core Razor 页面](#)

[查看或下载示例代码\(如何下载\)](#)

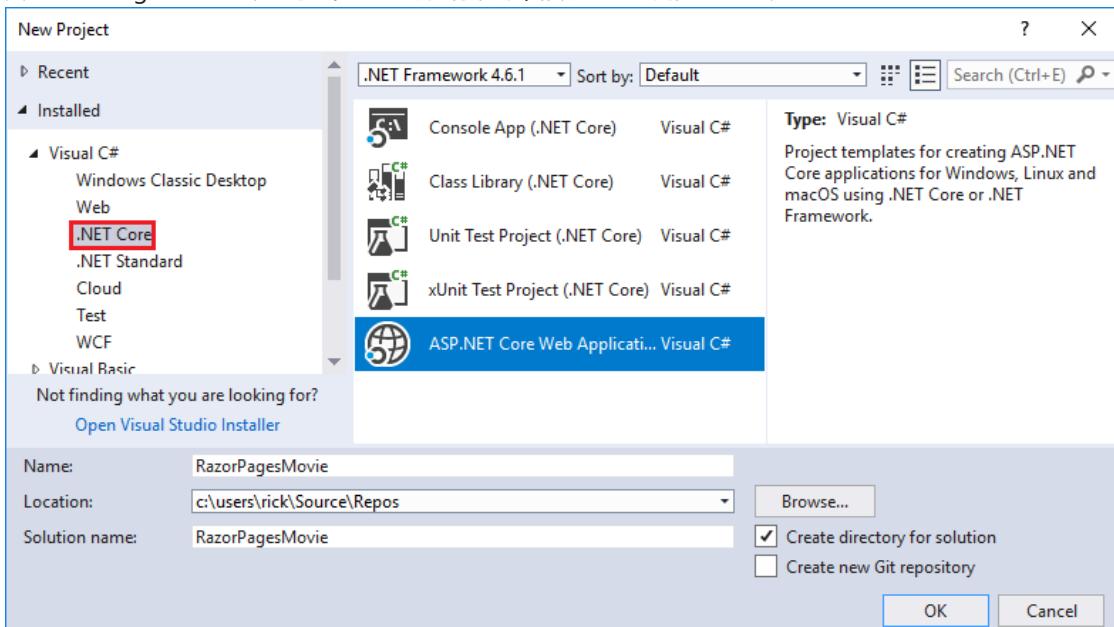
系统必备

[Visual Studio for Windows](#)

- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate

创建 Razor Web 应用

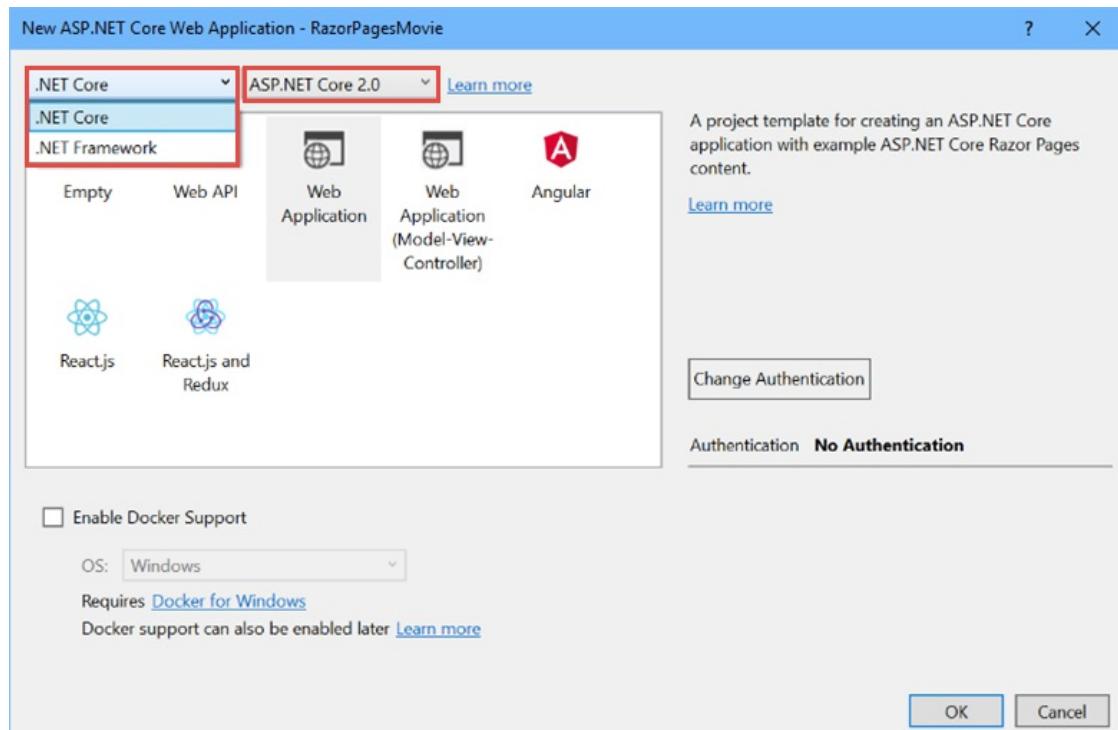
- 从 Visual Studio“文件”菜单中选择“新建”>“项目”。
- 创建新的 ASP.NET Core Web 应用程序。将项目命名为“RazorPagesMovie”。将项目命名为“RazorPagesMovie”，以便命名空间在你复制/粘贴代码时相互匹配。



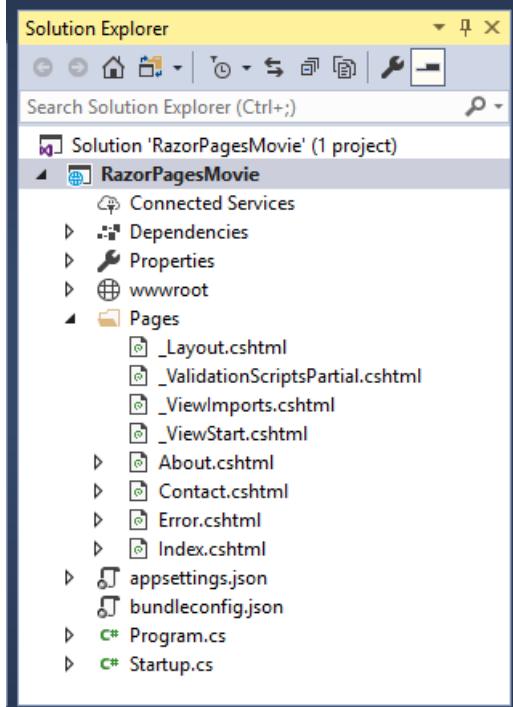
- 在下拉列表中选择“ASP.NET Core 2.0”，然后选择“Web 应用程序”。

注意

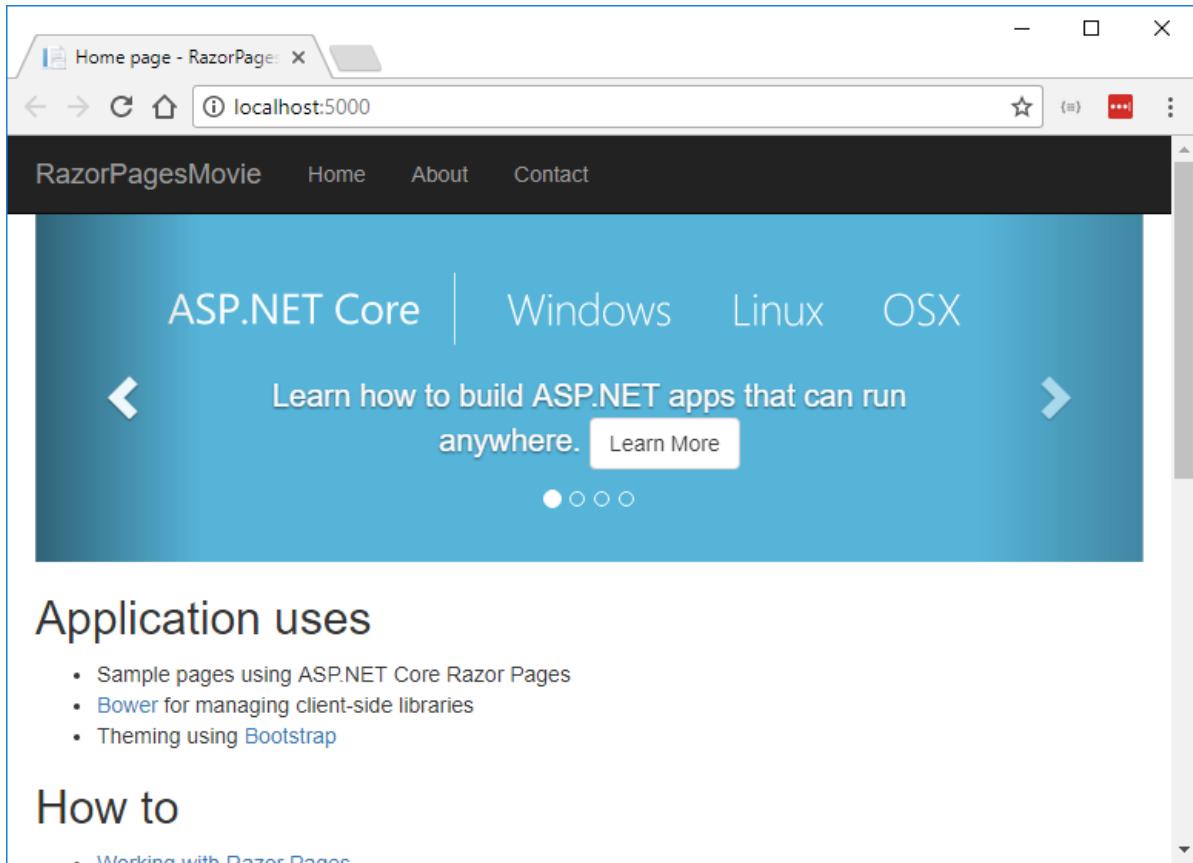
要在 .NET Framework 中使用 ASP.NET Core，必须先从对话框最左侧的下拉列表选择“.NET Framework”，然后才能选择所需的 ASP.NET Core 版本。



Visual Studio 模板创建初学者项目：

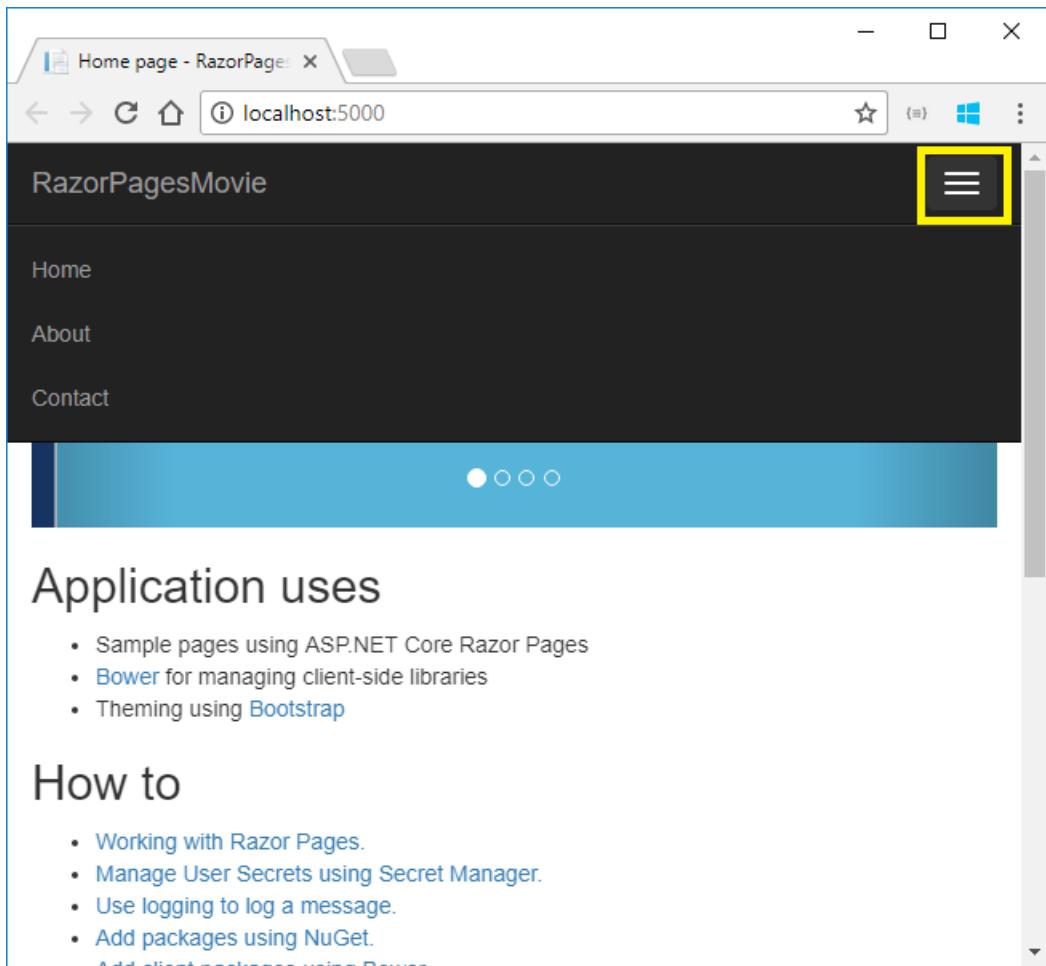


按 F5 在调试模式下运行应用, 或按 Ctrl-F5 在运行(不附加调试器)



- Visual Studio 启动 IIS Express 并运行你的应用。地址栏显示 `localhost:port#`，而不显示 `example.com`。这是因为 `localhost` 是本地计算机的标准主机名。Localhost 仅为来自本地计算机的 Web 请求提供服务。Visual Studio 创建 Web 项目时，为 Web 服务器使用随机端口。在上图中，端口号为 5000。运行应用时，将看到不同的端口号。
- 使用“Ctrl+F5”启动应用（非调试模式）后，可执行代码更改、保存文件、刷新浏览器和查看代码更改等操作。许多开发人员更喜欢使用非调试模式快速启动应用并查看更改。

默认模板创建“RazorPagesMovie”、“主页”、“关于”和“联系人”链接和页面。可能需要单击导航图标才能显示这些链接，具体取决于浏览器窗口的大小。



测试链接。“RazorPagesMovie”和“主页”链接转到“索引”页。“关于”和“联系人”链接分别转到 [About](#) 和 [Contact](#) 页面。

项目文件和文件夹

下表列出了项目中的文件和文件夹。对于本教程而言，Startup.cs 是最有必要了解的文件。无需查看下面提供的每一个链接。需要详细了解项目中的某个文件或文件夹时，可参考此处提供的链接。

文件或文件夹	目标
wwwroot	包含静态文件。请参阅 使用静态文件 。
页数	Razor Pages 的文件夹。
appsettings.json	配置
Program.cs	托管 ASP.NET Core 应用。
Startup.cs	配置服务和请求管道。请参阅 启动 。

“页面”文件夹

_Layout.cshtml 文件包含常见的 HTML 元素(脚本和样式表)，并设置应用程序的布局。例如，单击“RazorPagesMovie”、“主页”、“关于”或“联系人”时，将看到相同的元素。常见的元素包括顶部的导航菜单和窗口底部的标题。请参阅[布局](#)了解详细信息。

_ViewStart.cshtml 将 Razor 页面 [Layout](#) 属性设置为使用 _Layout.cshtml 文件。请参阅[布局](#)了解详细信息。

_ViewImports.cshtml 文件包含要导入每个 Razor 页面的 Razor 指令。请参阅[导入共享指令](#)了解详细信息。

_ValidationScriptsPartial.cshtml 文件提供对 [jQuery](#) 验证脚本的引用。在本教程的后续部分中添加 [Create](#) 和 [Edit](#) 页面时，将使用 _ValidationScriptsPartial.cshtml 文件。

[About](#)、[Contact](#) 和 [Index](#) 页面是基本页面，可用于启动应用。[Error](#) 页面用于显示错误信息。

[下一篇：添加模型](#)

在 ASP.NET Core 中向 Razor 页面应用添加模型

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中将添加用于管理数据库中的电影的类。可以结合使用这些类和 Entity Framework Core (EF Core) 来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化必须要编写的数据访问代码。

要创建的模型类称为 POCO 类(源自“简单传统 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们定义数据库中存储的数据属性。

在本教程中，首先要编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍):[从现有数据库生成模型类](#)。

[查看或下载示例。](#)

添加数据模型

在解决方案资源管理器中，右键单击“RazorPagesMovie”项目 > “添加” > “新建文件夹”。将文件夹命名为“Models”。

右键单击“Models”文件夹。选择“添加” > “类”。将类命名为“Movie”，并添加以下属性：

向 `Movie` 类添加以下属性：

```
using System;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段作为主键。

添加数据库上下文类

将以下 `MovieContext.cs` 类添加到“模型”文件夹:[!code-csharp]

前面的代码为实体集创建 `DbSet` 属性。在实体框架术语中，实体集通常与数据库表相对应，实体与表中的行相对应。

添加数据库连接字符串

将连接字符串添加到 `appsettings.json` 文件。

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "ConnectionStrings": {  
        "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movie-  
1;Trusted_Connection=True;MultipleActiveResultSets=true"  
    }  
}
```

注册数据库上下文

使用 Startup.cs 文件中的[依存关系注入](#)容器注册数据库上下文。

```
public void ConfigureServices(IServiceCollection services)  
{  
    // requires  
    // using RazorPagesMovie.Models;  
    // using Microsoft.EntityFrameworkCore;  
  
    services.AddDbContext<MovieContext>(options =>  
        options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));  
    services.AddMvc();  
}
```

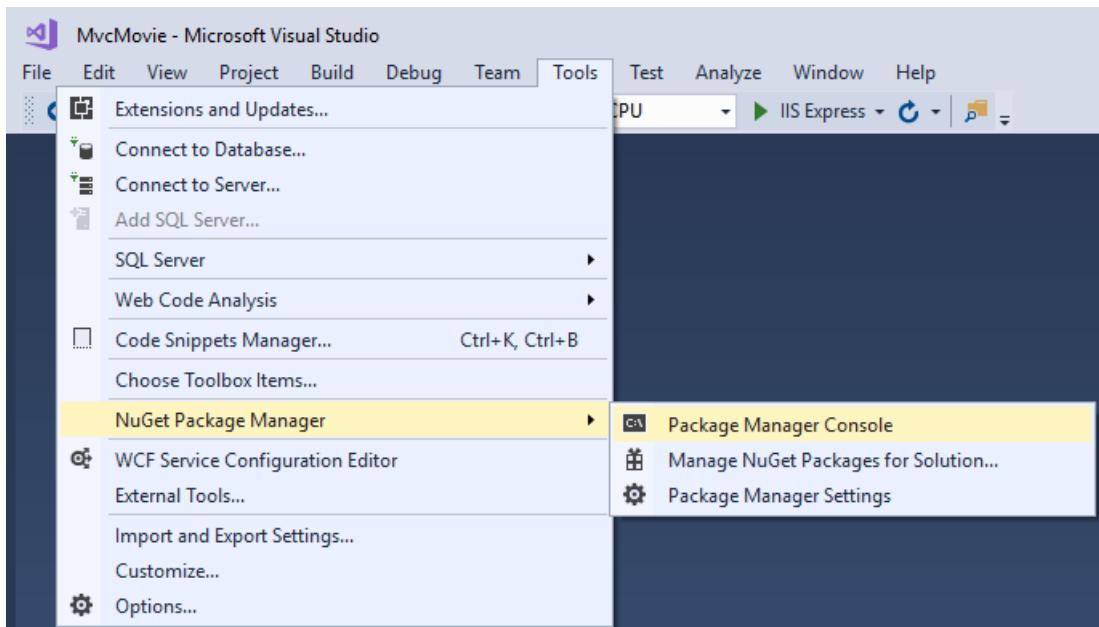
生成项目以验证有没有任何错误存在。

添加基架工具并执行初始迁移

在本部分中，使用程序包管理器控制台 (PMC) 执行以下操作：

- 添加 Visual Studio Web 代码生成包。必须添加此包才能运行基架引擎。
- 添加初始迁移。
- 使用初始迁移来更新数据库。

从“工具”菜单中，选择“NuGet 包管理器” > “包管理器控制台”。



在 PMC 中，输入以下命令：

```
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design  
Add-Migration Initial  
Update-Database
```

或者, 可使用以下 .NET Core CLI 命令:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design  
dotnet ef migrations add Initial  
dotnet ef database update
```

`Install-Package` 命令安装运行基架引擎所需的工具。

`Add-Migration` 命令生成用于创建初始数据库架构的代码。此架构以(Models/MovieContext.cs 文件中的 `DbContext` 中指定的模型为基础。`Initial` 参数用于为迁移命名。可以使用任意名称, 但是按照惯例应选择描述迁移的名称。有关详细信息, 请参阅[迁移简介](#)。

`Update-Database` 命令在用于创建数据库的 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 `Up` 方法。

搭建“电影”模型的基架

- 从命令行(在包含 Program.cs、Startup.cs 和 .csproj 文件的项目目录中)中运行如下命令:

```
dotnet aspnet-codegenerator razorpage -m Movie -dc MovieContext -udl -outDir Pages\Movies --referenceScriptLibraries
```

如果收到错误:

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

当处于错误的目录时, 会发生上述错误。打开命令行界面, 进入项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录), 然后运行上述命令。

如果收到错误:

```
The process cannot access the file  
'RazorPagesMovie/bin/Debug/netcoreapp2.0/RazorPagesMovie.dll'  
because it is being used by another process.
```

退出 Visual Studio, 然后重新运行命令。

下表详细说明了 ASP.NET Core 代码生成器的参数:

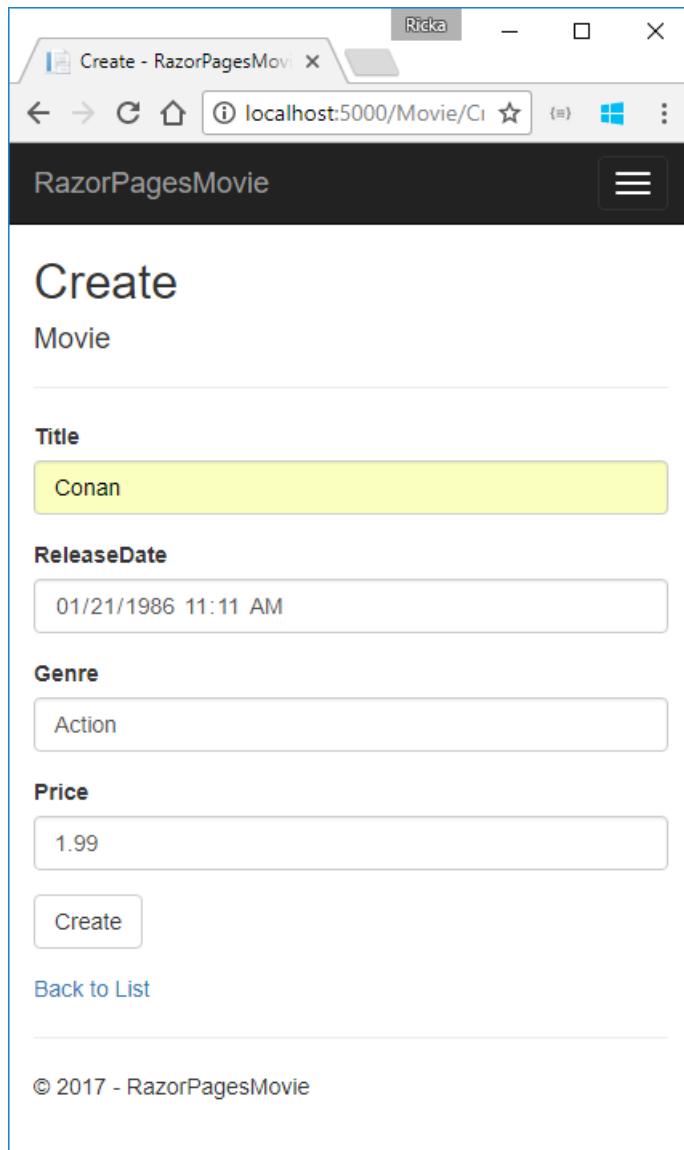
参数	描述
-m	模型的名称。
-dc	数据上下文。
-udl	使用默认布局。
-outDir	用于创建视图的相对输出文件夹路径。
--referenceScriptLibraries	向“编辑”和“创建”页面添加 <code>_ValidationScriptsPartial</code>

使用 `h` 开关获取 `aspnet-codegenerator razorpage` 命令方面的帮助：

```
dotnet aspnet-codegenerator razorpage -h
```

测试应用

- 运行应用并将 `/Movies` 追加到浏览器中的 URL (`http://localhost:port/movies`)。
- 测试“创建”链接。



- 测试“编辑”、“详细信息”和“删除”链接。

如果收到 SQL 异常，则验证是否已运行迁移并更新了数据库：

下一个教程介绍由基架创建的文件。

[上一篇：入门](#)

[下一篇：已搭建基架的 RAZOR 页面](#)

ASP.NET Core 中已搭建基架的 Razor 页面

2018/5/8 • 8 min to read • [Edit Online](#)

ASP.NET Core 中已搭建基架的 Razor 页面

作者: [Rick Anderson](#)

本教程介绍上一教程中通过搭建基架创建的 Razor 页面。

[查看或下载示例。](#)

“创建”、“删除”、“详细信息”和“编辑”页。

检查 Pages/Movies/Index.cshtml.cs 页面模型:[!code-csharp]

Razor 页面派生自 `PageModel`。按照约定, `PageModel` 派生的类称为 `<PageName>Model`。此构造函数使用 [依赖注入](#) 将 `MovieContext` 添加到页。所有已搭建基架的页面都遵循此模式。请参阅[异步代码](#), 了解有关使用实体框架的异步编程的详细信息。

对页面发出请求时, `OnGetAsync` 方法向 Razor 页面返回影片列表。在 Razor 页面上调用 `OnGetAsync` 或 `OnGet` 以初始化页面状态。在这种情况下, `OnGetAsync` 将获得影片列表并显示出来。

当 `OnGet` 返回 `void` 或 `OnGetAsync` 返回 `Task` 时, 不使用任何返回方法。当返回类型是 `IActionResult` 或 `Task<IActionResult>` 时, 必须提供返回语句。例如, Pages/Movies/Create.cshtml.cs `OnPostAsync` 方法:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

检查 Pages/Movies/Index.cshtml Razor 页面:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
    </tbody>
</table>

```

Razor 可以从 HTML 转换为 C# 或 Razor 特定标记。当 `@` 符号后跟 **Razor 保留关键字** 时，它会转换为 Razor 特定标记，否则会转换为 C#。

`@page` Razor 指令将文件转换为一个 MVC 操作 —，这意味着它可以处理请求。`@page` 必须是页面上的第一个 Razor 指令。`@page` 是转换到 Razor 特定标记的一个示例。有关详细信息，请参阅 [Razor 语法](#)。

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```

@Html.DisplayNameFor(model => model.Movie[0].Title))

```

`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。检查 Lambda 表达式(而非求值)。这意味着当 `model`、`model.Movie` 或 `model.Movie[0]` 为 `null` 或为空时，不会存在任何访问冲突。对 Lambda 表达式求值时(例如，使用 `@Html.DisplayNameFor(modelItem => item.Title)`)，将求得该模型的属性值。

@model 指令

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel
```

`@model` 指令指定传递给 Razor 页面的模型类型。在前面的示例中，`@model` 行使 `PageModel` 派生的类可用于 Razor 页面。在页面上的 `@Html.DisplayNameFor` 和 `@Html.DisplayName` HTML 帮助程序中使用该模型。

ViewData 和布局

考虑下列代码：

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel  
  
{@  
    ViewData["Title"] = "Index";  
}
```

前面突出显示的代码是 Razor 转换为 C# 的一个示例。`{` 和 `}` 字符括住 C# 代码块。

`PageModel` 基类具有 `ViewData` 字典属性，可用于添加要传递到某个视图的数据。可以使用键/值模式将对象添加到 `ViewData` 字典。在前面的示例中，“Title”属性被添加到 `ViewData` 字典。“Title”属性用于 `Pages/_Layout.cshtml` 文件。以下标记显示 `Pages/_Layout.cshtml` 文件的前几行。

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <title>@ViewData["Title"] - RazorPagesMovie</title>  
  
    /* Markup removed for brevity. */
```

行 `/* Markup removed for brevity. */` 为 Razor 注释。与 HTML 注释不同 (`<!-- -->`)，Razor 注释不会发送到客户端。

运行应用并测试项目中的链接(“主页”、“关于”、“联系人”、“创建”、“编辑”和“删除”)。每个页面都设置有标题，可以在浏览器选项卡中看到标题。将某个页面加入书签时，标题用于该书签。`Pages/Index.cshtml` 和 `Pages/Movies/Index.cshtml` 当前具有相同的标题，但可以修改它们以具有不同的值。

在 `Pages/_ViewStart.cshtml` 文件中设置 `Layout` 属性：

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

前面的标记针对所有 Razor 文件将布局文件设置为 `Pages` 文件夹下的 `Pages/_Layout.cshtml`。请参阅[布局](#)了解详细信息。

更新布局

更改 `Pages/_Layout.cshtml` 文件中的 `<title>` 元素以使用较短的字符串。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Movie</title>
```

查找 Pages/_Layout.cshtml 文件中的以下定位点元素。

```
<a asp-page="/Index" class="navbar-brand">RazorPagesMovie</a>
```

将前面的元素替换为以下标记。

```
<a asp-page="/Movies/Index" class="navbar-brand">RpMovie</a>
```

前面的定位点元素是一个[标记帮助程序](#)。此处它是[定位点标记帮助程序](#)。`asp-page="/Movies/Index"` 标记帮助程序属性和值可以创建指向 `/Movies/Index` Razor 页面的链接。

保存所做的更改，并通过单击“RpMovie”链接测试应用。请参阅 GitHub 中的 [_Layout.cshtml](#) 文件。

“创建”页面模型

检查 Pages/Movies/Create.cshtml.cs 页面模型：

```

// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public CreateModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

`OnGet` 方法初始化页面所需任何状态。“创建”页没有任何要初始化的状态。`Page` 方法创建用于呈现 Create.cshtml 页的 `PageResult` 对象。

`Movie` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。当“创建”表单发布表单值时，ASP.NET Core 运行时将发布的值绑定到 `Movie` 模型。

当页面发布表单数据时，运行 `OnPostAsync` 方法：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

如果不存在任何模型错误，将重新显示表单，以及发布的任何表单数据。在发布表单前，可以在客户端捕获到大部分模型错误。模型错误的一个示例是，发布的日期字段值无法转换为日期。我们将在本教程后面的内容中讨论有

关于客户端验证和模型验证的更多信息。

如果不存在模型错误，将保存数据，并且浏览器会重定向到索引页。

创建 Razor 页面

检查 Pages/Movies/Create.cshtml Razor 页面文件：

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />


<div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}


```

Visual Studio 以用于标记帮助程序的特殊字体显示 `<form method="post">` 标记：

The screenshot shows the Visual Studio IDE with the 'Create.cshtml.cs' file open. A tooltip is displayed over the opening tag of a form element, providing information about the FormTagHelper. The tooltip content is as follows:

```
16 The form element represents a collection of form-associated elements, some of which can represent editable  
17 values that can be submitted to a server for processing.  
18 Learn more \(F1\)  
19  
20  
21 ↳ Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper  
22 Microsoft.AspNetCore.Razor.TagHelpers.ITagHelper implementation targeting <form> elements.
```

The main code area shows the generated Razor markup for a form:

```
1  @page  
2  @model RazorPagesMovie.Pages_Movie.CreateModel  
3  
4  @{  
5      ViewData["Title"] = "Create";  
6  }  
7  
8  <h2>Create</h2>  
9  
10 <h4>Movie</h4>  
11 <hr />  
12 <div class="row">  
13     <div class="col-md-4">  
14         <form method="post">  
15  
16             The form element represents a collection of form-associated elements, some of which can represent editable  
17             values that can be submitted to a server for processing.  
18             Learn more \(F1\)  
19  
20  
21             ↳ Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper  
22             Microsoft.AspNetCore.Razor.TagHelpers.ITagHelper implementation targeting <form> elements.  
23  
24                 <span asp-validation-for="Movie.ReleaseDate" class="text-danger">  
25                     </div>  
26                 <div class="form-group">  
27                     <label asp-for="Movie.Genre" class="control-label"></label>  
28                     <input asp-for="Movie.Genre" class="form-control" />  
29                     <span asp-validation-for="Movie.Genre" class="text-danger"></span>  
30                 </div>  
31                 <div class="form-group">  
32                     <label asp-for="Movie.Price" class="control-label"></label>  
33                     <input asp-for="Movie.Price" class="form-control" />  
34                     <span asp-validation-for="Movie.Price" class="text-danger"></span>  
35                 </div>  
36                 <div class="form-group">  
37                     <input type="submit" value="Create" class="btn btn-default" />  
38                 </div>  
39             </form>  
40         </div>
```

<form method="post"> 元素是一个表单标记帮助程序。表单标记帮助程序会自动包含防伪令牌。

基架引擎在模型中为每个字段(ID 除外)创建 Razor 标记, 如下所示:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>  
<div class="form-group">  
    <label asp-for="Movie.Title" class="control-label"></label>  
    <input asp-for="Movie.Title" class="form-control" />  
    <span asp-validation-for="Movie.Title" class="text-danger"></span>  
</div>
```

验证标记帮助程序(`<div asp-validation-summary>` 和 ``)显示验证错误。本系列后面的部分将更详细地讨论有关验证的信息。

标签标记帮助程序(`<label asp-for="Movie.Title" class="control-label"></label>`)生成标签描述和 `Title` 属性的 `for` 特性。

输入标记帮助程序(`<input asp-for="Movie.Title" class="form-control" />`)使用 `DataAnnotations` 属性并在客户端生成 jQuery 验证所需的 HTML 属性。

下一教程将介绍 SQL Server LocalDB 和数据库的种子设定。

使用 SQL Server LocalDB 和 ASP.NET Core

2018/5/14 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Joe Audette](#)

`MovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using RazorPagesMovie.Models;
    // using Microsoft.EntityFrameworkCore;

    services.AddDbContext<MovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MovieContext")));
    services.AddMvc();
}
```

ASP.NET Core 配置系统会读取 `ConnectionString`。为了进行本地开发, 它会从 `appsettings.json` 文件获取连接字符串:

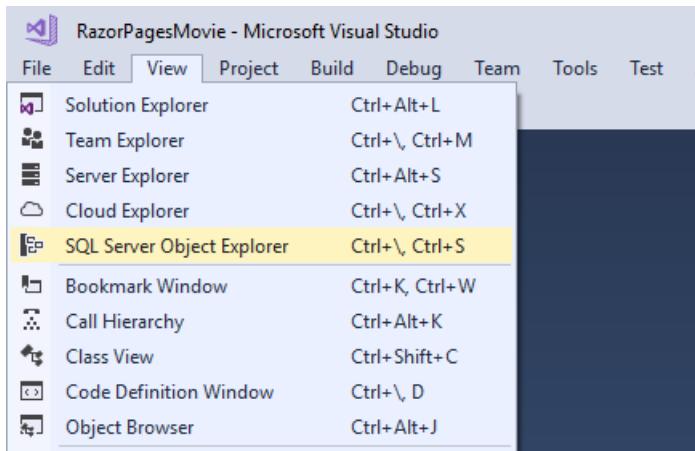
```
"ConnectionStrings": {
    "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

将应用部署到测试或生产服务器时, 可使用环境变量或另一种方法将连接字符串设置为实际的 SQL Server。有关详细信息, 请参阅[配置](#)。

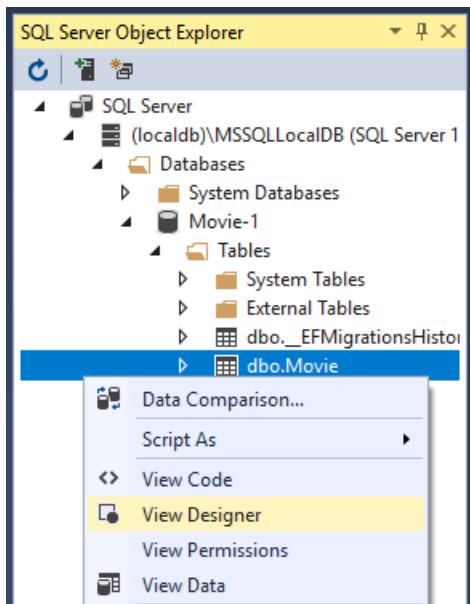
SQL Server Express LocalDB

LocalDB 是轻型版的 SQL Server Express 数据库引擎, 以程序开发为目标。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下, LocalDB 数据库在 `C:/Users/<user>` 目录中创建“*.mdf”文件。

- 从“视图”菜单中, 打开“SQL Server 对象资源管理器”(SSOX)。



- 右键单击 `Movie` 表, 然后选择“视图设计器”:



The screenshot shows the 'dbo.Movie [Design]' window. The table structure includes columns: ID (int, primary key, clustered), Genre (nvarchar(MAX)), Price (decimal(18,2)), ReleaseDate (datetime2(7)), and Title (nvarchar(MAX)). The T-SQL script pane displays the CREATE TABLE statement:

```

CREATE TABLE [dbo].[Movie] (
    [ID] INT IDENTITY (1, 1) NOT NULL,
    [Genre] NVARCHAR (MAX) NULL,
    [Price] DECIMAL (18, 2) NOT NULL,
    [ReleaseDate] DATETIME2 (7) NOT NULL,
    [Title] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);

```

请注意 `ID` 旁边的密钥图标。默认情况下，EF 为该主键创建一个名为 `ID` 的属性。

- 右键单击 `Movie` 表，然后选择“查看数据”：

The screenshot shows the 'dbo.Movie [Data]' window. The table data is as follows:

	ID	Genre	Price	ReleaseDate	Title
3	Action	1.99	1/1/0001 12:00:00 AM	Conan	
2003	Comedy	1.99	8/4/2017 6:27:31 PM	Back to the Future	
2004	Western	1.19	8/4/2017 7:08:31 PM	The Good, the bad, and the ugly	
*	NULL	NULL	NULL	NULL	NULL

设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

如果 DB 中没有任何电影，则会返回种子初始值设定项，并且不会添加任何电影。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法末端：

```
// Unused usings removed.
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

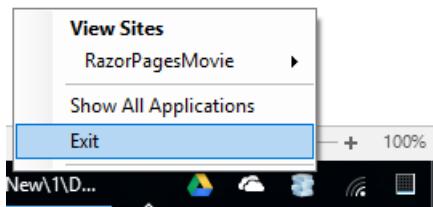
                try
                {
                    var context = services.GetRequiredService<MovieContext>();
                    // requires using Microsoft.EntityFrameworkCore;
                    context.Database.Migrate();
                    // Requires using RazorPagesMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

- 删除 DB 中的所有记录。可以使用浏览器中的删除链接，也可以从 [SSOX](#) 执行此操作
- 强制应用初始化（调用 `Startup` 类中的方法），使种子方法能够正常运行。若要强制进行初始化，必须先停止 IIS Express，然后再重新启动它。可以使用以下任一方法来执行此操作：
 - 右键单击通知区域中的 IIS Express 系统任务栏图标，然后点击“退出”或“停止站点”：



- 如果是在非调试模式下运行 VS 的, 请按 F5 以在调试模式下运行。
- 如果是在调试模式下运行 VS 的, 请停止调试程序并按 F5。

应用将显示设定为种子的数据:

Title	ReleaseDate	Genre	Price
When Harry Met Sally	2/12/1989 12:00:00 AM	Romantic Comedy	7.99
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99

在下一教程中将对数据的展示进行整理。

[上一篇：已搭建基架的 RAZOR 页](#)

[下一篇：更新页面](#)

在 ASP.NET Core 应用中更新生成的页面

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(如下图所示的 12:00:00 AM), 并且“ReleaseDate”应为“Release Date”(两个词)。

Title	ReleaseDate	Genre	Price
When Harry Met Sally	2/12/1989 12:00:00 AM	Romantic Comedy	7.99
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99

更新生成的代码

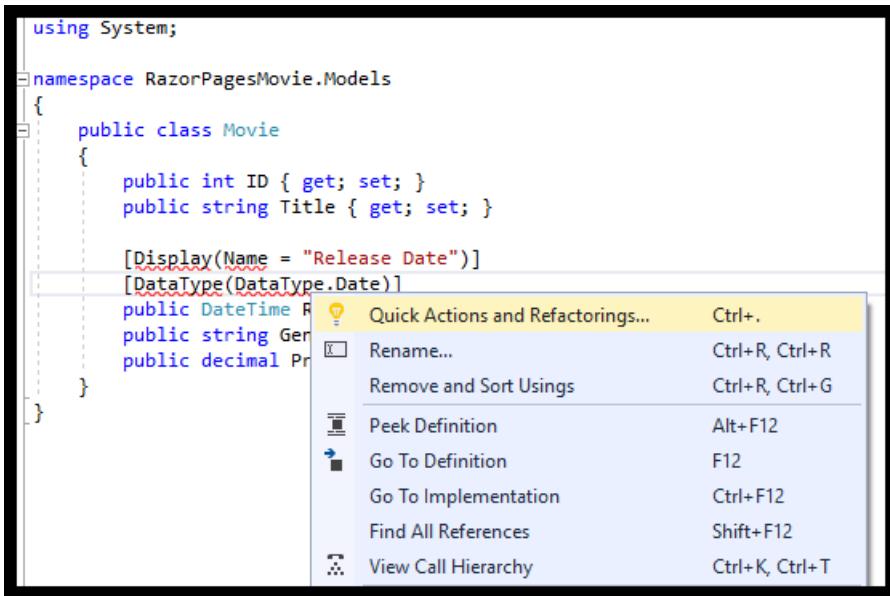
打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

```
using System;

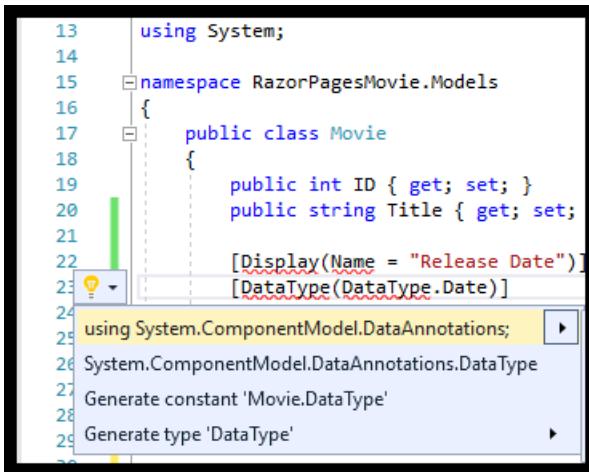
namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

右键单击红色波浪线, 然后单击“快速操作和重构”。



选择 `using System.ComponentModel.DataAnnotations;`



Visual studio 添加 `using System.ComponentModel.DataAnnotations;`。

我们将在下一教程中介绍 `DataAnnotations`。`Display` 特性指定要显示的字段名称的内容(本例中应为“Release Date”，而不是“ReleaseDate”)。`DataType` 属性指定数据的类型(日期)，使字段中存储的时间信息不会显示。

浏览到 Pages/Movies，并将鼠标悬停在“编辑”链接上以查看目标 URL。

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959	Western	3.99	Edit Details Delete

© 2017 - RazorPagesMovie

localhost:5000/Movies/Edit?id=2

“编辑”、“详细信息”和“删除”链接是在 Pages/Movies/Index.cshtml 文件中由 [定位标记帮助程序](#) 生成的。

```
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
```

[标记帮助程序](#) 使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在前面的代码中，[AnchorTagHelper](#) 从 Razor 页面(路由是相对的)、[asp-page](#) 和路由 ID ([asp-route-id](#)) 动态生成 HTML [href](#) 特性值。有关详细信息，请参阅[页面的 URL 生成](#)。

在最喜欢的浏览器中使用“查看源”来检查生成的标记。生成的 HTML 的一部分如下所示：

```
<td>
    <a href="/Movies/Edit?id=1">Edit</a> |
    <a href="/Movies/Details?id=1">Details</a> |
    <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

动态生成的链接通过查询字符串传递电影 ID(例如 `http://localhost:5000/Movies/Details?id=2`)。

更新“编辑”、“详细信息”和“删除”Razor 页面以使用“{id:int?}”路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int?}"`。运行应用，然后查看源。生成的 HTML 会将 ID 添加到 URL 的路径部分：

```
<td>
    <a href="/Movies/Edit/1">Edit</a> |
    <a href="/Movies/Details/1">Details</a> |
    <a href="/Movies/Delete/1">Delete</a>
</td>
```

如果对具有“{id: int?}”路由模板的页面进行的请求中不包含整数，则将返回 HTTP 404(未找到)错误。例如，`http://localhost:5000/Movies/Details` 将返回 404 错误。若要使 ID 可选，请将 `?>` 追加到路由约束：

```
@page "{id:int?}"
```

更新并发异常处理

在 Pages/Movies/Edit.cshtml.cs 文件中更新 `OnPostAsync` 方法。下列突出显示的代码显示了更改：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!_context.Movie.Any(e => e.ID == Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}
```

之前的代码仅检测当第一个并发客户端删除电影以及第二个并发客户端对电影发布更改时的并发异常。

测试 `catch` 块：

- 在 `catch (DbUpdateConcurrencyException)` 上设置断点
- 编辑电影。

- 在其他浏览器窗口中，选择同一电影的“删除”链接，然后删除此电影。
- 在之前的浏览器窗口中，将更改发布到电影。

当两个或更多客户端同时更新记录时，生产代码通常将检测到并发冲突。有关详细信息，请参阅[处理并发冲突](#)。

发布和绑定审阅

检查 `Pages/Movies/Edit.cshtml.cs` 文件：[!code-csharp]

当对 `Movies/Edit` 页面进行 HTTP GET 请求时（例如 `http://localhost:5000/Movies/Edit/2`）：

- `OnGetAsync` 方法从数据库提取电影并返回 `Page` 方法。
- `Page` 方法呈现“`Pages/Movies/Edit.cshtml`”Razor 页面。`Pages/Movies/Edit.cshtml` 文件包含模型指令（`@model RazorPagesMovie.Pages.Movies.EditModel`），这使电影模型在页面上可用。
- “编辑”表单中会显示电影的值。

当发布 `Movies/Edit` 页面时：

- 此页面上的表单值将绑定到 `Movie` 属性。`[BindProperty]` 特性会启用[模型绑定](#)。

```
[BindProperty]  
public Movie Movie { get; set; }
```

- 如果模型状态中存在错误（例如，`ReleaseDate` 无法被转换为日期），则会使用已提交的值再次发布表单。
- 如果没有模型错误，则电影已保存。

“索引”、“创建”和“删除”Razor 页面中的 HTTP GET 方法遵循一个类似的模式。“创建”Razor 页面中的 HTTP POST `OnPostAsync` 方法遵循的模式类似于“编辑”Razor 页面中的 `OnPostAsync` 方法所遵循的模式。

在下一教程中将添加搜索。

[上一篇：使用 SQL SERVER](#)

LOCALDB

[添加搜索](#)

将搜索添加到 ASP.NET Core Razor 页面

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本文档中，将向索引页面添加搜索功能以实现按“流派”或“名称”搜索电影。

使用以下代码更新索引页面的 `OnGetAsync` 方法：

```
public async Task OnGetAsync(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    Movie = await movies.ToListAsync();
}
```

`OnGetAsync` 方法的第一行创建了 [LINQ](#) 查询用于选择电影：

```
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义，它还不会针对数据库运行。

如果 `searchString` 参数包含一个字符串，电影查询则会被修改为根据搜索字符串进行筛选：

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

`s => s.Title.Contains()` 代码是 [Lambda 表达式](#)。Lambda 在基于方法的 [LINQ](#) 查询中用作标准查询运算符方法的参数，如 [Where](#) 方法或 `Contains`（前面的代码中所使用的）。在对 LINQ 查询进行定义或通过调用方法（如 `Where`、`Contains` 或 `OrderBy`）进行修改后，此查询不会被执行。相反，会延迟执行查询。这意味着表达式的计算会延迟，直到循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关详细信息，请参阅 [Query Execution](#)（查询执行）。

注意：`Contains` 方法在数据库中运行，而不是在 C# 代码中。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上，`Contains` 映射到 [SQL LIKE](#)，这是不区分大小写的。在 SQLite 中，由于使用了默认排序规则，因此需要区分大小写。

导航到电影页面，并向 URL 追加一个如 `?searchString=Ghost` 的查询字符串（例如 <http://localhost:5000/Movies?searchString=Ghost>）。筛选的电影将显示出来。

Title	Release Date	Genre	Price
Ghostbusters	3/13/1984	Comedy	8.99
Ghostbusters 2	2/23/1986	Comedy	9.99

© 2017 - RazorPagesMovie

如果向索引页面添加了以下路由模板，搜索字符串则可作为 URL 段传递(例如

`http://localhost:5000/Movies/ghost`)。

```
@page "{searchString?}"
```

前面的路由约束允许按路由数据(URL 段)搜索标题, 而不是按查询字符串值进行搜索。`"{searchString?}"` 中的
? 表示这是可选路由参数。

Title	Release Date	Genre	Price
Ghostbusters	3/13/1984	Comedy	8.99
Ghostbusters 2	2/23/1986	Comedy	9.99

© 2017 - RazorPagesMovie

但是, 不能指望用户修改 URL 来搜索电影。在此步骤中, 会添加 UI 来筛选电影。如果已添加路由约束
`"{searchString?}"`, 请将它删除。

打开 Pages/Movies/Index.cshtml 文件, 并添加以下代码中突出显示的 `<form>` 标记:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@

```

HTML `<form>` 标记使用表单标记帮助程序。提交表单时，筛选器字符串将发送到 Pages/Movies/Index 页面。保存更改并测试筛选器。

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

© 2017 - RazorPagesMovie

按流派搜索

将以下突出显示的属性添加到 Pages/Movies/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.MovieContext _context;

    public IndexModel(RazorPagesMovie.Models.MovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    public SelectList Genres { get; set; }
    public string MovieGenre { get; set; }
```

`SelectList Genres` 包含流派列表。这使用户能够从列表中选择一种流派。

`MovieGenre` 属性包含用户选择的特定流派(例如“西部”)。

使用以下代码更新 `OnGetAsync` 方法：

```
// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task OnGetAsync(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

下面的代码是一种 LINQ 查询，可从数据库中检索所有流派。

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;
```

流派的 `SelectList` 是通过投影截然不同的流派创建的。

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

添加“按流派搜索”

更新 `Index.cshtml`, 如下所示：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

通过按流派或/和电影标题搜索来测试应用。

[上一篇：更新页面](#)

[下一篇：添加新字段](#)

将新字段添加到 ASP.NET Core 中的 Razor 页面

2018/5/14 • 5 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中，将使用 [Entity Framework](#) Code First 迁移将新字段添加到模型，并将此更改迁移到数据库。

使用 EF Code First 自动创建数据库时，Code First 会向数据库添加表格，以帮助跟踪数据库的架构是否与从其中生成它的模型类同步。如果它们不同步，EF 则会引发异常。这使查找不一致的数据库/代码问题变得更加轻松。

向电影模型添加分级属性

打开 Models/Movie.cs 文件，并添加 Rating 属性：

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

生成应用 (Ctrl+Shift+B)。

编辑 Pages/Movies/Index.cshtml，并添加 Rating 字段：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
```

```

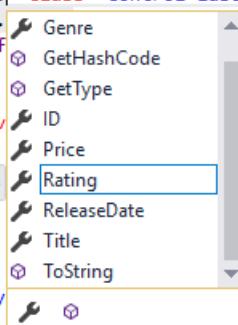
        @Html.DisplayNameFor(model => model.Movie[0].Title)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movie[0].Genre)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movie[0].Price)
    </th>
    <th>
        @Html.DisplayNameFor(model => model.Movie[0].Rating)
    </th>
    <th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

将 `Rating` 字段添加到“删除”和“详细信息”页面。

使用 `Rating` 字段更新 Create.cshtml。可以复制/粘贴之前的 `<div>` 元素，并让 intelliSense 帮助更新字段。IntelliSense 适用于[标记帮助程序](#)。

```
    </div>
    <div class="form-group">
        <label asp-for="Movie.Price" class="control-label"></label>
        <input asp-for="Movie.Price" class="form-control" />
        <span asp-validation-for="Movie.Price" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Movie." class="control-label"></label>
        <input asp-for="Movie." class="form-control" />
        <span asp-validation-for="Movie." class="text-danger"></span>
    </div>
    <div class="form-group">
        <input type="submit" value="Create" class="btn btn-default" />
    </div>
<string Models.Movie.Rating { get; set; }>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
```



下面的代码显示具有 `Rating` 字段的 Create.cshtml:

```

@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Rating" class="control-label"></label>
                <input asp-for="Movie.Rating" class="form-control" />
                <span asp-validation-for="Movie.Rating" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

将 `Rating` 字段添加到“编辑”页面。

在 DB 更新为包括新字段之前，应用将不会正常工作。如果立即运行，应用会引发 `SqlException`：

```
SqlException: Invalid column name 'Rating'.
```

此错误是由于更新的 Movie 模型类与数据库的 Movie 表架构不同导致的。（数据库表中没有 `Rating` 列。）

可通过几种方法解决此错误：

- 让 Entity Framework 自动丢弃并使用新的模型类架构重新创建数据库。此方法在开发周期早期很方便；通过它可以一起快速改进模型和数据库架构。此方法的缺点是会导致数据库中的现有数据丢失。请勿对生产数据库使用此方法！当架构更改时丢弃数据库并使用初始值设定项以使用测试数据自动设定数据库种子，这通常是开发应用的有效方式。
- 对现有数据库架构进行显式修改，使它与模型类相匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。
- 使用 Code First 迁移更新数据库架构。

对于本教程，请使用 Code First 迁移。

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 块做出此更改。

```
context.Movie.AddRange(  
    new Movie  
    {  
        Title = "When Harry Met Sally",  
        ReleaseDate = DateTime.Parse("1989-2-12"),  
        Genre = "Romantic Comedy",  
        Price = 7.99M,  
        Rating = "R"  
    },
```

请参阅[已完成的 SeedData.cs 文件](#)。

生成解决方案。

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。在 PMC 中，输入以下命令：

```
Add-Migration Rating  
Update-Database
```

`Add-Migration` 命令会通知框架执行以下操作：

- 将 `Movie` 模型与 `Movie` DB 架构进行比较。
- 创建代码以将 DB 架构迁移到新模型。

名称“Rating”是任意的，用于对迁移文件进行命名。为迁移文件使用有意义的名称是有帮助的。

如果删除 DB 中的所有记录，种子初始值设定项会设定 DB 种子，并将包括 `Rating` 字段。可以使用浏览器中的删除链接，也可以从 [Sql Server 对象资源管理器 \(SSOX\)](#) 执行此操作。从 SSOX 中删除数据库：

- 在 SSOX 中选择数据库。
- 右键单击数据库，并选择“删除”。
- 检查“关闭现有连接”。
- 选择“确定”。
- 在 [PMC](#) 中更新数据库：

```
Update-Database
```

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。如果数据库未设定种子，请先停止 IIS Express，然后再运行应用。

[上一篇：添加搜索](#)

[下一篇：添加验证](#)

将验证添加到 ASP.NET Core Razor 页面

2018/5/17 • 9 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本部分中向 `Movie` 模型添加了验证逻辑。每当用户创建或编辑电影时，都会强制执行验证规则。

验证

软件开发的一个关键原则被称为 **DRY**(即“不要自我重复”)。Razor 页面鼓励进行仅指定一次功能的开发，且功能在整个应用中反映。DRY 有助于减少应用中的代码量。DRY 使代码更加不易出错，且更易于测试和维护。

Razor 页面和 Entity Framework 提供的验证支持是 DRY 原则的极佳示例。验证规则在模型类中的某处以声明方式指定，且在应用的所有位置强制执行。

将验证规则添加到电影模型

打开 `Movie.cs` 文件。[DataAnnotations](#) 提供一组内置验证特性，可通过声明方式应用于类或属性。

`DataAnnotations` 还包含 `DataType` 等格式特性，有助于格式设置但不提供验证。

更新 `Movie` 类以使用 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性用于指定模型属性上强制执行的行为：

- `Required` 和 `MinimumLength` 特性指示属性必须具有一个值。但是，用户可以随时输入空格以对可以为 `null` 的类型进行验证约束。从本质上来说，需要不可以为 `null` 的值类型(如 `decimal`、`int`、`float` 和 `DateTime`)，但不需要 `Required` 特性。
- `RegularExpression` 特性限制用户可以输入的字符。在上述代码中，`Genre` 和 `Rating` 仅可使用字母(禁用空格、数字和特殊字符)。
- `Range` 特性将值限制在指定范围内。

- `StringLength` 特性设置字符串的最大长度，且可视情况设置最小长度。

让 ASP.NET Core 强制自动执行验证规则有助于提升应用的可靠性。自动验证模型有助于保护应用，因为添加新代码时无需手动应用它们。

Razor 页面中的“验证错误”UI

运行应用并导航到“页面/电影”。

选择“新建”链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

The screenshot shows a browser window titled "Create - Movie" at the URL "localhost:5000/Movies/Create". The page has a header "RpMovie" with a three-dot menu icon. The main content area is titled "Create" and has a sub-section "Movie". There are five input fields with validation errors:

- Title**: The input contains "a". Below it, an error message says: "The field Title must be a string with a minimum length of 3 and a maximum length of 60."
- Release Date**: The input contains "00/01/0001". Below it, an error message says: "The Release Date field is required."
- Genre**: The input contains "a". Below it, an error message says: "The field Genre must match the regular expression '^([A-Z]+[a-zA-Z'-\s]*\$)'."
- Price**: The input contains "Dog". Below it, an error message says: "The field Price must be a number."
- Rating**: The input contains "z". Below it, an error message says: "The field Rating must match the regular expression '^([A-Z]+[a-zA-Z'-\s]*\$)'."

At the bottom left is a "Create" button, and at the bottom right is a "Back to List" link.

注意

可能无法在 `Price` 字段中输入小数点或逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点及使用非美国英语日期格式的非英语区域设置，必须执行使应用全球化的步骤。有关详细信息，请参阅[其他资源](#)。目前只能输入整数，例如 10。

请注意表单如何自动呈现每个包含无效值的字段中的验证错误消息。客户端（使用 JavaScript 和 jQuery）和服务器端（若用户禁用 JavaScript）都必定会遇到这些错误。

一项重要优势是，无需在“创建”或“编辑”页面中更改代码。在模型应用 DataAnnotations 后，即已启用验证 UI。本教程中自动创建的 Razor 页面自动选取了验证规则（使用 `Movie` 模型类的属性上的验证特性）。使用“编辑”页面测试验证后，即已应用相同验证。

存在客户端验证错误时，不会将表单数据发布到服务器。请通过以下一种或多种方法验证是否未发布表单数据：

- 在 `OnPostAsync` 方法中放置一个断点。提交表单（选择“创建”或“保存”）。从未命中断点。
- 使用 [Fiddler 工具](#)。
- 使用浏览器开发人员工具监视网络流量。

服务器端验证

在浏览器中禁用 JavaScript 后，提交出错表单将发布到服务器。

（可选）测试服务器端验证：

- 在浏览器中禁用 JavaScript。如果无法在浏览器中禁用 JavaScript，请尝试其他浏览器。
- 在“创建”或“编辑”页面的 `OnPostAsync` 方法中设置断点。
- 提交带有验证错误的表单。
- 验证模型状态是否无效：

```
if (!ModelState.IsValid)
{
    return Page();
}
```

以下代码显示了之前在本教程中设定其基架的“Create.cshtml”的一部分。它用于在“创建”和“编辑”页面中显示初始表单并在发生错误后重新显示表单。

```
<form method="post">
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

输入标记帮助程序使用 [DataAnnotations](#) 特性并在客户端生成 jQuery 验证所需的 HTML 特性。[验证标记帮助程序](#) 用于显示验证错误。有关详细信息，请参阅[验证](#)。

“创建”和“编辑”页面中没有验证规则。仅可在 `Movie` 类中指定验证规则和错误字符串。这些验证规则将自动应用于编辑 `Movie` 模型的 Razor 页面。

需要更改验证逻辑时，也只能在该模型中更改。将始终在整个应用程序中应用验证（在一处定义验证逻辑）。单处验证有助于保持代码干净，且更易于维护和更新。

使用 `DataType` 特性

检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。`DataType` 特性应用于 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 特性仅提供相关提示来帮助视图引擎设置数据格式（并提供特性，例如向 URL 提供 `<a>` 和向电子邮件提供 ``）。使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型。`DataType` 特性不是验证特性。示例应用程序中仅显示日期，不显示时间。

`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可为 `DataType.EmailAddress` 创建 `mailto:` 链接。可在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`（读作 data dash）特性供 HTML 5 浏览器使用。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置用于指定在显示值进行编辑时需应用格式。可能不希望某些字段具有此行为。例如，在货币值中，可能不希望编辑 UI 中存在货币符号。

可单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能（例如显示日历控件、区域设置适用的货币符号、电子邮件链接等）
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- 借助 `DataType` 特性，ASP.NET Core 框架可选择适当的字段模板来呈现数据。单独使用时，`DisplayFormat` 特性将使用字符串模板。

注意：jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^([A-Z][a-zA-Z]*\s-]*)$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^([A-Z][a-zA-Z]*\s-*)$"), StringLength(5)]
    public string Rating { get; set; }
}
```

[Razor 页面和 EF Core 入门](#)显示了 Razor 页面的更多高级 EF Core 操作。

发布到 Azure

有关如何将该应用发布到 Azure 的说明, 请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。

其他资源

- [使用表单](#)
- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)

[上一篇 : 添加新字段](#)

[下一篇 : 上传文件](#)

将文件上传到 ASP.NET Core 中的 Razor 页面

2018/5/14 • 14 min to read • [Edit Online](#)

作者: [Luke Latham](#)

本部分演示使用 Razor 页面上传文件。

本教程中的 [Razor 页面 Movie 示例应用](#) 使用简单的模型绑定上传文件，非常适合上传小型文件。有关流式传输大文件的信息，请参阅[通过流式传输上传大文件](#)。

在下列步骤中，向示例应用添加电影计划文件上传功能。每个电影计划由一个 `Schedule` 类表示。该类包括两个版本的计划。其中一个版本 (`PublicSchedule`) 提供给客户。另一个版本 (`PrivateSchedule`) 用于公司员工。每个版本作为单独的文件进行上传。本教程演示如何通过单个 POST 将两个文件上传至服务器。

安全注意事项

向用户提供向服务器上传文件的功能时，必须格外小心。攻击者可能对系统执行[拒绝服务](#)和其他攻击。一些降低成功攻击可能性的安全措施如下：

- 将文件上传到系统上的专用文件上传区域，这样可以更轻松地对上传内容实施安全措施。如果允许文件上传，请确保在上传位置禁用执行权限。
- 使用由应用确定的安全文件名，而不是采用用户输入或已上传文件的文件名。
- 仅允许使用一组特定的已批准文件扩展名。
- 验证是否在服务器上执行了客户端检查。客户端检查很容易规避。
- 检查上传文件大小，防止上传比预期大的文件。
- 对上传的内容运行病毒/恶意软件扫描程序。

警告

将恶意代码上传到系统通常是执行代码的第一步，这些代码可以：

- 完全接管系统。
- 重载系统，导致系统完全崩溃。
- 泄露用户或系统数据。
- 将涂鸦应用于公共接口。

添加 FileUpload 类

创建 Razor 页以处理一对文件上传。添加 `FileUpload` 类（此类与页面绑定以获取计划数据）。右键单击“Models”文件夹。选择“添加”>“类”。将类命名为“FileUpload”，并添加以下属性：

```

using Microsoft.AspNetCore.Http;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class FileUpload
    {
        [Required]
        [Display(Name="Title")]
        [StringLength(60, MinimumLength = 3)]
        public string Title { get; set; }

        [Required]
        [Display(Name="Public Schedule")]
        public IFormFile UploadPublicSchedule { get; set; }

        [Required]
        [Display(Name="Private Schedule")]
        public IFormFile UploadPrivateSchedule { get; set; }
    }
}

```

此类有一个属性对应计划标题，另各有一个属性对应计划的两个版本。3个属性皆为必需属性，标题长度必须为3-60个字符。

添加用于上传文件的 helper 方法

为避免处理未上传计划文件时出现代码重复，请首先上传一个静态 helper 方法。在此应用中创建一个“Utilities”文件夹，然后在“FileHelpers.cs”文件中添加以下内容。helper 方法 `ProcessFormFile` 接受 `IFormFile` 和 `ModelStateDictionary`，并返回包含文件大小和内容的字符串。检查内容类型和长度。如果文件未通过验证检查，将向 `ModelState` 添加一个错误。

```

using System;
using System.ComponentModel.DataAnnotations;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Utilities
{
    public class FileHelpers
    {
        public static async Task<string> ProcessFormFile(IFormFile formFile, ModelStateDictionary modelState)
        {
            var fieldDisplayName = string.Empty;

            // Use reflection to obtain the display name for the model
            // property associated with this IFormFile. If a display
            // name isn't found, error messages simply won't show
            // a display name.
            MemberInfo property =
                typeof(FileUpload).GetProperty(formFile.Name.Substring(formFile.Name.IndexOf(".") + 1));

            if (property != null)
            {
                var displayAttribute =
                    property.GetCustomAttribute(typeof(DisplayAttribute)) as DisplayAttribute;

                if (displayAttribute != null)

```

```

        {
            fieldDisplayName = $"{displayAttribute.Name} ";
        }
    }

    // Use Path.GetFileName to obtain the file name, which will
    // strip any path information passed as part of the
    // FileName property. HtmlEncode the result in case it must
    // be returned in an error message.
    var fileName = WebUtility.HtmlEncode(Path.GetFileName(formFile.FileName));

    if (formFile.ContentType.ToLower() != "text/plain")
    {
        ModelState.AddModelError(formFile.Name,
                               $"The {fieldDisplayName}file ({fileName}) must be a text file.");
    }

    // Check the file length and don't bother attempting to
    // read it if the file contains no content. This check
    // doesn't catch files that only have a BOM as their
    // content, so a content length check is made later after
    // reading the file's content to catch a file that only
    // contains a BOM.
    if (formFile.Length == 0)
    {
        ModelState.AddModelError(formFile.Name, $"The {fieldDisplayName}file ({fileName}) is empty.");
    }
    else if (formFile.Length > 1048576)
    {
        ModelState.AddModelError(formFile.Name, $"The {fieldDisplayName}file ({fileName}) exceeds 1
MB.");
    }
    else
    {
        try
        {
            string fileContents;

            // The StreamReader is created to read files that are UTF-8 encoded.
            // If uploads require some other encoding, provide the encoding in the
            // using statement. To change to 32-bit encoding, change
            // new UTF8Encoding(...) to new UTF32Encoding().
            using (
                var reader =
                    new StreamReader(
                        formFile.OpenReadStream(),
                        new UTF8Encoding(encoderShouldEmitUTF8Identifier: false, throwOnInvalidBytes:
true),
                        detectEncodingFromByteOrderMarks: true))
            {
                fileContents = await reader.ReadToEndAsync();

                // Check the content length in case the file's only
                // content was a BOM and the content is actually
                // empty after removing the BOM.
                if (fileContents.Length > 0)
                {
                    return fileContents;
                }
                else
                {
                    ModelState.AddModelError(formFile.Name,
                                           $"The {fieldDisplayName}file ({fileName}) is empty.");
                }
            }
        }
        catch (Exception ex)
        {
            ModelState.AddModelError(formFile.Name,

```

```
        ModelState.AddModelError(fieldName,  
            $"The {fieldDisplayName} file ({fileName}) upload failed. " +  
            $"Please contact the Help Desk for support. Error:  
{ex.Message}");  
        // Log the exception  
    }  
}  
  
return string.Empty;  
}  
}  
}
```

将文件保存到磁盘

示例应用将已上传的文件保存到数据库字段中。要将文件保存到磁盘，请使用 [FileStream](#)。以下示例将 `FileUpload.UploadPublicSchedule` 所保存的文件复制到 `OnPostAsync` 方法中的 `FileStream`。`FileStream` 将文件写入 `<PATH-AND-FILE-NAME>` 提供的磁盘：

```
public async Task<IActionResult> OnPostAsync()  
{  
    // Perform an initial check to catch FileUpload class attribute violations.  
    if (!ModelState.IsValid)  
    {  
        return Page();  
    }  
  
    var filePath = "<PATH-AND-FILE-NAME>";  
  
    using (var fileStream = new FileStream(filePath, FileMode.Create))  
    {  
        await FileUpload.UploadPublicSchedule.CopyToAsync(fileStream);  
    }  
  
    return RedirectToPage("./Index");  
}
```

工作进程必须对 `filePath` 指定的位置具有写入权限。

注意

`filePath` 必须包含文件名。如果未提供文件名，则会在运行时引发 [UnauthorizedAccessException](#)。

警告

切勿将上传的文件保存在与应用相同的目录树中。

代码示例不提供针对恶意文件上传的服务器端保护。有关在接受用户文件时减少攻击外围应用的信息，请参阅以下资源：

- [Unrestricted File Upload\(不受限制的文件上传\)](#)
- [Azure 安全性: 确保在接受用户文件时采取适当的控制措施](#)

将文件保存到 Azure Blob 存储

若要将文件内容上传到 Azure Blob 存储，请参阅[使用 .NET 的 Azure Blob 存储入门](#)。本主题演示如何使用 `UploadFromStream` 将文件流保存到 blob 存储。

添加 Schedule 类

右键单击“Models”文件夹。选择“添加”>“类”。将类命名为“Schedule”，并添加以下属性：

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

namespace RazorPagesMovie.Models
{
    public class Schedule
    {
        public int ID { get; set; }
        public string Title { get; set; }

        public string PublicSchedule { get; set; }

        [Display(Name = "Public Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PublicScheduleSize { get; set; }

        public string PrivateSchedule { get; set; }

        [Display(Name = "Private Schedule Size (bytes)")]
        [DisplayFormat(DataFormatString = "{0:N1}")]
        public long PrivateScheduleSize { get; set; }

        [Display(Name = "Uploaded (UTC)")]
        [DisplayFormat(DataFormatString = "{0:F}")]
        public DateTime UploadDT { get; set; }
    }
}
```

此类使用 `Display` 和 `DisplayFormat` 特性，呈现计划数据时，这些特性会生成友好型的标题和格式。

更新 MovieContext

在 `MovieContext` (`Models/MovieContext.cs`) 中为计划指定 `DbSet`：

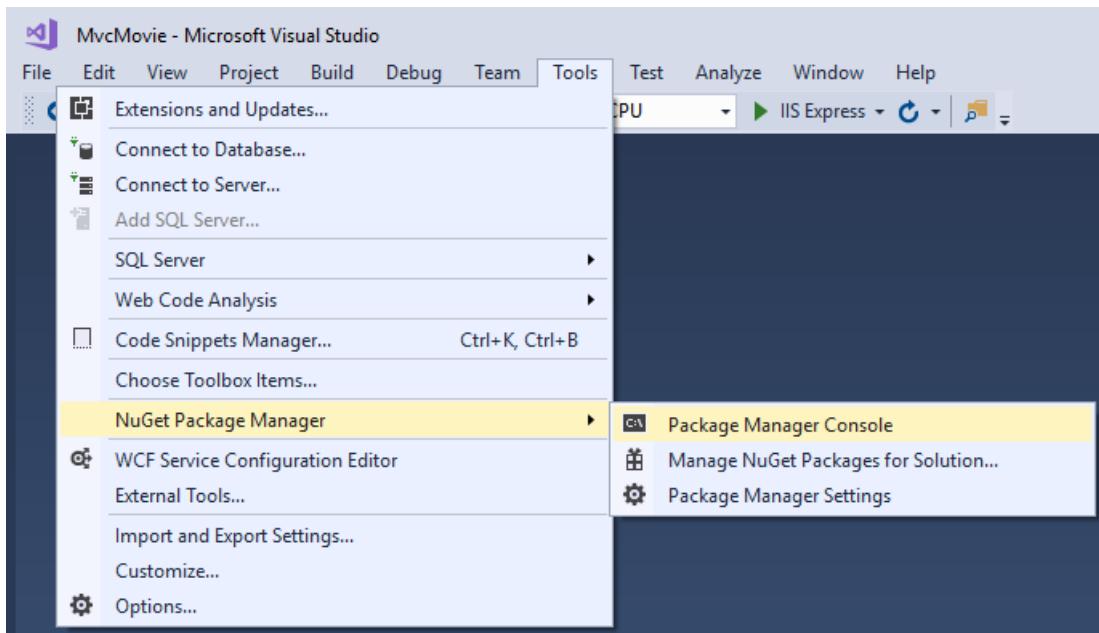
```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
        public DbSet<Schedule> Schedule { get; set; }
    }
}
```

将 Schedule 表添加到数据库

打开包管理器控制台 (PMC)：“工具”>“NuGet 包管理器”>“包管理器控制台”。



在 PMC 中执行以下命令。这些命令将向数据库添加 Schedule 表：

```
Add-Migration AddScheduleTable  
Update-Database
```

添加文件上传 Razor 页面

在“Pages”文件夹中创建“Schedules”文件夹。在“Schedules”文件夹中，创建名为“Index.cshtml”的页面，用于上传具有如下内容的计划：

```
@page  
@model RazorPagesMovie.Pages.Schedules.IndexModel  
  
{@  
    ViewData["Title"] = "Schedules";  
}  
  
<h2>Schedules</h2>  
<hr />  
  
<h3>Upload Schedules</h3>  
<div class="row">  
    <div class="col-md-4">  
        <form method="post" enctype="multipart/form-data">  
            <div class="form-group">  
                <label asp-for="FileUpload.Title" class="control-label"></label>  
                <input asp-for="FileUpload.Title" type="text" class="form-control" />  
                <span asp-validation-for="FileUpload.Title" class="text-danger"></span>  
            </div>  
            <div class="form-group">  
                <label asp-for="FileUpload.UploadPublicSchedule" class="control-label"></label>  
                <input asp-for="FileUpload.UploadPublicSchedule" type="file" class="form-control" style="height:auto" />  
                <span asp-validation-for="FileUpload.UploadPublicSchedule" class="text-danger"></span>  
            </div>  
            <div class="form-group">  
                <label asp-for="FileUpload.UploadPrivateSchedule" class="control-label"></label>  
                <input asp-for="FileUpload.UploadPrivateSchedule" type="file" class="form-control" style="height:auto" />  
                <span asp-validation-for="FileUpload.UploadPrivateSchedule" class="text-danger"></span>  
            </div>  
            <input type="submit" value="Upload" class="btn btn-default" />  
        </form>  
    </div>
```

```

</div>
</div>

<h3>Loaded Schedules</h3>
<table class="table">
    <thead>
        <tr>
            <th></th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Schedule[0].UploadDT)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PublicScheduleSize)
            </th>
            <th class="text-center">
                @Html.DisplayNameFor(model => model.Schedule[0].PrivateScheduleSize)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Schedule) {
            <tr>
                <td>
                    <a href="#" asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.UploadDT)
                </td>
                <td class="text-center">
                    @Html.DisplayFor(modelItem => item.PublicScheduleSize)
                </td>
                <td class="text-center">
                    @Html.DisplayFor(modelItem => item.PrivateScheduleSize)
                </td>
            </tr>
        }
    </tbody>
</table>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

每个窗体组包含一个 `<label>`, 它显示每个类属性的名称。`FileUpload` 模型中的 `Display` 特性提供这些标签的显示值。例如, `UploadPublicSchedule` 特性的显示名称通过 `[Display(Name="Public Schedule")]` 进行设置, 因此呈现窗体时会在此标签中显示“Public Schedule”。

每个窗体组包含一个验证 ``。如果用户输入未能满足 `FileUpload` 类中设置的属性特性, 或者任何 `ProcessFormFile` 方法文件检查失败, 则模型验证会失败。模型验证失败时, 会向用户呈现有用的验证消息。例如, `Title` 属性带有 `[Required]` 和 `[StringLength(60, MinimumLength = 3)]` 注释。用户若未提供标题, 会接收到一条指示需要提供值的消息。如果用户输入的值少于 3 个字符或多于 60 个字符, 则会接收到一条指示值长度不正确的消息。如果提供不含内容的文件, 则会显示一条指示文件为空的消息。

添加页面模型

将页面模型 (Index.cshtml.cs) 添加到“Schedules”文件夹中:

```
using System;
```

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using RazorPagesMovie.Utilities;

namespace RazorPagesMovie.Pages.Schedules
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public IndexModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public FileUpload FileUpload { get; set; }

        public IList<Schedule> Schedule { get; private set; }

        public async Task OnGetAsync()
        {
            Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            // Perform an initial check to catch FileUpload class
            // attribute violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var publicScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPublicSchedule, ModelState);

            var privateScheduleData =
                await FileHelpers.ProcessFormFile(FileUpload.UploadPrivateSchedule, ModelState);

            // Perform a second check to catch ProcessFormFile method
            // violations.
            if (!ModelState.IsValid)
            {
                Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
                return Page();
            }

            var schedule = new Schedule()
            {
                PublicSchedule = publicScheduleData,
                PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
                PrivateSchedule = privateScheduleData,
                PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
                Title = FileUpload.Title,
                UploadDT = DateTime.UtcNow
            };

            _context.Schedule.Add(schedule);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

```
}
```

页面模型 (Index.cshtml.cs 中的 `IndexModel`) 绑定 `FileUpload` 类:

```
[BindProperty]  
public FileUpload FileUpload { get; set; }
```

此模型还使用计划列表 (`IList<Schedule>`) 在页面上显示数据库中存储的计划:

```
public IList<Schedule> Schedule { get; private set; }
```

页面加载 `OnGetAsync` 时, 会从数据库填充 `Schedules`, 用于生成已加载计划的 HTML 表:

```
public async Task OnGetAsync()  
{  
    Schedule = await _context.Schedule.AsNoTracking().ToListAsync();  
}
```

将窗体发布到服务器时, 会检查 `ModelState`。如果无效, 会重新生成 `Schedule`, 且页面会呈现一个或多个验证消息, 陈述页面验证失败的原因。如果有效, `FileUpload` 属性将用于“`OnPostAsync`”中, 以完成两个计划版本的文件上传, 并创建一个用于存储数据的新 `Schedule` 对象。然后会将此计划保存到数据库:

```

public async Task<IActionResult> OnPostAsync()
{
    // Perform an initial check to catch FileUpload class
    // attribute violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var publicScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPublicSchedule, ModelState);

    var privateScheduleData =
        await FileHelpers.ProcessSchedule(FileUpload.UploadPrivateSchedule, ModelState);

    // Perform a second check to catch ProcessSchedule method
    // violations.
    if (!ModelState.IsValid)
    {
        Schedule = await _context.Schedule.AsNoTracking().ToListAsync();
        return Page();
    }

    var schedule = new Schedule()
    {
        PublicSchedule = publicScheduleData,
        PublicScheduleSize = FileUpload.UploadPublicSchedule.Length,
        PrivateSchedule = privateScheduleData,
        PrivateScheduleSize = FileUpload.UploadPrivateSchedule.Length,
        Title = FileUpload.Title,
        UploadDT = DateTime.UtcNow
    };

    _context.Schedule.Add(schedule);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

链接文件上传 Razor 页面

打开“_Layout.cshtml”，然后向导航栏添加一个链接以访问文件上传页面：

```

<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-page="/Index">Home</a></li>
        <li><a asp-page="/Schedules/Index">Schedules</a></li>
        <li><a asp-page="/About">About</a></li>
        <li><a asp-page="/Contact">Contact</a></li>
    </ul>
</div>

```

添加计划删除确认页面

用户单击删除计划时，为其提供取消此操作的机会。向“Schedules”文件夹添加删除确认页面 (Delete.cshtml)：

```

@page "{id:int}"
@model RazorPagesMovie.Pages.Schedules.DeleteModel

 @{
     ViewData["Title"] = "Delete Schedule";
 }

<h2>Delete Schedule</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Schedule</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PublicScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PublicScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.PrivateScheduleSize)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.PrivateScheduleSize)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Schedule.UploadDT)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Schedule.UploadDT)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Schedule.ID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page=".Index">Back to List</a>
    </form>
</div>

```

页面模型 (Delete.cshtml.cs) 在请求的路由数据中加载由 `id` 标识的单个计划。将“Delete.cshtml.cs”文件添加到“Schedules”文件夹：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Schedules
{
    public class DeleteModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public DeleteModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Schedule Schedule { get; set; }

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

            Schedule = await _context.Schedule.SingleOrDefaultAsync(m => m.ID == id);

            if (Schedule == null)
            {
                return NotFound();
            }
            return Page();
        }

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

            Schedule = await _context.Schedule.FindAsync(id);

            if (Schedule != null)
            {
                _context.Schedule.Remove(Schedule);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

OnPostAsync 方法按 `id` 处理计划删除：

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

    Schedule = await _context.Schedule.FindAsync(id);

    if (Schedule != null)
    {
        _context.Schedule.Remove(Schedule);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage("./Index");
}
```

成功删除计划后，`RedirectToPage` 将返回到计划的“Index.cshtml”页面。

有效的 Schedules Razor 页面

页面加载时，计划标题、公用计划和专用计划的标签和输入将呈现提交按钮：

The screenshot shows a web page titled "Upload Schedules". It contains three main sections: "Title" (a text input field), "Public Schedule" (a file input field showing "No file chosen"), and "Private Schedule" (another file input field showing "No file chosen"). Below these is an "Upload" button.

在不填充任何字段的情况下选择“上传”按钮会违反此模型上的 `[Required]` 特性。`ModelState` 无效。会向用户显示验证错误消息：

Upload Schedules

Title

The Title field is required.

Public Schedule

Choose File No file chosen

The Public Schedule field is required.

Private Schedule

Choose File No file chosen

The Private Schedule field is required.

Upload

在“标题”字段中键入两个字母。验证消息改为指示标题长度必须介于 3-60 个字符之间：

Title

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

上传一个或多个计划时，“已加载计划”部分会显示已加载计划：

Loaded Schedules

Title	Uploaded (UTC)	Public Schedule Size (bytes)	Private Schedule Size (bytes)
Delete Schedule #1	Monday, September 11, 2017 9:09:22 PM	3,805.0	2,417.0
Delete Schedule #2	Monday, September 11, 2017 9:09:35 PM	2,023.0	997.0

用户可单击该表中的“删除”链接以访问删除确认视图，并在其中选择确认或取消删除操作。

疑难解答

有关上传 `IFormFile` 的疑难解答信息，请参阅 [ASP.NET Core 中的文件上传：疑难解答](#)。

感谢读完这篇 Razor 页面简介。我们非常感谢你的反馈。[MVC 和 EF Core 入门](#)是本教程的优选后续教程。

其他资源

- [ASP.NET Core 中的文件上传](#)
- [IFormFile](#)

[上一篇：验证](#)

在带有 Visual Studio 的 Windows 上使用 ASP.NET Core MVC 创建 Web 应用

2018/4/10 • 1 min to read • [Edit Online](#)

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#)，再使用 MVC 版本。Razor 页面教程：

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面](#) 版本选择本教程，请在此 [GitHub 问题](#) 中说明原因。

本教程提供 3 个版本：

- Windows: [本系列](#)
- macOS: [使用 Visual Studio for Mac 创建 ASP.NET Core MVC 应用](#)
- macOS、Linux 和 Windows: [使用 Visual Studio Code 创建 ASP.NET Core MVC 应用](#)。本系列教程包括：

1. [入门](#)
2. [添加控制器](#)
3. [添加视图](#)
4. [添加模型](#)
5. [使用 SQL Server LocalDB](#)
6. [控制器方法和视图](#)
7. [添加搜索](#)
8. [添加新字段](#)
9. [添加验证](#)
10. [检查 Details 和 Delete 方法](#)

ASP.NET Core MVC 和 Visual Studio 入门

2018/5/17 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择, 它是基于页面的编程模型, 可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#), 再使用 MVC 版本。Razor 页面教程:

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面](#) 版本选择本教程, 请在[此 GitHub 问题](#)中说明原因。

本教程提供 3 个版本:

- macOS: [使用 Visual Studio for Mac 创建 ASP.NET Core MVC 应用](#)
- Windows: [使用 Visual Studio 创建 ASP.NET Core MVC 应用](#)
- macOS、Linux 和 Windows: [使用 Visual Studio Code 创建 ASP.NET Core MVC 应用](#)

安装 Visual Studio 和 .NET Core

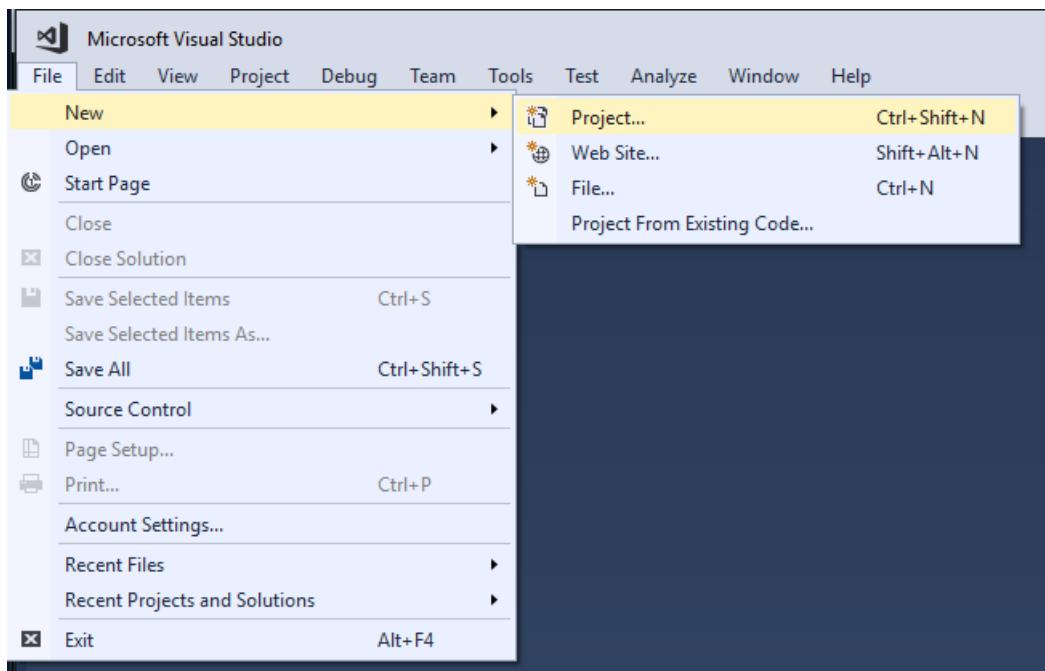
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

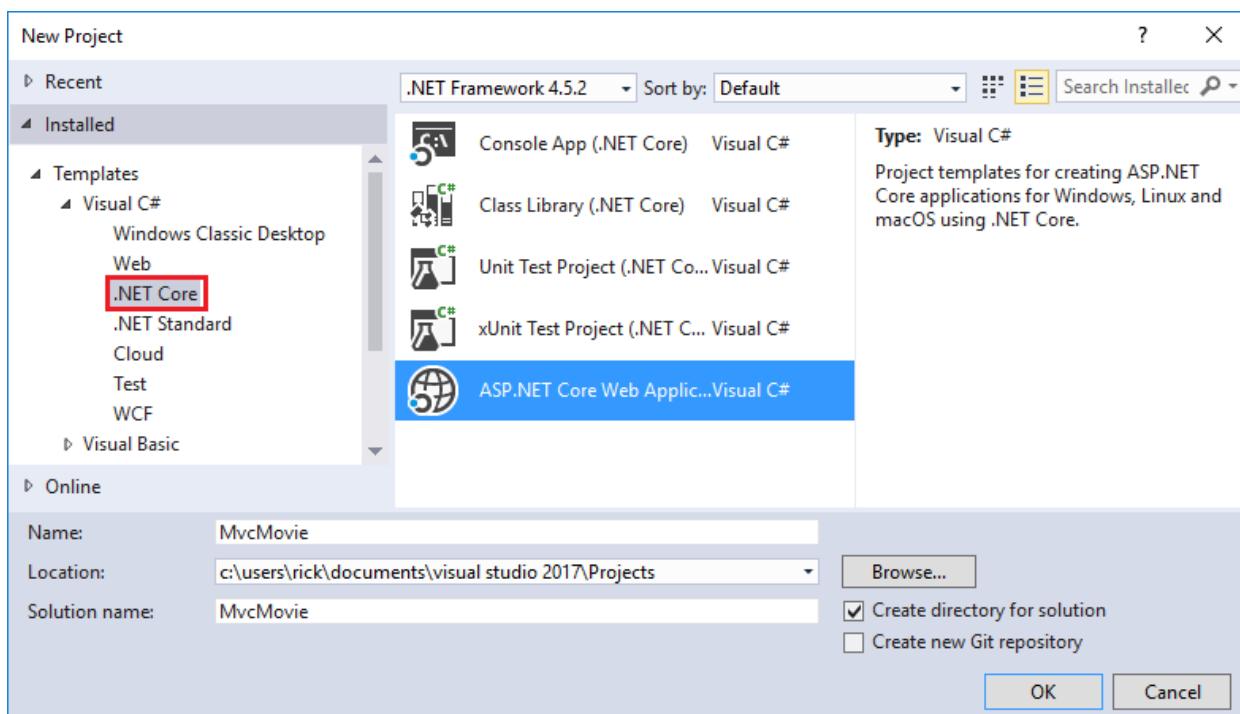
创建 Web 应用

在 Visual Studio 中, 选择“文件”>“新建”>“项目”。



填写“新建项目”对话框：

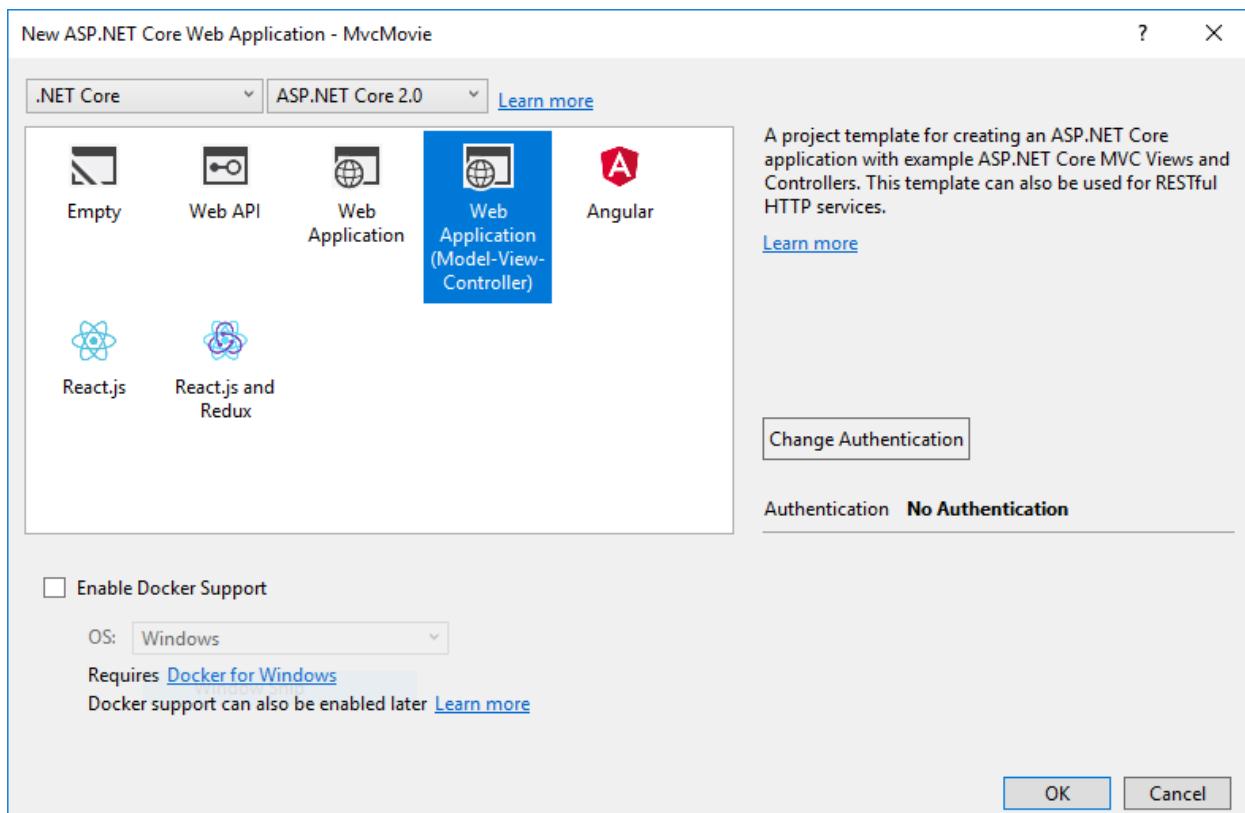
- 在左侧窗格中，点击“.NET Core”
- 在中间窗格中，点击“ASP.NET Core Web 应用程序(.NET Core)”
- 将项目命名为“MvcMovie”(请务必把项目命名为“MvcMovie”，以便在复制代码时可以与命名空间匹配。)
- 点击“确定”



- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

完成“新建 ASP.NET 核心 Web 应用程序(.NET Core) - MvcMovie”对话框：

- 在版本选择器下拉框中选择“ASP.NET Core 2.-”
- 选择“Web 应用程序(Model-View-Controller)”
- 点击“确定”



Visual Studio 为刚刚创建的 MVC 项目使用默认模板。输入项目名称并选择几个选项后，就拥有了一个可正常运行的应用。这是一个简单的初学者项目，适合入门使用。

点击“F5”在调试模式下运行应用，或按“Ctrl-F5”在非调试模式下运行应用。

MvcMovie

Microsoft Azure |

Application uses

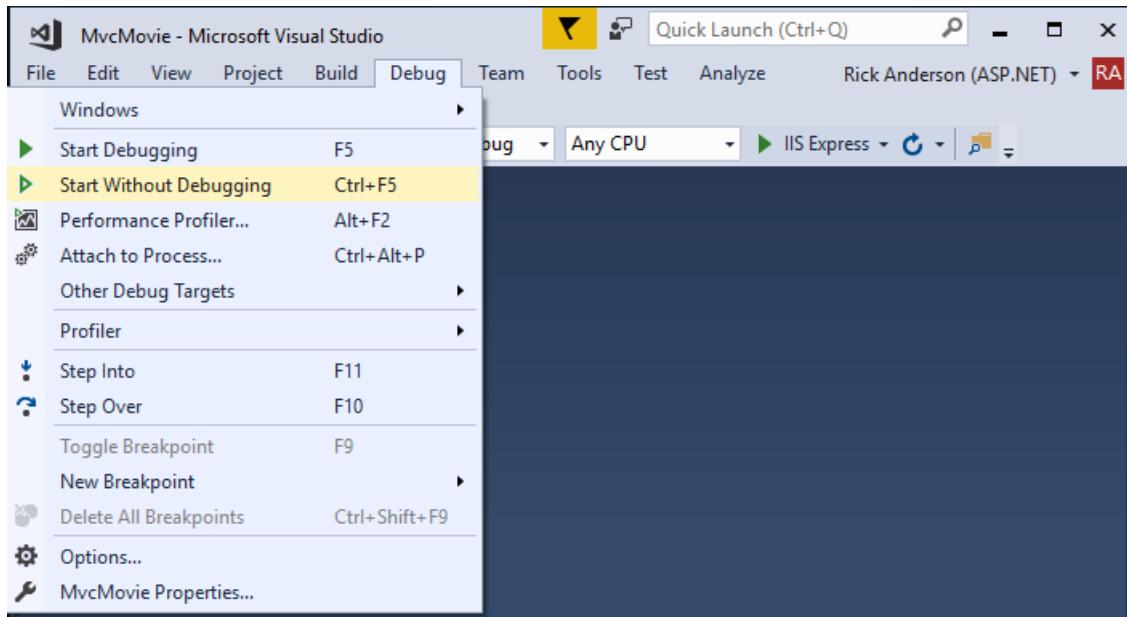
- Sample pages using ASP.NET Core MVC
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

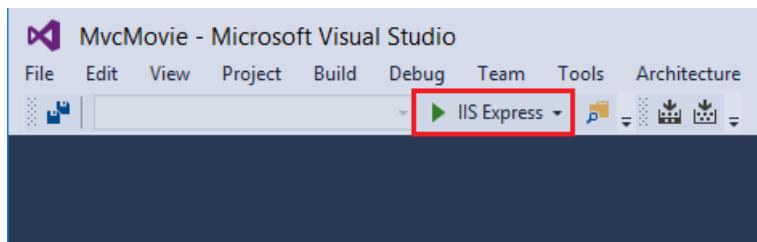
- Add a Controller and View
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet

- Visual Studio 启动 IIS Express 并运行应用。请注意，地址栏显示 `localhost:port#`，而不显示 `example.com` 之类的内容。这是因为 `localhost` 是本地计算机的标准主机名。Visual Studio 创建 Web 项目时，Web 服务器使用的是随机端口。在上图中，端口号为 5000。浏览器中的 URL 显示 `localhost:5000`。运行应用时，将看到不同的端口号。

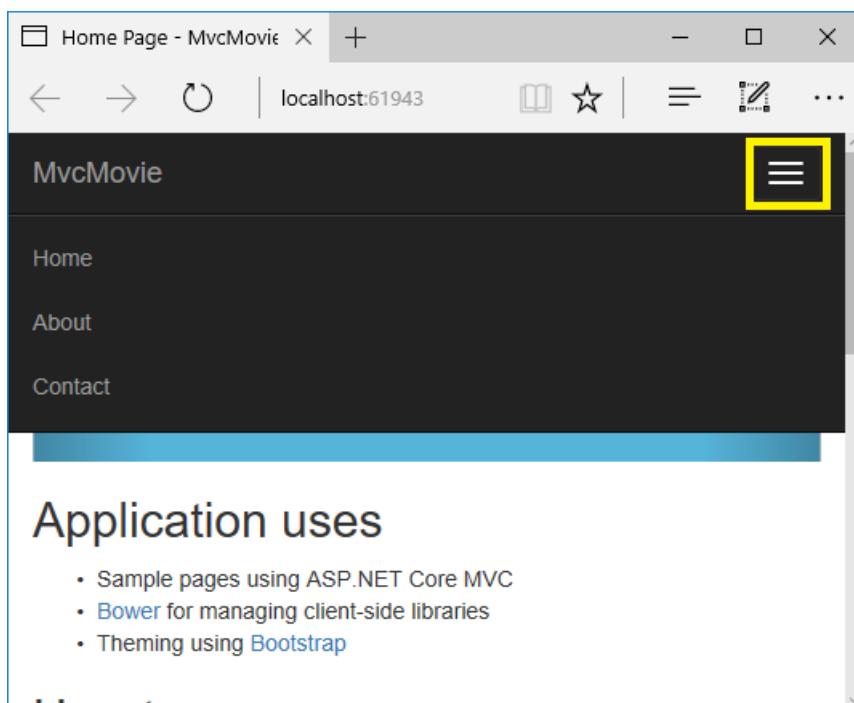
- 使用“Ctrl+F5”启动应用(非调试模式)后, 可执行代码更改、保存文件、刷新浏览器和查看代码更改等操作。许多开发人员更喜欢使用非调试模式快速启动应用并查看更改。
- 可以从“调试”菜单项中以调试或非调试模式启动应用:



- 可以通过点击“IIS Express”按钮来调试应用



默认模板提供可用的“主页”、“关于”和“联系”链接。上面的浏览器图像未显示这些链接。根据浏览器的大小, 可能需要单击导航图标才能显示这些链接。



如果正在调试模式下运行, 点击“Shift-F5”以停止调试。

在本教程的下一部分中, 我们将了解 MVC 并开始编写一些代码。

[下一篇](#)

将控制器添加到 ASP.NET Core MVC 应用

2018/5/14 • 7 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

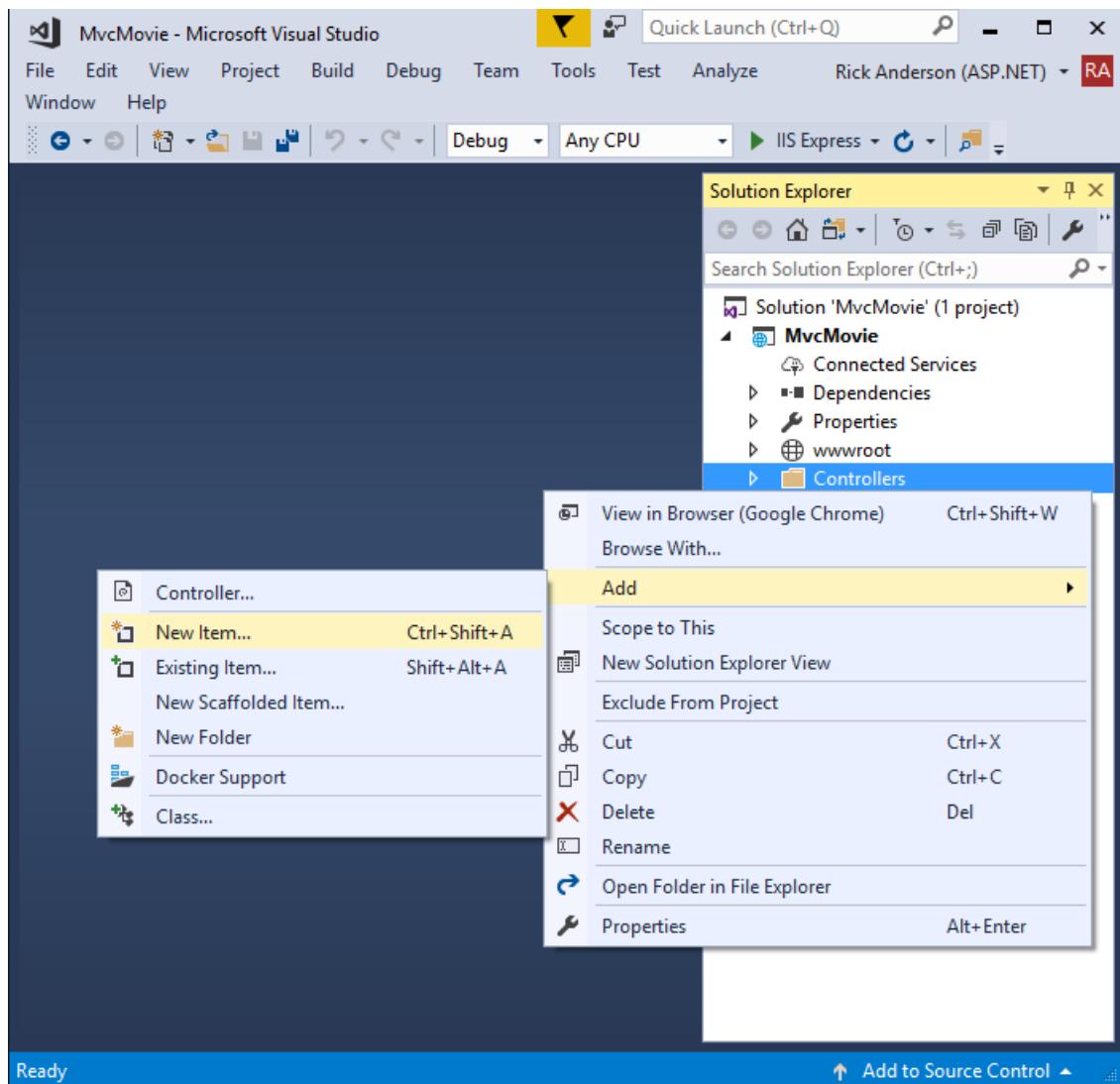
模型-视图-控制器 (MVC) 体系结构模式将应用分成 3 个主要组件: 模型 (M)、视图 (V) 和控制器 (C)。MVC 模式有助于创建比传统单片应用更易于测试和更新的应用。基于 MVC 的应用包含:

- **模式:** 表示应用数据的类。模型类使用验证逻辑来对该数据强制实施业务规则。通常, 模型对象检索模型状态并将其存储在数据库中。本教程中, `Movie` 模型将从数据库中检索电影数据, 并将其提供给视图或对其进行更新。更新后的数据将写入到数据库。
- **视图:** 视图是显示应用用户界面 (UI) 的组件。此 UI 通常会显示模型数据。
- **控制器:** 处理浏览器请求的类。它们检索模型数据并调用返回响应的视图模板。在 MVC 应用中, 视图仅显示信息; 控制器处理并响应用户输入和交互。例如, 控制器处理路由数据和查询字符串值, 并将这些值传递给模型。该模型可使用这些值查询数据库。例如, `http://localhost:1234/Home/About` 具有 `Home` (控制器) 的路由数据和 `About` (在 `Home` 控制器上调用的操作方法)。`http://localhost:1234/Movies/Edit/5` 是一个请求, 用于通过电影控制器编辑 ID 为 5 的电影。本教程稍后将探讨路由数据。

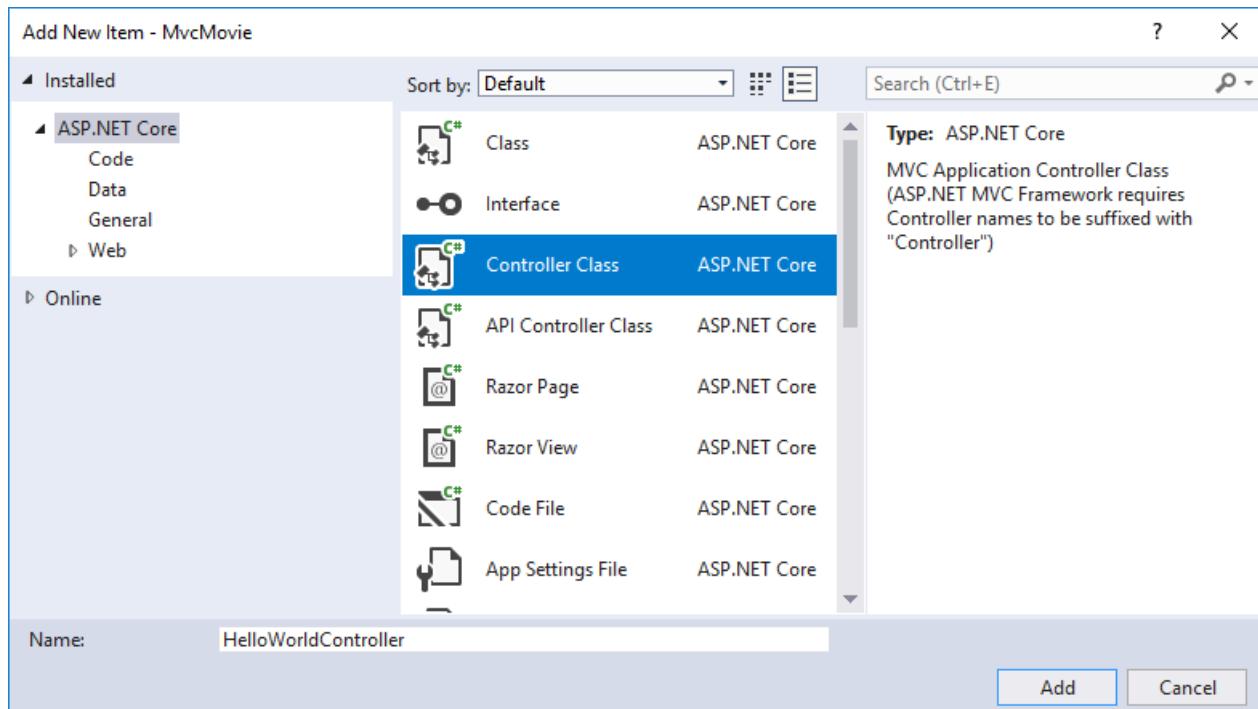
MVC 模式可帮助创建分隔不同应用特性(输入逻辑、业务逻辑和 UI 逻辑)的应用, 同时让这些元素之间实现松散耦合。该模式可指定应用中每种逻辑的位置。UI 逻辑位于视图中。输入逻辑位于控制器中。业务逻辑位于模型中。这种隔离有助于控制构建应用时的复杂程度, 因为它可用于一次处理一个实现特性, 而不影响其他特性的代码。例如, 处理视图代码时不必依赖业务逻辑代码。

本教程系列中介绍了这些概念, 并展示了如何使用它们构建电影应用。MVC 项目包含“控制器”和“视图”文件夹。

- 在“解决方案资源管理器”中, 右键单击“控制器”, 选择“添加”>“新项”



- 选择“控制器类”
- 在“添加新项”对话框中，输入“HelloWorldController”。



将“Controllers/HelloWorldController.cs”的内容替换为以下内容：

```

using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}

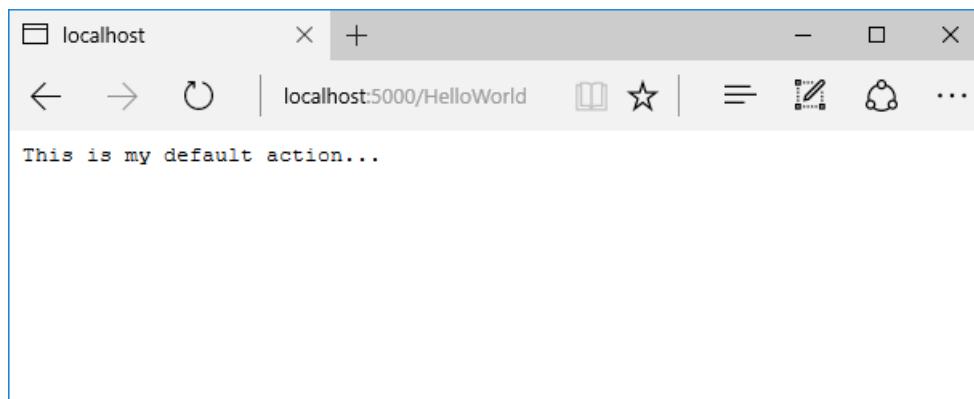
```

控制器中的每个 `public` 方法均可作为 HTTP 终结点调用。上述示例中，两种方法均返回一个字符串。请注意每个方法前面的注释。

HTTP 终结点是 Web 应用程序中可定向的 URL(例如 `http://localhost:1234/HelloWorld`)，其中结合了所用的协议 `HTTP`、`TCP` 端口等 Web 服务器的网络位置 `localhost:1234`，以及目标 URI `HelloWorld`。

第一条注释指出这是一个 `HTTP GET` 方法，它通过向基 URL 追加“/HelloWorld/”进行调用。第二条注释指定一个 `HTTP GET` 方法，它通过向 URL 追加“/HelloWorld/Welcome/”进行调用。本教程稍后将使用基架引擎生成 `HTTP POST` 方法。

在非调试模式下运行应用，并将“HelloWorld”追加到地址栏中的路径。`Index` 方法返回一个字符串。



MVC 根据入站 URL 调用控制器类(及其中的操作方法)。MVC 所用的默认 URL 路由逻辑使用如下格式来确定调用的代码：

`/[Controller]/[ActionName]/[Parameters]`

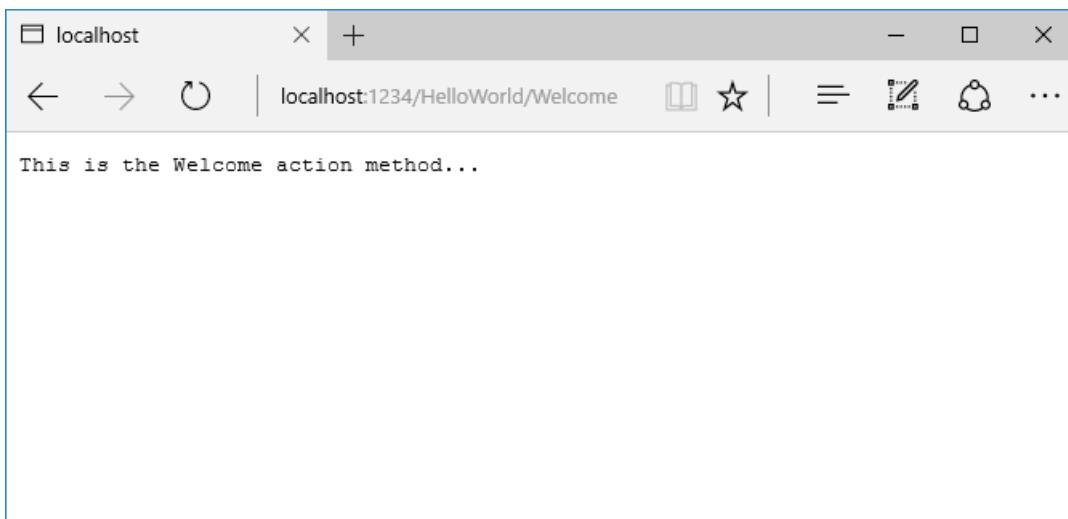
在 `Startup.cs` 文件的 `Configure` 方法中设置路由的格式。

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

如果运行应用且不提供任何 URL 段，它将默认为上面突出显示的模板行中指定的“Home”控制器和“Index”方法。

第一个 URL 段决定要运行的控制器类。因此 `localhost:xxxx/HelloWorld` 映射到 `HelloWorldController` 类。该 URL 段的第二部分决定类上的操作方法。因此 `localhost:xxxx/HelloWorld/Index` 将触发 `HelloWorldController` 类的 `Index` 运行。请注意，只需浏览到 `localhost:xxxx/HelloWorld`，而 `Index` 方法默认调用。原因是 `Index` 是默认方法，如果未显式指定方法名称，则将在控制器上调用它。URL 段的第三部分 (`id`) 针对的是路由数据。稍后可在本教程中看到路由数据。

浏览到 `http://localhost:xxxx/HelloWorld/Welcome`。`Welcome` 方法运行并返回字符串“这是 Welcome 操作方法...”。对于此 URL，采用 `HelloWorld` 控制器和 `Welcome` 操作方法。目前尚未使用 URL 的 `[Parameters]` 部分。



修改代码，将一些参数信息从 URL 传递到控制器。例如 `/HelloWorld/Welcome?name=Rick&numtimes=4`。更改 `Welcome` 方法以包括以下代码中显示的两个参数：

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

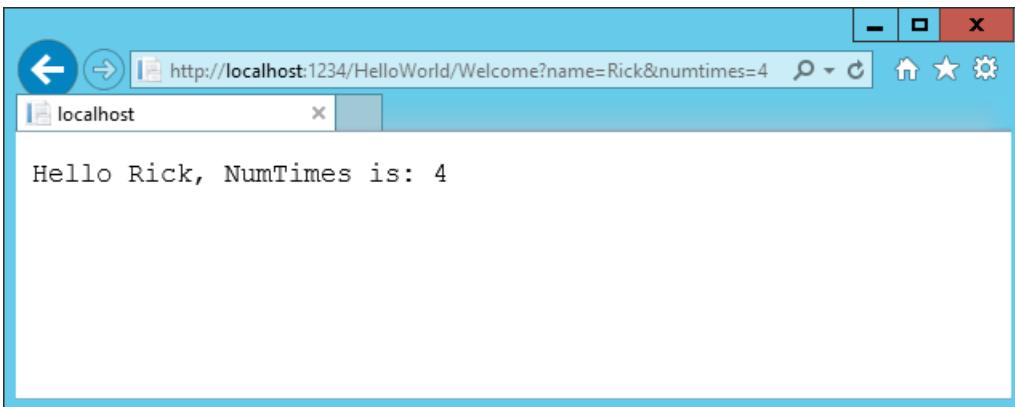
前面的代码：

- 使用 C# 可选参数功能指示，未为 `numTimes` 参数传递值时该参数默认为 1。
- 使用 `HtmlEncoder.Default.Encode` 防止恶意输入（即 JavaScript）损害应用。
- 使用内插字符串。

运行应用并浏览到：

`http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

（将 xxxx 替换为端口号。）可在 URL 中对 `name` 和 `numtimes` 使用其他值。MVC 模型绑定系统可将命名参数从地址栏中的查询字符串自动映射到方法中的参数。有关详细信息，请参阅[模型绑定](#)。

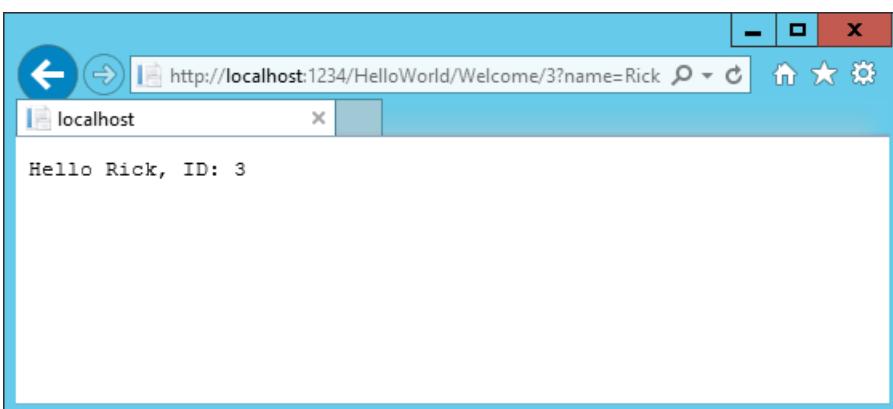


在上图中，未使用 URL 段（Parameters），且 name 和 numTimes 参数作为查询字符串进行传递。上述 URL 中的 ? (问号) 为分隔符，后接查询字符串。& 字符用于分隔查询字符串。

将 Welcome 方法替换为以下代码：

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

运行应用并输入以下 URL：http://localhost:xxx/HelloWorld/Welcome/3?name=Rick



此时，第三个 URL 段与路由参数 id 相匹配。Welcome 方法包含 MapRoute 方法中匹配 URL 模板的参数 id。后面的 ? (id?) 中表示 id 参数可选。

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

上述示例中，控制器始终执行 MVC 的“VC”部分，即视图和控制器工作。控制器将直接返回 HTML。通常不希望控制器直接返回 HTML，因为编码和维护非常繁琐。通常，需使用单独的 Razor 视图模板文件来帮助生成 HTML 响应。可在下一教程中执行该操作。

在 Visual Studio 的非调试模型下 (Ctrl+F5)，不需要在更改代码后生成应用。只需要保存文件并更新浏览器，就可以看到所做的更改。

将视图添加到 ASP.NET Core MVC 应用

2018/5/14 • 9 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将修改 `HelloWorldController` 类, 进而使用 Razor 视图模板文件来顺利封装为客户端生成 HTML 响应的过程。

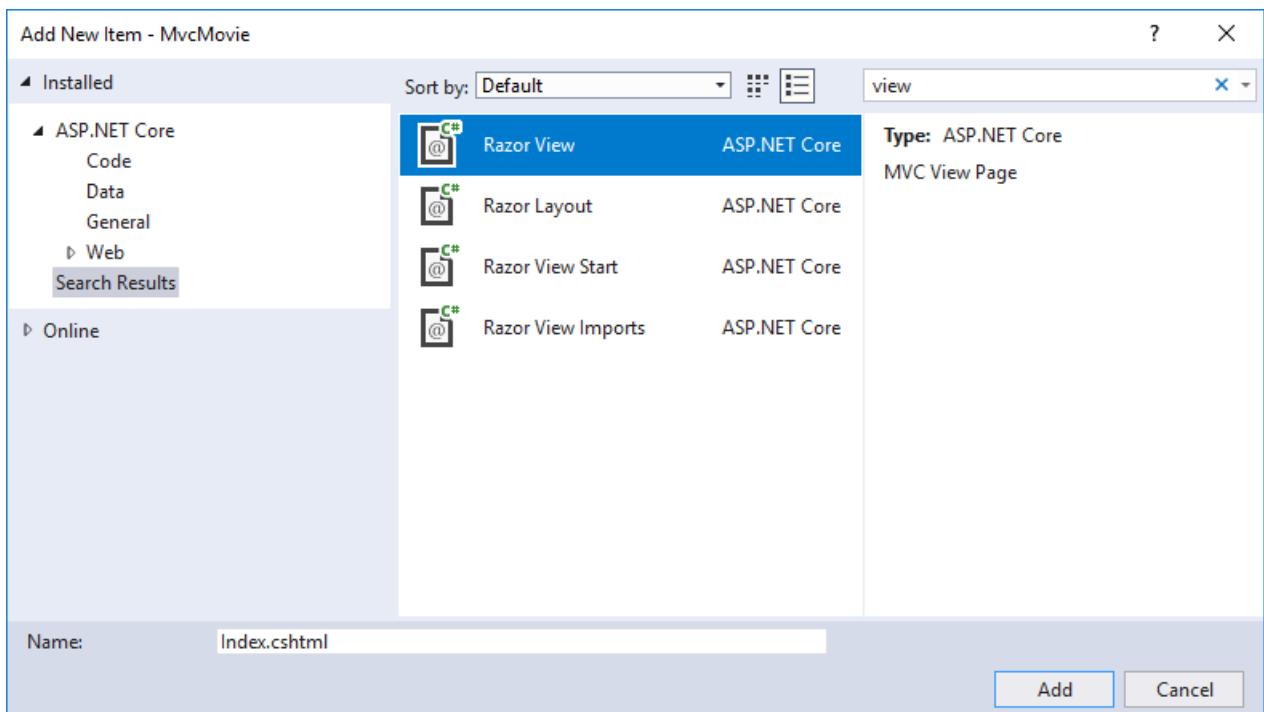
使用 Razor 创建视图模板文件。基于 Razor 的模板具有".cshtml"文件扩展名。它们提供了一种巧妙的方法来使用 C# 创建 HTML 输出。

当前, `Index` 方法返回带有在控制器类中硬编码的消息的字符串。在 `HelloWorldController` 类中, 将 `Index` 方法替换为以下代码:

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

上述代码返回 `View` 对象。它使用视图模板对浏览器生成 HTML 响应。类似上述 `Index` 方法的控制器方法(也称为操作方法)通常返回 `IActionResult`(或派生自 `ActionResult` 的类), 而非类似字符串的类型。

- 右键单击“视图”文件夹, 然后单击“添加”>“新文件夹”, 并将文件夹命名为“HelloWorld”。
- 右键单击“Views/HelloWorld”文件夹, 然后单击“添加”>“新项”。
- 在“添加新项 - MvcMovie”对话框中
 - 在右上角的搜索框中, 输入“视图”
 - 点击“Razor 视图”
 - 如有必要, 在“名称”框中将名称更改为“Index.cshtml”。
 - 点击“添加”



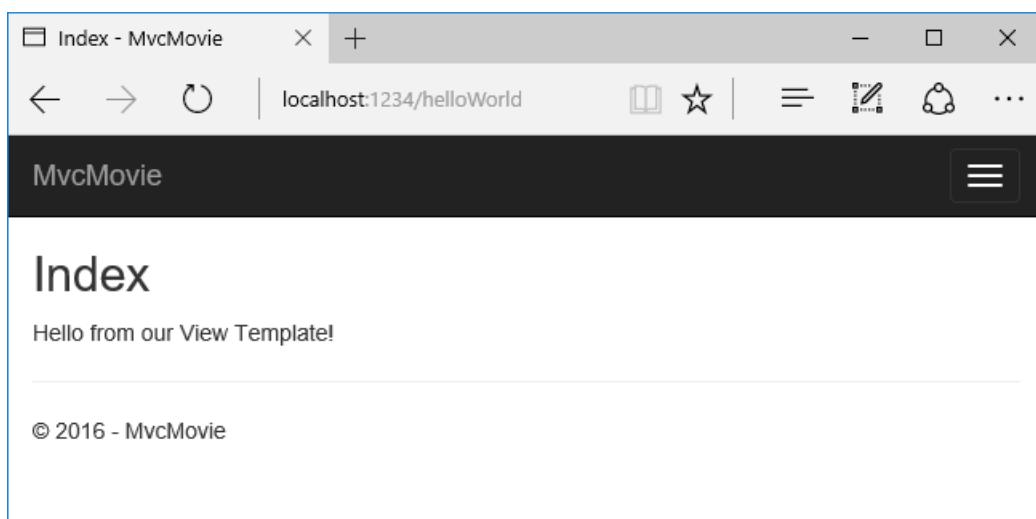
使用以下内容替换 Razor 视图文件 Views/HelloWorld/Index.cshtml 的内容：

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

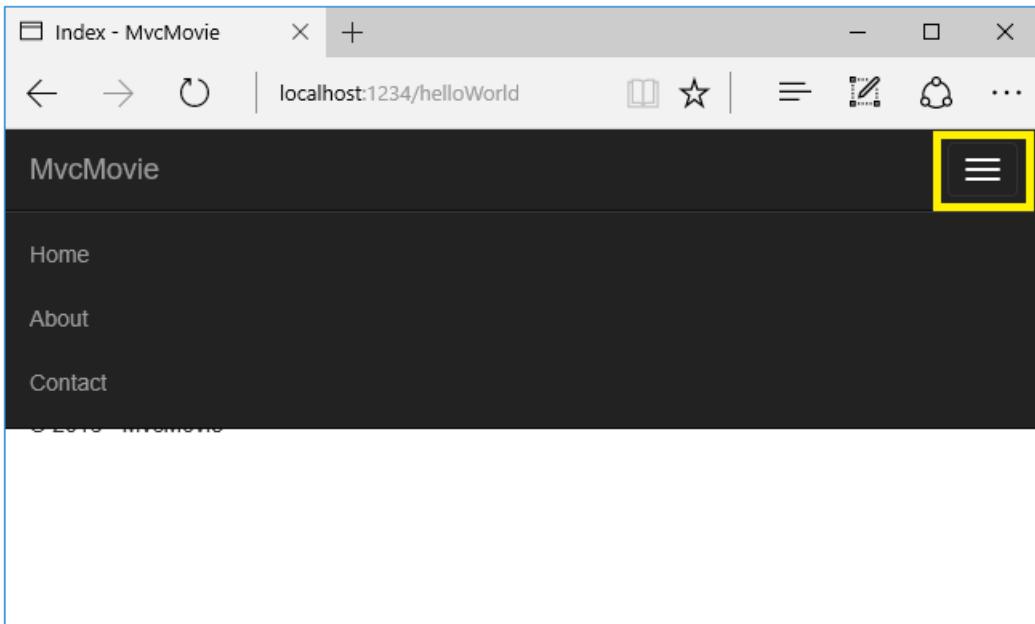
<h2>Index</h2>

<p>Hello from our View Template!</p>
```

导航到 `http://localhost:xxxx/HelloWorld`。`HelloWorldController` 中的 `Index` 方法作用不大；它运行 `return View();` 语句，指定此方法应使用视图模板文件来呈现对浏览器的响应。因为没有显式指定视图模板文件的名称，所以 MVC 默认使用 /Views/HelloWorld 文件夹中的 `Index.cshtml` 视图文件。下面图片显示了视图中硬编码的字符串“Hello from our View Template!”



如果浏览器窗口较小（例如在移动设备上），则可能需要在右上角切换（点击）Bootstrap 导航按钮以查看“首页”、“关于”和“联系人”链接。



更改视图和布局页面

点击菜单链接（“MvcMovie”、“首页”、“关于”）。每页显示相同的菜单布局。菜单布局是在 Views/Shared/_Layout.cshtml 文件中实现的。打开 Views/Shared/_Layout.cshtml 文件。

布局模板使你能够在一个位置指定网站的 HTML 容器布局，然后将它应用到网站中的多个页面。查找 `@RenderBody()` 行。`RenderBody` 是显示创建的所有特定于视图的页面的占位符，已包装在布局页面中。例如，如果选择“关于”链接，Views/Home/About.cshtml 视图将在 `RenderBody` 方法中呈现。

更改布局文件中的标题和菜单链接

在标题元素中，将 `MvcMovie` 更改为 `Movie App`。将布局模板中的定位文本从 `MvcMovie` 更改为 `Movie App`，并将控制器从 `Home` 更改为 `Movies`，如下所示：

注意：ASP.NET Core 2.0 版本略有不同。它不包含 `@inject ApplicationInsights` 和 `@Html.Raw(JavaScriptSnippet.FullScript)`。

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Movie App</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href("~/css/site.css") />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
              value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
    @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
```

```

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
    <span class="sr-only">Toggle navigation</span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>
<a asp-area="" asp-controller="Movies" asp-action="Index" class="navbar-brand">Movie App</a>
</div>
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
        <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
        <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
    </ul>
</div>
</div>
</nav>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2017 - MvcMovie</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-Tc5IQib027qvjSMfHj0MaLkfUWVxZxUPnCJA712mCWNIPG9mGCD8wGNICPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

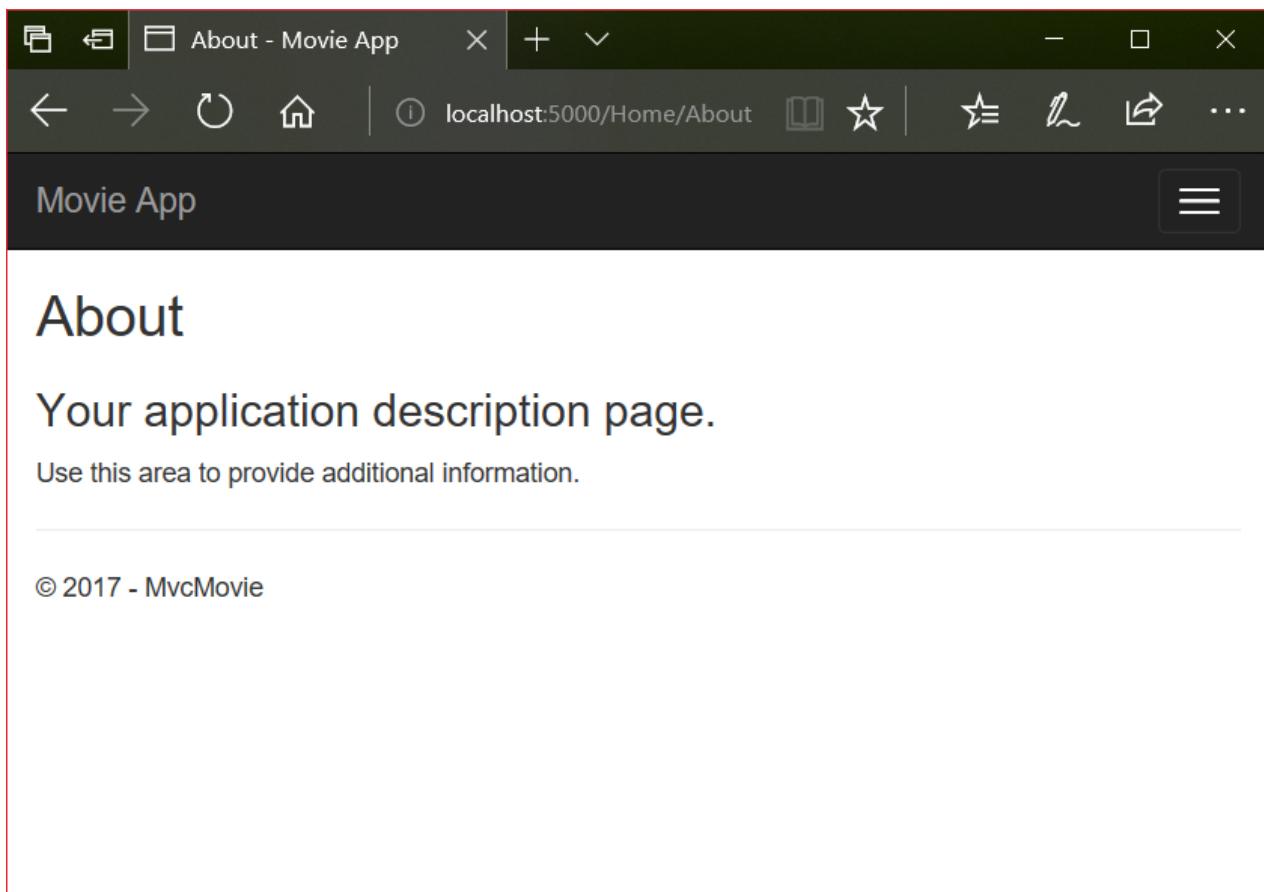
    @RenderSection("Scripts", required: false)
</body>
</html>

```

警告

我们尚未实现 `Movies` 控制器，所以如果单击此链接，将收到 404(未找到)错误。

保存更改并点击“关于”链接。请注意浏览器选项卡上的标题现在显示的是“关于 - 电影应用”，而不是“关于 - **Mvc 电影**”：



点击“联系人”链接，请注意，标题和定位文本还会显示“电影应用”。我们能够在布局模板中进行一次更改，让网站上的所有页面都反映新的链接文本和新标题。

检查 Views/_ViewStart.cshtml 文件：

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

Views/_ViewStart.cshtml 文件将 Views/Shared/_Layout.cshtml 文件引入到每个视图中。可以使用 `Layout` 属性设置不同的布局视图，或将它设置为 `null`，这样将不会使用任何布局文件。

更改 `Index` 视图的标题。

打开 Views/HelloWorld/Index.cshtml。有两处需要更改的地方：

- 浏览器标题中显示的文字。
- 辅助标题 (`<h2>` 元素)。

稍稍对它们进行一些更改，这样可以看出哪一段代码更改了应用的哪部分。

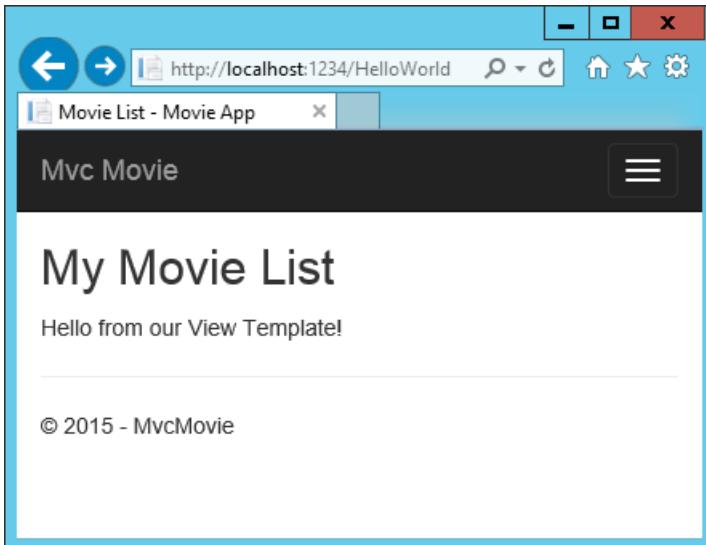
```
@{  
    ViewData["Title"] = "Movie List";  
}  
  
<h2>My Movie List</h2>  
  
<p>Hello from our View Template!</p>
```

上述代码中的 `ViewData["Title"] = "Movie List";` 将 `ViewData` 字典的 `Title` 属性设置为“Movie List”。`Title` 属性在布局页面中的 `<title>` HTML 元素中使用：

```
<title>@ViewData["Title"] - Movie App</title>
```

保存更改并导航到 <http://localhost:xxxx>HelloWorld>。请注意，浏览器标题、主标题和辅助标题已更改。(如果没有在浏览器中看到更改，则可能正在查看缓存的内容。在浏览器中按 Ctrl + F5 强制加载来自服务器的响应。)浏览器标题是使用我们在 Index.cshtml 视图模板中设置的 `ViewData["Title"]` 以及在布局文件中添加的额外“- Movie App”创建的。

还要注意，Index.cshtml 视图模板中的内容是如何与 Views/Shared/_Layout.cshtml 视图模板合并的，并且注意单个 HTML 响应被发送到了浏览器。凭借布局模板可以很容易地对应用程序中所有页面进行更改。若要了解更多信息，请参阅[布局](#)。



但我们一点点“数据”(在此示例中为“Hello from our View Template!”消息)是硬编码的。MVC 应用程序有一个“V”(视图)，而你已有一个“C”(控制器)，但还没有“M”(模型)。

将数据从控制器传递给视图

控制器操作会被调用以响应传入的 URL 请求。控制器类是编写处理传入浏览器请求的代码的地方。控制器从数据源检索数据，并决定将哪些类型的响应发送回浏览器。可以从控制器使用视图模板来生成并格式化对浏览器的 HTML 响应。

控制器负责提供所需的数据，使视图模板能够呈现响应。最佳做法：视图模板不应该直接执行业务逻辑或与数据库进行交互。相反，视图模板应仅使用由控制器提供给它的数据。保持此“关注点分离”有助于保持代码干净、可测试性和可维护性。

目前，`HelloWorldController` 类中的 `Welcome` 方法采用 `name` 和 `ID` 参数，然后将值直接输出到浏览器。应将控制器更改为使用视图模板，而不是使控制器将此响应呈现为字符串。视图模板会生成动态响应，这意味着必须将适当的数据位从控制器传递给视图以生成响应。为此，可以让控制器将视图模板所需的动态数据(参数)放置在视图模板稍后可以访问的 `ViewData` 字典中。

返回到 `HelloWorldController.cs` 文件，并更改 `Welcome` 方法以将 `Message` 和 `NumTimes` 值添加到 `ViewData` 字典。`ViewData` 字典是一个动态对象，这意味着你可以将任何所需的内容放在其中；只有将内容放在其中后 `ViewData` 对象才具有定义的属性。[MVC 模型绑定系统](#)自动将命名参数(`name` 和 `numTimes`)从地址栏中的查询字符串映射到方法中的参数。完整的 `HelloWorldController.cs` 文件如下所示：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

`ViewData` 字典对象包含将传递给视图的数据。

创建一个名为 Views/HelloWorld/Welcome.cshtml 的 Welcome 视图模板。

在 Welcome.cshtml 视图模板中创建一个循环，显示“Hello” `NumTimes`。使用以下内容替换 Views/HelloWorld/Welcome.cshtml 的内容：

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

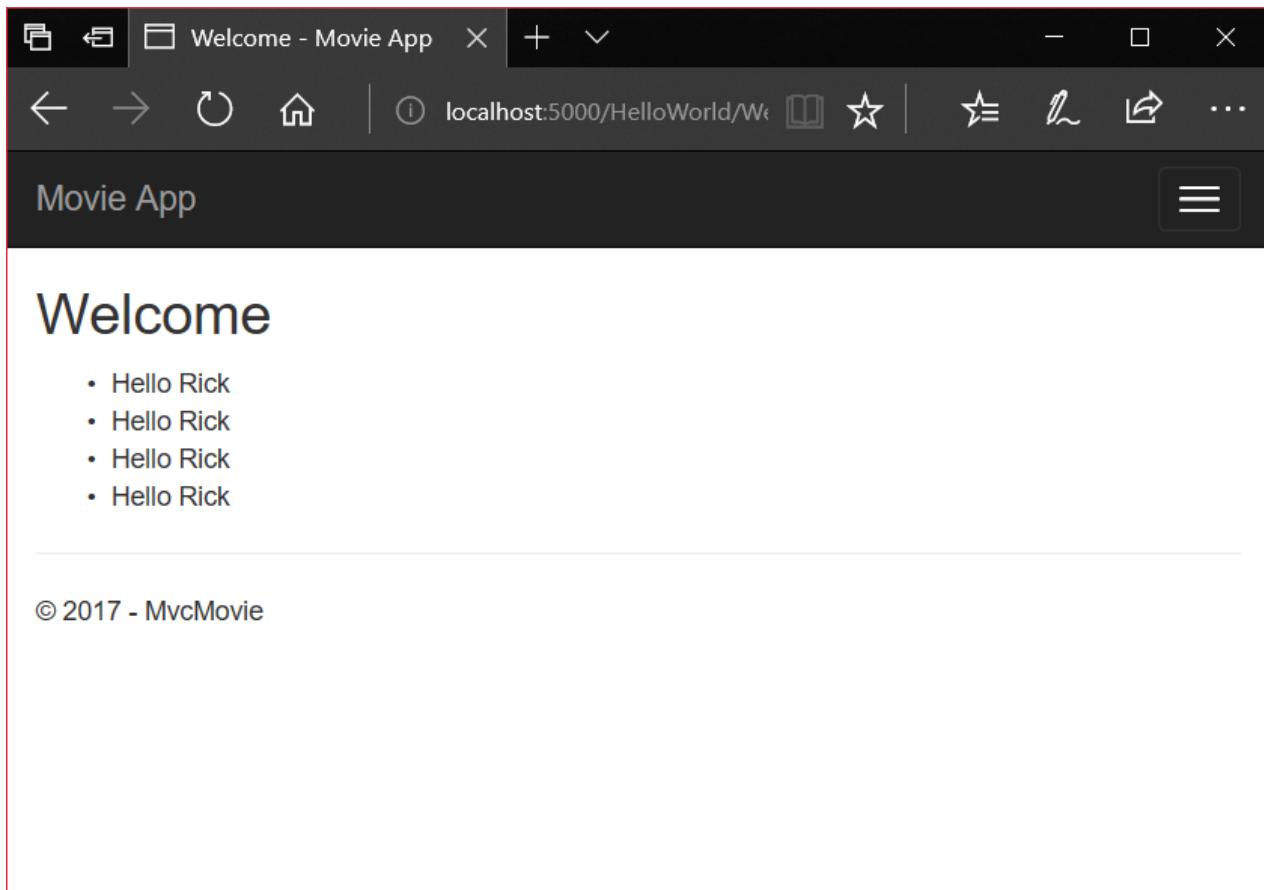
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

保存更改并浏览到以下 URL：

```
http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4
```

数据取自 URL，并传递给使用 [MVC 模型绑定器](#) 的控制器。控制器将数据打包到 `ViewData` 字典中，并将该对象传递给视图。然后，视图将数据作为 HTML 呈现给浏览器。



在上面的示例中，我们使用 `ViewData` 字典将数据从控制器传递给视图。稍后在本教程中，我们将使用视图模型将数据从控制器传递给视图。传递数据的视图模型方法通常比 `ViewData` 字典方法更为优先。有关详细信息，请参阅 [ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC](#) (MVC 中 ViewModel、ViewData、ViewBag、TempData 和 Session 之间的比较)。

当然，这是模型的一种“M”类型，而不是数据库类。让我们用学到的内容来创建一个电影数据库。

[上一页](#) [下一页](#)

将模型添加到 ASP.NET Core MVC 应用

2018/5/14 • 11 min to read • [Edit Online](#)

将模型添加到 ASP.NET Core MVC 应用

作者: [Rick Anderson](#) 和 [Tom Dykstra](#)

在本部分中，将添加用于管理数据库中电影的一些类。这些类将是 MVC 应用的“Model”部分。

可以结合 [Entity Framework Core](#) (EF Core) 使用这些类来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化需要编写的数据访问代码。[EF Core 支持许多数据库引擎](#)。

要创建的模型类称为 POCO 类(源自“普通旧 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们只定义将存储在数据库中的数据的属性。

在本教程中，首先将编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍)是从已存在的数据库生成模型类。有关此方法的信息，请参阅 [ASP.NET Core - Existing Database](#)([ASP.NET Core - 现有数据库](#))。

添加数据模型类

请注意: ASP.NET Core 2.0 模板包含 Models 文件夹。

右键单击 Models 文件夹，然后单击“添加”>“类”。将类命名为“Movie”，并添加以下属性：

```
using System;

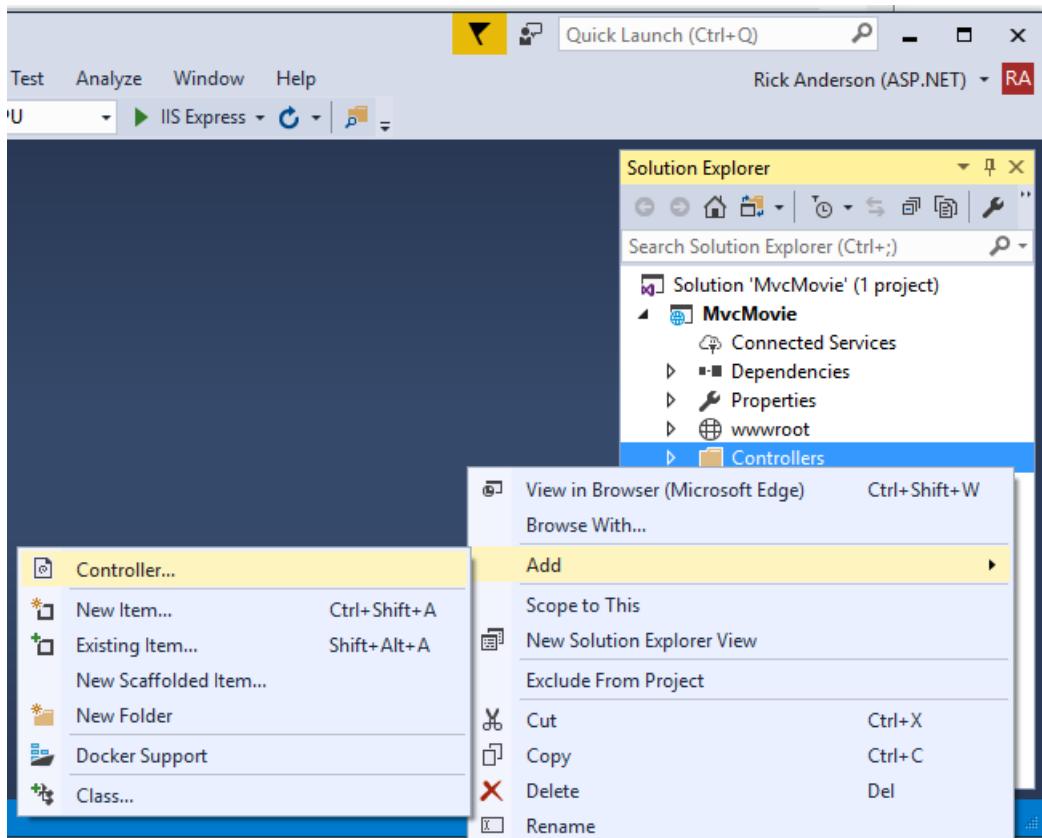
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段以获取主键。

生成项目以验证有没有任何错误存在。现在 MVC 应用中已具有模型。

搭建控制器基架

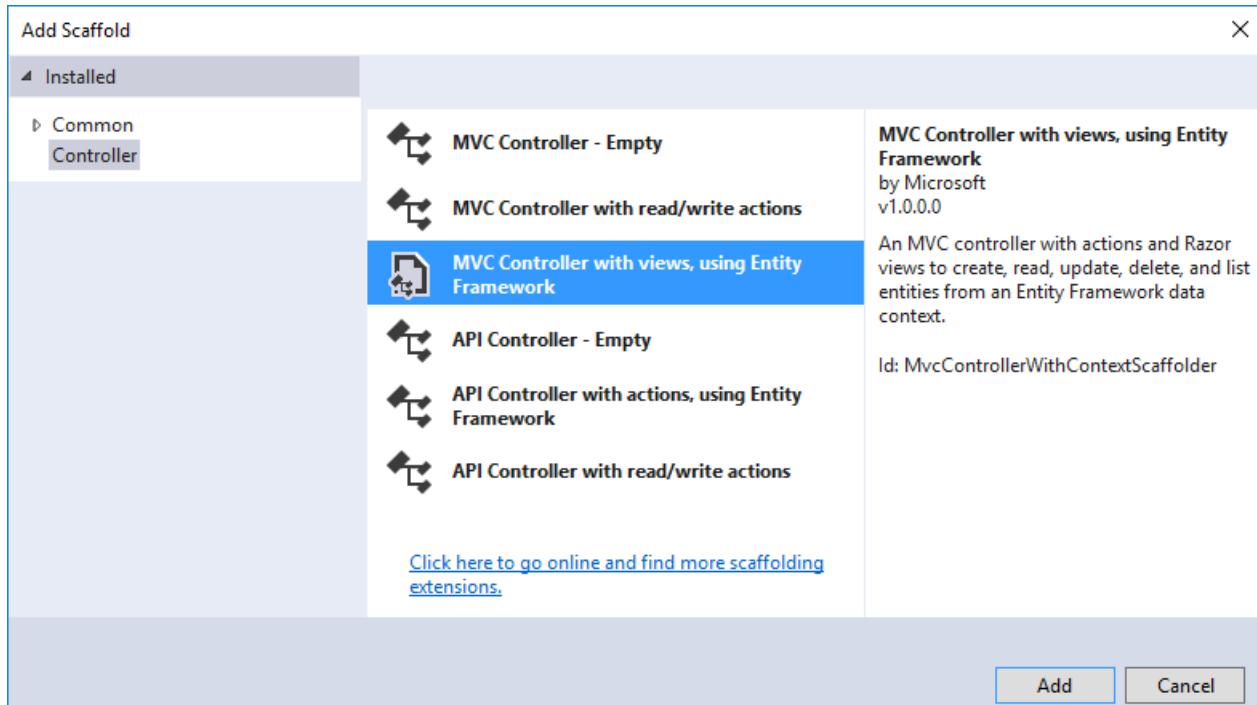
在“解决方案资源管理器”中，右键单击“控制器”文件夹，然后单击“添加”>“控制器”。



如果出现“添加 MVC 依赖项”对话框：

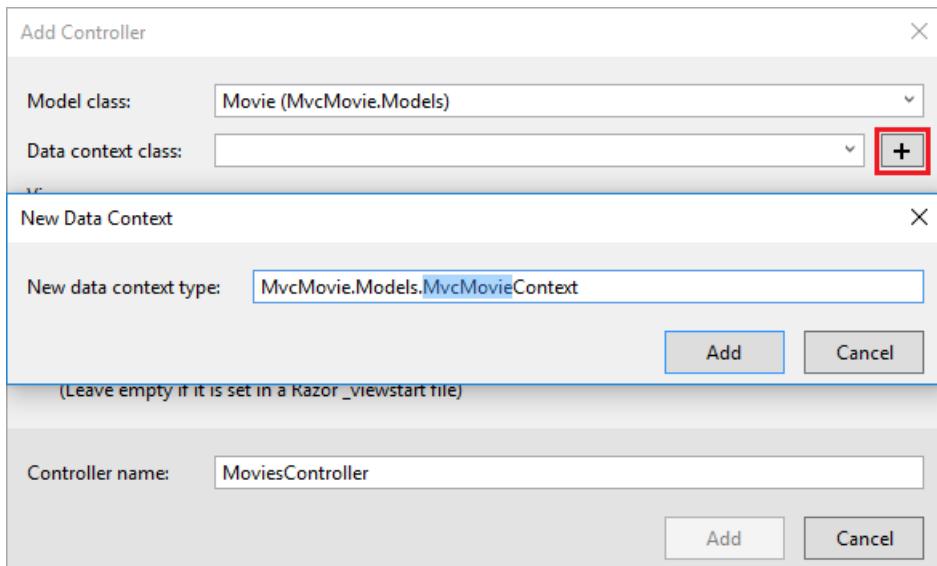
- 将 Visual Studio 更新到最新版本。15.5 之前的 Visual Studio 版本显示此对话框。
- 如果无法更新，请选择“添加”，然后再次按照添加控制器步骤操作。

在“添加基架”对话框中，点击“包含视图的 MVC 控制器(使用 Entity Framework)”>“添加”。

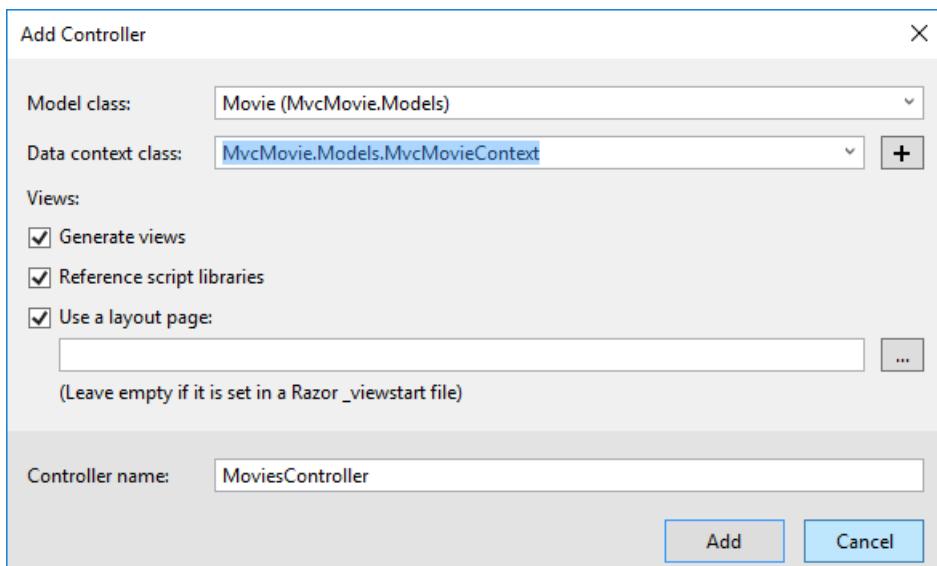


填写“添加控制器”对话框：

- 模型类: Movie(MvcMovie.Models)
- 数据上下文类: 选择 + 图标并添加默认的 MvcMovie.Models.MvcMovieContext



- 视图: 将每个选项保持为默认选中状态
- 控制器名称: 保留默认的 MoviesController
- 点击“添加”



Visual Studio 将创建:

- Entity Framework Core 数据库上下文类 (Data/MvcMovieContext.cs)
- 电影控制器 (Controllers/MoviesController.cs)
- “创建”、“删除”、“详细信息”、“编辑”和“索引”页面的 Razor 视图文件 (Views/Movies/*.cshtml)

自动创建数据库上下文和 **CRUD** (创建、读取、更新和删除) 操作方法和视图的过程称为“搭建基架”。你很快将具有功能完整的 Web 应用程序，可使用此应用程序管理电影数据库。

如果运行应用并单击“Mvc 电影”链接，则会出现以下类似的错误：



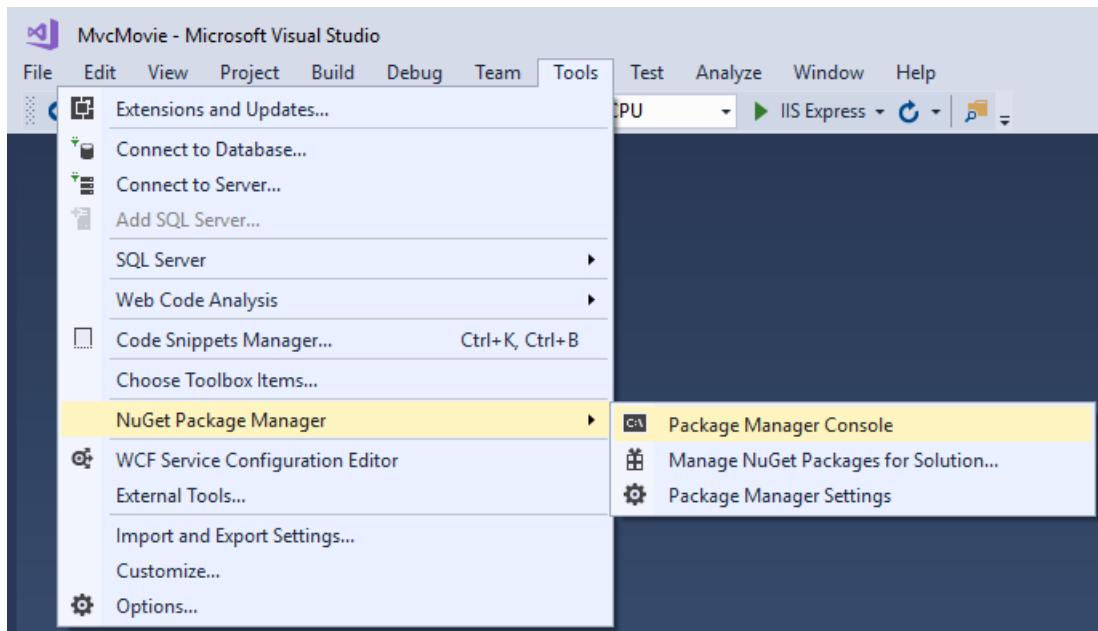
你需要创建数据库，并且使用 EF Core **迁移** 功能来执行此操作。通过迁移可创建与数据模型匹配的数据库，并在数据模型更改时更新数据库架构。

添加 EF 工具并执行初始迁移

在本部分中，将使用包管理器控制台 (PMC) 执行以下操作：

- 添加 Entity Framework Core 工具包。添加迁移并更新数据库需要此工具包。
- 添加初始迁移。
- 使用初始迁移来更新数据库。

从“工具”菜单中，选择“NuGet 包管理器”>“程序包管理器控制台”。



在 PMC 中，输入以下命令：

```
Install-Package Microsoft.EntityFrameworkCore.Tools  
Add-Migration Initial  
Update-Database
```

请注意：如果在使用 `Install-Package` 命令时收到错误，请打开 NuGet 包管理器并搜索 `Microsoft.EntityFrameworkCore.Tools` 包。通过此操作，可以安装包或检查是否已安装包。或者，如果 PMC 存在任何问题，请参阅 [CLI 方法](#)。

`Add-Migration` 命令创建代码以创建初始数据库架构。此架构基于 (Data/MvcMovieContext.cs 文件中的 `DbContext` 中指定的模型。`Initial` 参数用于为迁移命名。可以使用任意名称，但是按照惯例应选择描述迁移的名称。有关详细信息，请参阅 [迁移简介](#)。

`Update-Database` 命令在用于创建数据库的 Migrations/<time-stamp>_Initial.cs 文件中运行 `Up` 方法。

还可以使用命令行接口 (CLI) 来执行前面的步骤，而不使用 PMC：

- 将 [EF Core 工具](#) 添加到 .csproj 文件。
- 从控制台 (在项目目录中) 运行以下命令：

```
dotnet ef migrations add Initial  
dotnet ef database update
```

如果运行应用并收到错误消息：

```
SqlException: Cannot open database "Movie" requested by the login.  
The login failed.  
Login failed for user 'user name'.
```

可能尚未运行 `dotnet ef database update`。

测试应用

- 运行应用并点击“Mvc Movie”链接。
- 点击“新建”链接，创建电影。

The screenshot shows a browser window titled 'Create - Movie App'. The address bar displays 'localhost:1234/Movies/'. The main content area is titled 'MvcMovie' and contains a 'Create' form for a 'Movie'. The form fields are: 'Title' (input value 'Conan'), 'Release Date' (input value '3/3/2017'), 'Genre' (input value 'Comedy'), and 'Price' (input value '1.99'). Below the form is a 'Create' button and a 'Back to List' link. At the bottom, there is a copyright notice: '© 2017 - MvcMovie'.

- 可能无法在 `Price` 字段中输入小数点或逗号。若要使 jQuery 验证支持使用逗号（","）表示小数点及使用非美国英语日期格式的非英语区域设置，必须执行使应用全球化的步骤。请参阅 <https://github.com/aspnet/Docs/issues/4076> 和[其他资源](#)了解详细信息。目前，只能输入整数，例如 10。
- 在一些区域设置中，需要指定日期格式。请参阅下方突出显示的代码。

```

using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}

```

我们将在本教程的后续部分中探讨 `DataAnnotations`。

点击“创建”后，窗体会发布到服务器，其中电影信息会保存在数据库中。应用重定向到 /Movies URL，其中会显示新创建的电影信息。

Title	Release Date	Genre	Price
Conan	3/3/2017	Comedy	\$1.99

再创建几个其他的电影条目。试用“编辑”、“详细信息”和“删除”链接，它们均可正常工作。

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}

```

上面突出显示的代码显示了要添加到 [依赖关系注入](#) 容器的电影数据库上下文(在 Startup.cs 文件中)。

`services.AddDbContext<MvcMovieContext>(options =>` 指定要使用的数据库和连接字符串。`=>` 是 [Lambda 运算符](#)。

打开 Controllers/MoviesController.cs 文件并检查构造函数：

```

public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
}

```

构造函数使用[依赖关系注入](#)将数据库上下文 (`MvcMovieContext`) 注入到控制器中。数据库上下文将在控制器中的每个 [CRUD](#) 方法中使用。

强类型模型和 `@model` 关键词

在本教程之前的内容中，已经介绍了控制器如何使用 `ViewData` 字典将数据或对象传递给视图。`ViewData` 字典是一个动态对象，提供了将信息传递给视图的方便的后期绑定方法。

MVC 还提供将强类型模型对象传递给视图的功能。凭借此强类型方法可更好地对代码进行编译时检查。基架机制在创建方法和视图时，通过 `MoviesController` 类和视图使用了此方法（即传递强类型模型）。

检查 `Controllers/MoviesController.cs` 文件中生成的 `Details` 方法：

```

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

```

`id` 参数通常作为路由数据传递。例如 <http://localhost:5000/movies/details/1> 的设置如下：

- 控制器被设置为 `movies` 控制器（第一个 URL 段）。
- 操作被设置为 `details`（第二个 URL 段）。
- ID 被设置为 1（最后一个 URL 段）。

还可以使用查询字符串传入 `id`，如下所示：

```
http://localhost:1234/movies/details?id=1
```

在未提供 ID 值的情况下，`id` 参数可定义为[可以为 null 的类型](#) (`int?`)。

[Lambda 表达式](#)会被传入 `SingleOrDefaultAsync` 以选择与路由数据或查询字符串值相匹配的电影实体。

```

var movie = await _context.Movie
    .SingleOrDefaultAsync(m => m.ID == id);

```

如果找到了电影，`Movie` 模型的实例则会被传递到 `Details` 视图：

```
return View(movie);
```

检查 Views/Movies/Details.cshtml 文件的内容：

```
@model MvcMovie.Models.Movie

#{@
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.ID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

通过将 `@model` 语句包括在视图文件的顶端，可以指定视图期望的对象类型。创建电影控制器时，Visual Studio 会自动在 Details.cshtml 文件的顶端包括以下 `@model` 语句：

```
@model MvcMovie.Models.Movie
```

此 `@model` 指令使你能够使用强类型的 `Model` 对象访问控制器传递给视图的电影。例如，在 Details.cshtml 视图中，代码通过强类型的 `Model` 对象将每个电影字段传递给 `DisplayNameFor` 和 `DisplayFor` HTML 帮助程序。`Create` 和 `Edit` 方法以及视图也传递一个 `Movie` 模型对象。

检查电影控制器中的 Index.cshtml 视图和 `Index` 方法。请注意代码在调用 `view` 方法时是如何创建 `List` 对象的。代码将此 `Movies` 列表从 `Index` 操作方法传递给视图：

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

创建电影控制器时，基架会自动在 Index.cshtml 文件的顶端包含以下 `@model` 语句：

```
@model IEnumerable<MvcMovie.Models.Movie>
```

`@model` 指令使你能够使用强类型的 `Model` 对象访问控制器传递给视图的电影列表。例如，在 Index.cshtml 视图中，代码使用 `foreach` 语句通过强类型 `Model` 对象对电影进行循环遍历：

```

@model IEnumerable<MvcMovie.Models.Movie>

 @{
     ViewData["Title"] = "Index";
 }

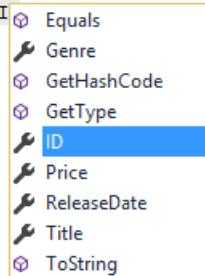
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>

```

因为 `Model` 对象为强类型(作为 `IEnumerable<Movie>` 对象), 因此循环中的每个项都被类型化为 `Movie`。除其他优点之外, 这意味着可对代码进行编译时检查:

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</table>
```



其他资源

- 标记帮助程序
- 全球化和本地化

[上一篇 : 添加视图](#)

[下一篇 : 使用 SQL](#)

在 ASP.NET Core 中使用 SQL Server LocalDB

2018/5/17 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

`MvcMovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

ASP.NET Core 配置系统会读取 `ConnectionString`。为了进行本地开发, 它会从 `appsettings.json` 文件获取连接字符串:

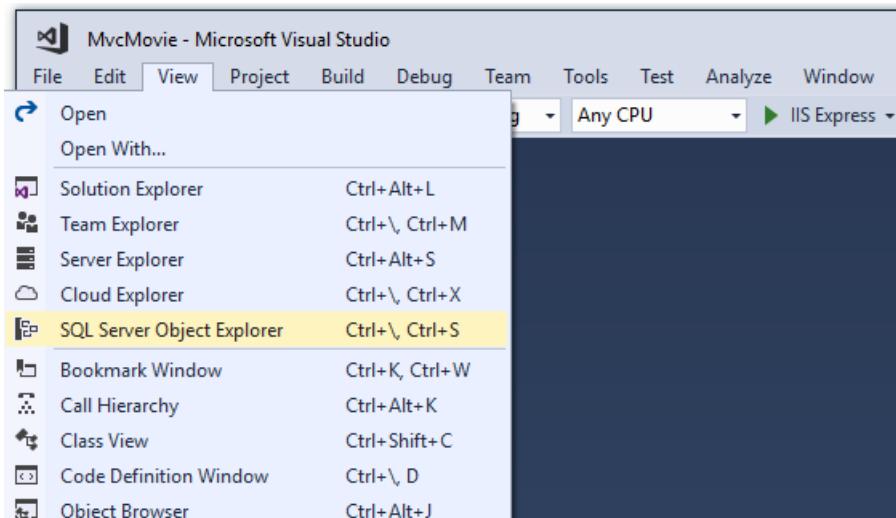
```
"ConnectionStrings": {
    "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

将应用部署到测试或生产服务器时, 可使用环境变量或另一种方法将连接字符串设置为实际的 SQL Server。有关详细信息, 请参阅[配置](#)。

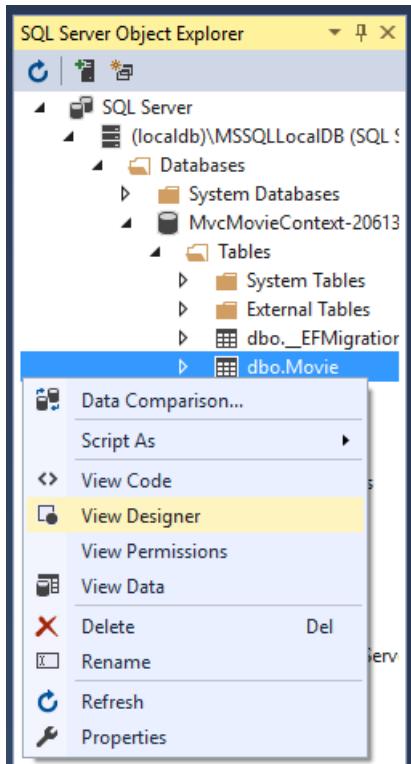
SQL Server Express LocalDB

LocalDB 是轻型版的 SQL Server Express 数据库引擎, 以程序开发为目标。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下, LocalDB 数据库在 `C:/Users/<user>` 目录中创建`*.mdf`文件。

- 从“视图”菜单中, 打开“SQL Server 对象资源管理器”(SSOX)。



- 右键单击 `Movie` 表, 然后单击“视图设计器”



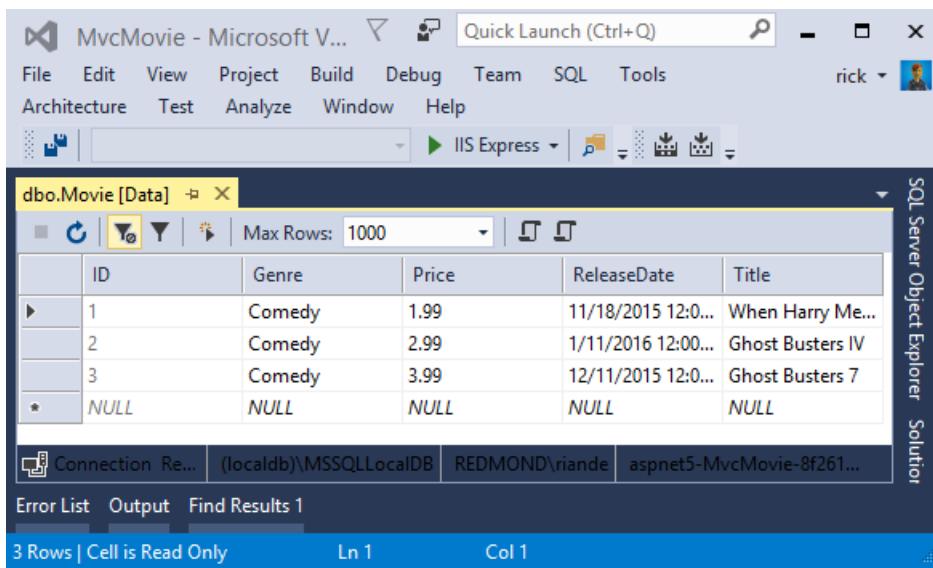
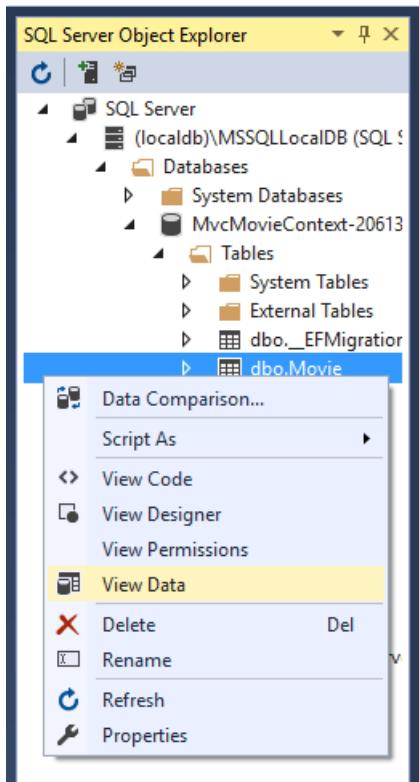
The screenshot shows the 'dbo.Movie [Design]' window. On the left is a grid view of the table columns: 'Name' (ID), 'Data Type' (int), 'Allow Nulls' (unchecked), 'Genre' (nvarchar(MAX)), 'Price' (decimal(18,2)), 'ReleaseDate' (datetime2(7)), and 'Title' (nvarchar(MAX)). To the right of the grid are sections for 'Keys' (1), 'Check Constraints' (0), 'Indexes' (0), 'Foreign Keys' (0), and 'Triggers' (0). Below the grid is a tab bar with 'Design' (selected) and 'T-SQL'. The 'T-SQL' tab displays the following CREATE TABLE statement:

```
1 CREATE TABLE [dbo].[Movie] (
2     [ID] INT IDENTITY (1, 1) NOT NULL,
3     [Genre] NVARCHAR (MAX) NULL,
4     [Price] DECIMAL (18, 2) NOT NULL,
5     [ReleaseDate] DATETIME2 (7) NOT NULL,
6     [Title] NVARCHAR (MAX) NULL,
7     CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
8 );
```

At the bottom of the window, there are status bars for 'Connection Ready', '(localdb)\MSSQLLocalDB', 'REDMOND\riande', and 'aspnet5-MvcMovie-8f261...'. The overall title bar says 'dbo.Movie [Design]'

请注意 `ID` 旁边的密钥图标。默认情况下，EF 将名为 `ID` 的属性设置为主键。

- 右键单击 `Movie` 表，然后单击“查看数据”



设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-1-11"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}

```

如果 DB 中没有任何电影，则会返回种子初始值设定项，并且不会添加任何电影。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

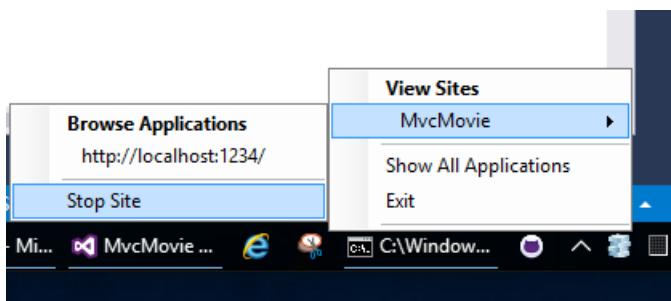
                try
                {
                    // Requires using MvcMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

- 删除 DB 中的所有记录。可以使用浏览器中的删除链接，也可从 SSOX 执行此操作。
- 强制应用初始化（调用 `Startup` 类中的方法），使种子方法能够正常运行。若要强制进行初始化，必须先停止 IIS Express，然后再重新启动它。可以使用以下任一方法来执行此操作：
 - 右键单击通知区域中的 IIS Express 系统任务栏图标，然后点击“退出”或“停止站点”



- 如果是在非调试模式下运行 VS 的, 请按 F5 以在调试模式下运行
- 如果是在调试模式下运行 VS 的, 请停止调试程序并按 F5

应用将显示设定为种子的数据。

The screenshot shows a Microsoft Edge browser window titled "Movie App". The address bar displays "localhost:5000/Movies". The main content area is titled "MvcMovie" and shows a table of movies:

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

At the bottom of the page, there is a copyright notice: "© 2017 - MvcMovie".

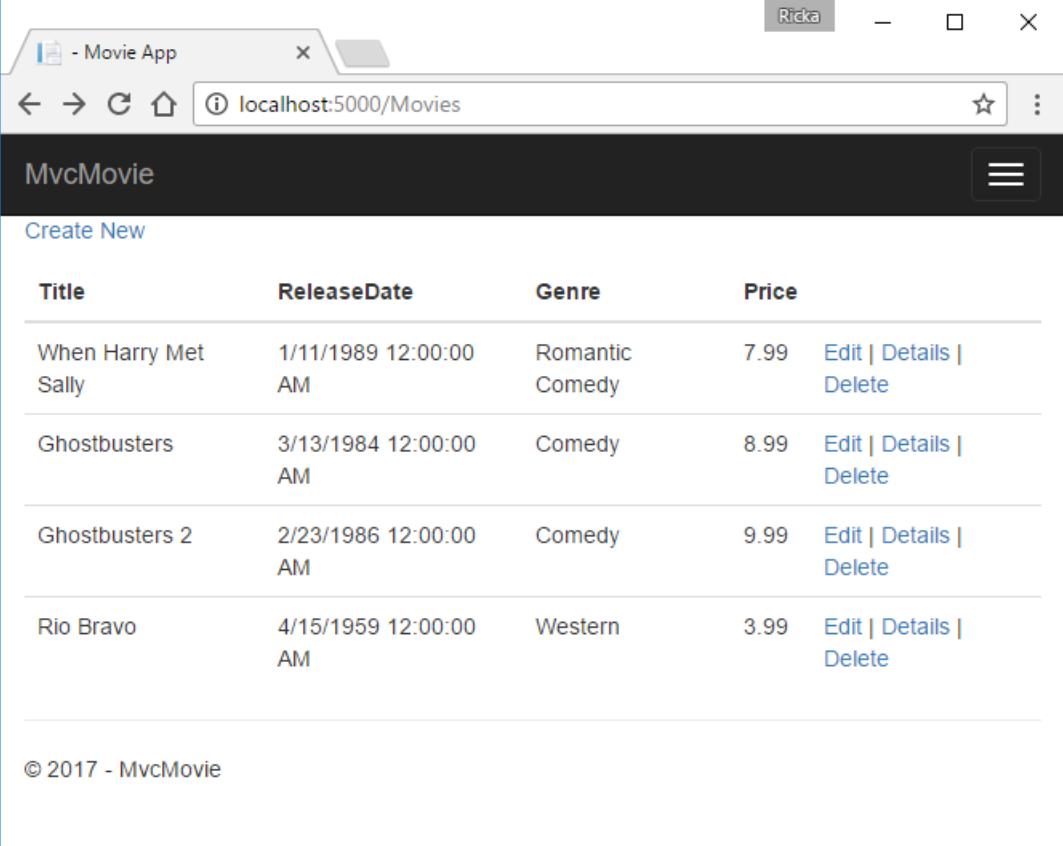
[上一页](#) [下一页](#)

ASP.NET Core 中的控制器方法和视图

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(如下图所示的 12:00:00 AM), 并且“ReleaseDate”应为两个词。



The screenshot shows a Microsoft Edge browser window titled "Movie App". The address bar displays "localhost:5000/Movies". The main content area is titled "MvcMovie" and contains a table with four columns: "Title", "ReleaseDate", "Genre", and "Price". The table lists four movies:

Title	ReleaseDate	Genre	Price
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99

Each row has "Edit | Details | Delete" links in the last column. At the bottom of the page, there is a copyright notice: "© 2017 - MvcMovie".

打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

右键单击红色的波浪线, 然后选择“快速操作和重构”。

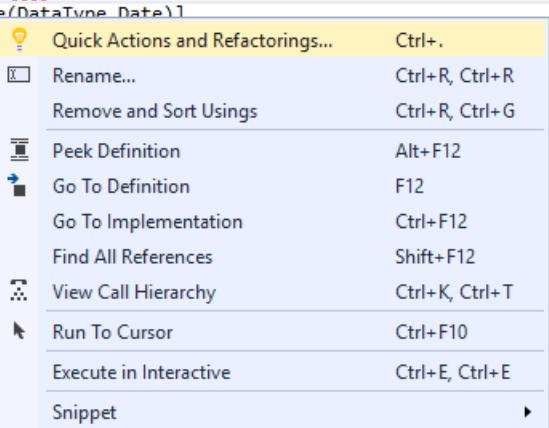
```

using System;

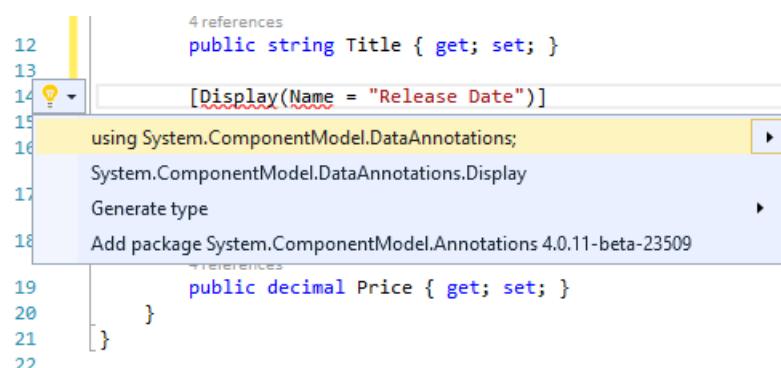
namespace MvcMovie.Models
{
    8 references | 0 changes | 0 authors, 0 changes
    public class Movie
    {
        7 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public int ID { get; set; }
        4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]

```



点击 `using System.ComponentModel.DataAnnotations;`



Visual studio 添加 `using System.ComponentModel.DataAnnotations;`。

删除不需要的 `using` 语句。默认情况下，它们显示为浅灰色字体。右键单击 Movie.cs 文件中的任意位置，然后选择“对 Using 进行删除和排序”。

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

Quick Actions and Refactorings... Ctrl+.

Rename... Ctrl+R, Ctrl+R

Remove and Sort Usings Ctrl+R, Ctrl+G

Peek Definition Alt+F12

Go To Definition F12

Go To Implementation Ctrl+F12

Find All References Shift+F12

View Call Hierarchy Ctrl+K, Ctrl+T

Run To Cursor Ctrl+F10

Execute in Interactive Ctrl+E, Ctrl+E

Snippet ▾

Cut Ctrl+X

Copy Ctrl+C

Paste Ctrl+V

Outlining ▾

Source Control ▾

更新的代码：

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

我们将在下一教程中介绍 [DataAnnotations](#)。`Display` 特性指定要显示的字段名称的内容(本例中应为“Release Date”，而不是“ReleaseDate”)。`DataType` 属性指定数据的类型(日期)，使字段中存储的时间信息不会显示。

浏览到 [Movies](#) 控制器，并将鼠标指针悬停在“编辑”链接上以查看目标 URL。

Index

Create New

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

© 2016 - MvcMovie

<http://localhost:1234/Movies/Edit/5>

“编辑”、“详细信息”和“删除”链接是在 Views/Movies/Index.cshtml 文件中由 Core MVC 定位标记帮助程序生成的。

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
<a asp-action="Details" asp-route-id="@item.ID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
```

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在上面的代码中，`AnchorTagHelper` 从控制器操作方法和路由 ID 动态生成 HTML `href` 特性值。在最喜欢的浏览器中使用“查看源”，或使用开发人员工具来检查生成的标记。生成的 HTML 的一部分如下所示：

```
<td>
<a href="/Movies/Edit/4"> Edit </a> |
<a href="/Movies/Details/4"> Details </a> |
<a href="/Movies/Delete/4"> Delete </a>
</td>
```

重新调用在 Startup.cs 文件中设置的路由的格式：

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

ASP.NET Core 将 `http://localhost:1234/Movies/Edit/4` 转换为对 `Movies` 控制器的 `Edit` 操作方法的请求，参数 `Id` 为 4。（控制器方法也称为操作方法。）

标记帮助程序是 ASP.NET Core 中最受欢迎的新功能之一。有关详细信息，请参阅[其他资源](#)。

打开 `Movies` 控制器并检查两个 `Edit` 操作方法。以下代码显示了 `HTTP GET Edit` 方法，此方法将提取电影并填

充由 Edit.cshtml Razor 文件生成的编辑表单。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

以下代码显示 `HTTP POST Edit` 方法，它会处理已发布的电影值：

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`[Bind]` 特性是防止过度发布的一种方法。只应在 `[Bind]` 特性中包含想要更改的属性。有关详细信息，请参阅 [Protect your controller from over-posting](#)(防止控制器过度发布)。[ViewModels](#) 提供了一种替代方法以防止过度发布。

请注意第二个 `Edit` 操作方法的前面是 `[HttpPost]` 特性。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

`HttpPost` 特性指定只能为 `POST` 请求调用此 `Edit` 方法。可将 `[HttpGet]` 属性应用于第一个编辑方法，但不是必需，因为 `[HttpGet]` 是默认设置。

`ValidateAntiForgeryToken` 特性用于[防止请求伪造](#)，并与编辑视图文件 (`Views/Movies/Edit.cshtml`) 中生成的防伪标记相配对。编辑视图文件使用[表单标记帮助程序](#)生成防伪标记。

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

[表单标记帮助程序](#)会生成隐藏的防伪标记，此标记必须与电影控制器的 `Edit` 方法中 `[ValidateAntiForgeryToken]` 生成的防伪标记相匹配。有关详细信息，请参阅[反请求伪造](#)。

`HttpGet` `Edit` 方法采用电影 `ID` 参数，使用 Entity Framework `SingleOrDefaultAsync` 方法查找电影，并将所选电影返回到“编辑”视图。如果无法找到电影，则返回 `NotFound` (HTTP 404)。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

当基架系统创建“编辑”视图时，它会检查 `Movie` 类并创建代码为类的每个属性呈现 `<label>` 和 `<input>` 元素。

以下示例显示由 Visual Studio 基架系统生成的“编辑”视图：

```

@model MvcMovie.Models.Movie

 @{
     ViewData["Title"] = "Edit";
 }

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Price" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

请注意视图模板在文件顶端有一个 `@model MvcMovie.Models.Movie` 语句。`@model MvcMovie.Models.Movie` 指定视图期望的视图模板的模型为 `Movie` 类型。

基架的代码使用几个标记帮助程序方法来简化 HTML 标记。[标签标记帮助程序](#) 显示字段的名称 ("Title"、"ReleaseDate"、"Genre" 或 "Price")。[输入标记帮助程序](#) 呈现 HTML `<input>` 元素。[验证标记帮助程序](#) 显示与该属性相关联的任何验证消息。

运行应用程序并导航到 `/Movies` URL。点击“编辑”链接。在浏览器中查看页面的源。为 `<form>` 元素生成的 HTML 如下所示。

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCILsdCRx9jDQC1rV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-MF_9rK8aAZdRd1Ori7FmKVkRe_2v5LIHGKFctjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

`<input>` 元素位于 `HTML <form>` 元素中，后者的 `action` 特性设置为发布到 `/Movies/Edit/id` URL。当单击 `Save` 按钮时，表单数据将发布到服务器。关闭 `</form>` 元素之前的最后一行显示 [表单标记帮助程序](#) 生成的隐藏的 `XSRF` 标记。

处理 POST 请求

以下列表显示了 `Edit` 操作方法的 `[HttpPost]` 版本。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

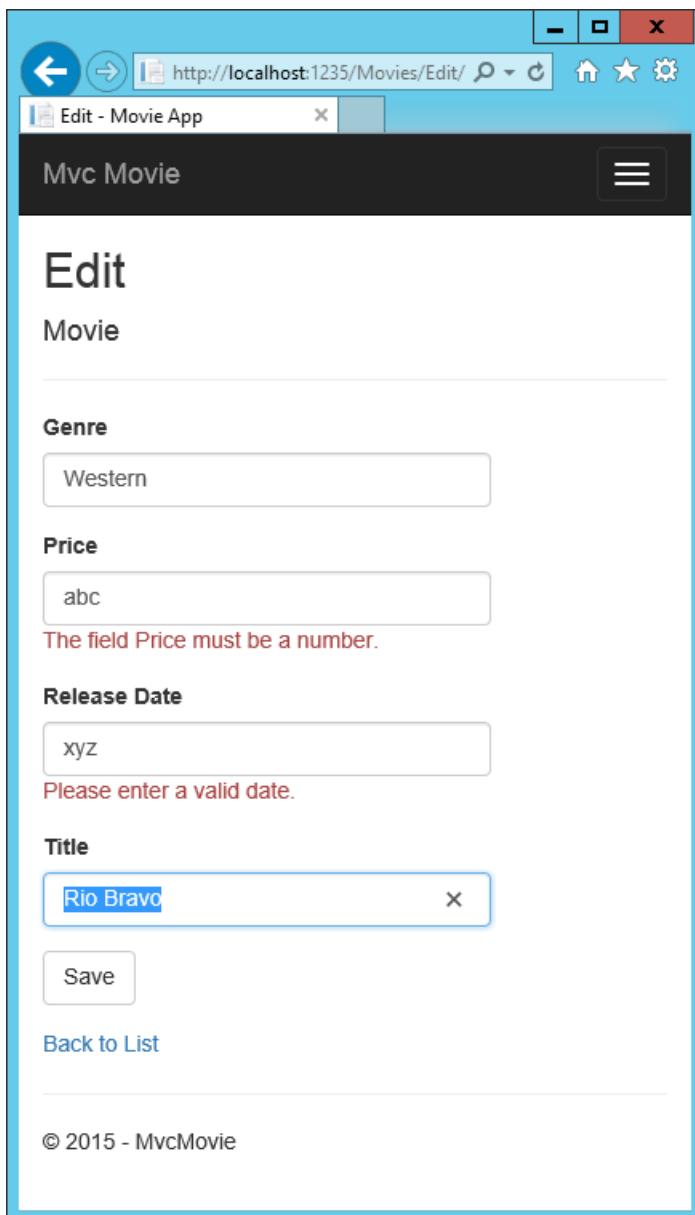
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

[ValidateAntiForgeryToken] 特性验证表单标记帮助程序中的防伪标记生成器生成的隐藏的 **XSRF** 标记

模型绑定 系统采用发布的表单值，并创建一个作为 `movie` 参数传递的 `Movie` 对象。`ModelState.IsValid` 方法验证表单中提交的数据是否可以用于修改(编辑或更新) `Movie` 对象。如果数据有效，将保存此数据。通过调用数据库上下文的 `SaveChangesAsync` 方法，将更新(编辑)的电影数据保存到数据库。保存数据后，代码将用户重定向到 `MoviesController` 类的 `Index` 操作方法，此方法显示电影集合，包括刚才所做的更改。

在表单发布到服务器之前，客户端验证会检查字段上的任何验证规则。如果有任何验证错误，则将显示错误消息，并且不会发布表单。如果禁用 JavaScript，则不会进行客户端验证，但服务器将检测无效的发布值，并且表单值将与错误消息一起重新显示。稍后在本教程中，我们将更详细地研究**模型验证**。Views/Movies/Edit.cshtml 视图模板中的**验证标记帮助程序**负责显示相应的错误消息。



电影控制器中的所有 `HttpGet` 方法都遵循类似的模式。它们获取电影对象(对于 `Index` 获取的是对象列表)并将对象(模型)传递给视图。`Create` 方法将空的电影对象传递给 `Create` 视图。在方法的 `[HttpPost]` 重载中, 创建、编辑、删除或其他方式修改数据的所有方法都执行此操作。以 `HTTP GET` 方式修改数据是一种安全隐患。以 `HTTP GET` 方法修改数据也违反了 HTTP 最佳做法和架构 REST 模式, 后者指定 GET 请求不应更改应用程序的状态。换句话说, 执行 GET 操作应是没有任何隐患的安全操作, 也不会修改持久数据。

其他资源

- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)
- [防请求伪造](#)
- [防止控制器过度发布](#)
- [ViewModels](#)
- [表单标记帮助程序](#)
- [输入标记帮助程序](#)
- [标签标记帮助程序](#)
- [选择标记帮助程序](#)
- [验证标记帮助程序](#)

[上一页](#)

[下一页](#)

将搜索添加到 ASP.NET Core MVC 应用

2018/5/8 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Index` 操作方法添加搜索功能, 以实现按“类型”或“名称”搜索电影。

使用以下代码更新 `Index` 方法:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

`Index` 操作方法的第一行创建了 `LINQ` 查询用于选择电影:

```
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义, 它还不会针对数据库运行。

如果 `searchString` 参数包含一个字符串, 电影查询则会被修改为根据搜索字符串的值进行筛选:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

上面的 `s => s.Title.Contains()` 代码是 `Lambda 表达式`。Lambda 在基于方法的 `LINQ` 查询中用作标准查询运算符方法的参数, 如 `Where` 方法或 `Contains` (上述的代码中所使用的)。在对 `LINQ` 查询进行定义或通过调用方法 (如 `Where`、`Contains` 或 `OrderBy`) 进行修改后, 此查询不会被执行。相反, 会延迟执行查询。这意味着表达式的计算会延迟, 直到真正循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关延迟执行查询的详细信息, 请参阅 [Query Execution](#) (查询执行)。

注意: `Contains` 方法在数据库上运行, 而不是在上面显示的 c# 代码中运行。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上, `Contains` 映射到 `SQL LIKE`, 这是不区分大小写的。在 SQLite 中, 由于使用了默认排序规则, 因此需要区分大小写。

导航到 `/Movies/Index`。将查询字符串 (如 `?searchString=Ghost`) 追加到 URL。筛选的电影将显示出来。

如果将 `Index` 方法的签名更改为具有名称为 `id` 的参数，则 `id` 参数将匹配 `Startup.cs` 中设置的默认路由的可选 `{id}` 占位符。

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

可使用“重命名”命令快速将 `searchString` 参数重命名为 `id`。右键单击“`searchString`”，选择“重命名”。

突出显示重命名目标。

```
public ActionResult Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(movies);
}
```

将参数更改为 `id`，并将出现的所有 `searchString` 更改为 `id`。

```
public ActionResult Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(movies);
}
```

之前的 `Index` 方法：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

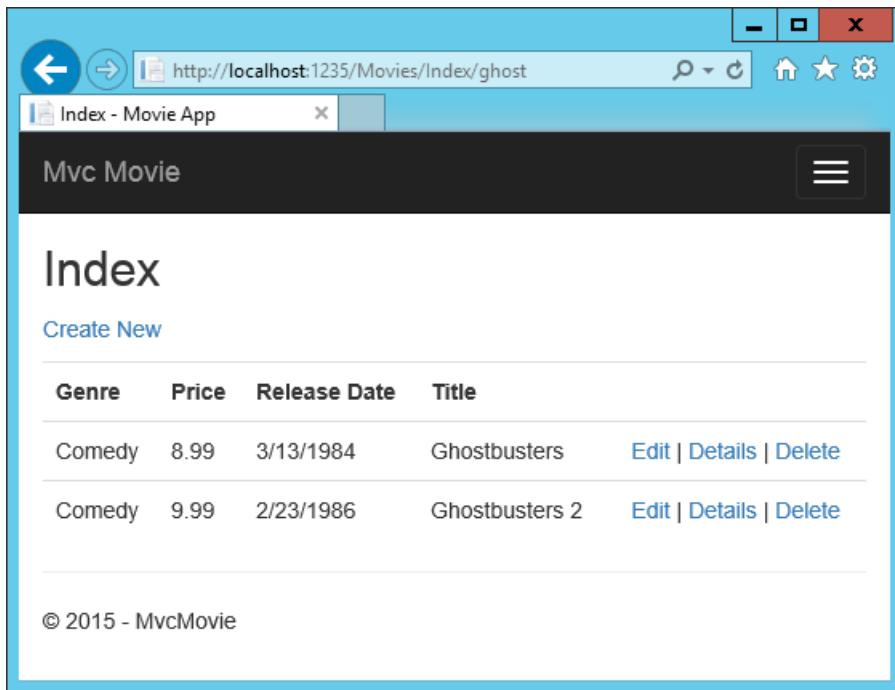
更新后带 `id` 参数的 `Index` 方法：

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

现可将搜索标题作为路由数据 (URL 段) 而非查询字符串值进行传递。



但是，不能指望用户在每次要搜索电影时都修改 URL。因此需要添加 UI 元素来帮助他们筛选电影。若已更改 `Index` 方法的签名，以测试如何传递绑定路由的 `ID` 参数，请改回原样，使其采用名为 `searchString` 的参数：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

打开“Views/Movies/Index.cshtml”文件，并添加以下突出显示的 `<form>` 标记：

```
ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

此 HTML `<form>` 标记使用表单标记帮助程序，因此提交表单时，筛选器字符串会发布到电影控制器的 `Index` 操作。保存更改，然后测试筛选器。

Index - Movie App

localhost:1899/Movies

MvcMovie

Create New

Title: ghost Filter

Genre	Price	Release Date	Title
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally
Comedy	8.99	3/13/1984	Ghostbusters
Comedy	9.99	2/23/1986	Ghostbusters 2
Western	3.99	4/15/1959	Rio Bravo

© 2016 - MvcMovie

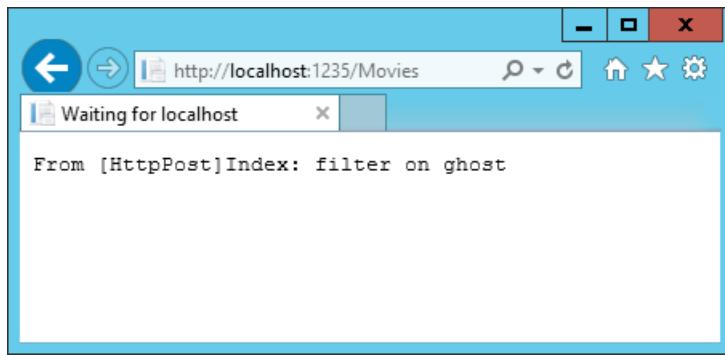
如你所料，不存在 `Index` 方法的 `[HttpPost]` 重载。无需重载，因为该方法不更改应用的状态，仅筛选数据。

可添加以下 `[HttpPost] Index` 方法。

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

`notUsed` 参数用于创建 `Index` 方法的重载。本教程稍后将对此进行探讨。

如果添加此方法，则操作调用程序将与 `[HttpPost] Index` 方法匹配，且将运行 `[HttpPost] Index` 方法，如下图所示。



但是，即使添加 `Index` 方法的 `[HttpPost]` 版本，其实现方式也受到限制。假设你想要将特定搜索加入书签，或向朋友发送一个链接，让他们单击链接即可查看筛选出的相同电影列表。请注意，HTTP POST 请求的 URL 与 GET 请求的 URL 相同 (`localhost:xxxxx/Movies/Index`)，其中不包含搜索信息。搜索字符串信息作为表单域值发送给服务器。可使用浏览器开发人员工具或出色的 `Fiddler` 工具对其进行验证。下图展示了 Chrome 浏览器开发人员工具：

The screenshot shows the Chrome DevTools Network tab with a red box highlighting the 'Network' tab itself. A second red box highlights the 'General' section of the request details, which displays the following information:

- Request URL: <http://localhost:5000/Movies>
- Request Method: POST
- Status Code: 200 OK
- Remote Address: [::1]:5000
- Referrer Policy: no-referrer-when-downgrade

A third red box highlights the 'Form Data' section, which shows the following form fields:

- SearchString: Ghost
- _RequestVerificationToken: CfDJ8B98MxUFL5pAq2aeCj59HP1g2HXMD176Mablw7uuk20AGreBb3y0NufBTMajxmJCjRFe-2sF50PVla72IyfCA9Pao3muZ0f4jtjDND1XEagdJk_g67wBX12qOKI7LD980GjmjBB_-5rvRhJuQCroPRw

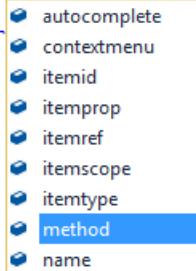
At the bottom of the Network tab, there is a message: "4 requests | 3.9 KB transferred | Fi...".

在请求正文 中，可看到搜索参数和 **XSRF** 标记。请注意，正如之前教程所述，[表单标记帮助程序](#) 会生成一个 **XSRF** 防伪标记。不会修改数据，因此无需验证控制器方法中的标记。

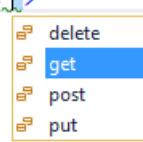
搜索参数位于请求正文而非 URL 中，因此无法捕获该搜索信息进行书签设定或与他人共享。将通过指定请求为 **HTTP GET** 进行修复。

请注意 intelliSense 如何帮助更新标记。

```
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```



```
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```



请注意 `<form>` 标记中独特的字体。该独特字体表明标记受标记帮助程序支持。

```
<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

现在提交搜索后，URL 将包含搜索查询字符串。即使具备 `HttpPost Index` 方法，搜索也将转到 `HttpGet Index` 操作方法。

Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete

以下标记显示对 `form` 标记的更改：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

添加“按流派搜索”

将以下 `MovieGenreViewModel` 类添加到“模型”文件夹：

```

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

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}

```

“电影流派”视图模型将包含：

- 电影列表。
- 包含流派列表的 `SelectList`。用户可通过它从列表中选择一种流派。
- 包含所选流派的 `movieGenre`。

将 `MoviesController.cs` 中的 `Index` 方法替换为以下代码：

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();

    return View(movieGenreVM);
}

```

以下代码是一种 `LINQ` 查询，可从数据库中检索所有流派。

```

// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

```

通过投影不同的流派创建 `SelectList`（我们不希望选择列表中的流派重复）。

```

movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())

```

向索引视图添加“按流派搜索”

按如下更新 `Index.cshtml`：

```

@model MvcMovie.Models.MovieGenreViewModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="movieGenre" asp-items="Model.genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.movies[0].Title)
```

在上述代码中，`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。由于只检查但未计算 Lambda 表达式，因此当 `model`、`model.movies[0]` 或 `model.movies` 为 `null` 或空时，你不会收到访问冲突。对 Lambda 表达式求值时（例如，`@Html.DisplayFor(modelItem => item.Title)`），将求得该模型的属性值。

通过按流派或/和电影标题搜索来测试应用。

上一页

下一页

将新字段添加到 ASP.NET Core 应用

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将使用 [Entity Framework](#) Code First 迁移将新字段添加到模型, 并将此更改迁移到数据库。

使用 EF Code First 自动创建数据库时, Code First 会向数据库添加表格, 以帮助跟踪数据库的架构是否与从其中生成它的模型类同步。如果它们不同步, EF 则会引发异常。这使查找不一致的数据库/代码问题变得更加轻松。

向电影模型添加分级属性

打开 Models/Movie.cs 文件, 并添加 `Rating` 属性:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

生成应用 (Ctrl+Shift+B)。

因为已经添加新字段到 `Movie` 类, 所以还需要更新绑定允许名单, 将此新属性纳入其中。在 MoviesController.cs 中, 更新 `Create` 和 `Edit` 操作方法的 `[Bind]` 属性, 以包括 `Rating` 属性:

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

还需要更新视图模板, 以便在浏览器视图中显示、创建和编辑新的 `Rating` 属性。

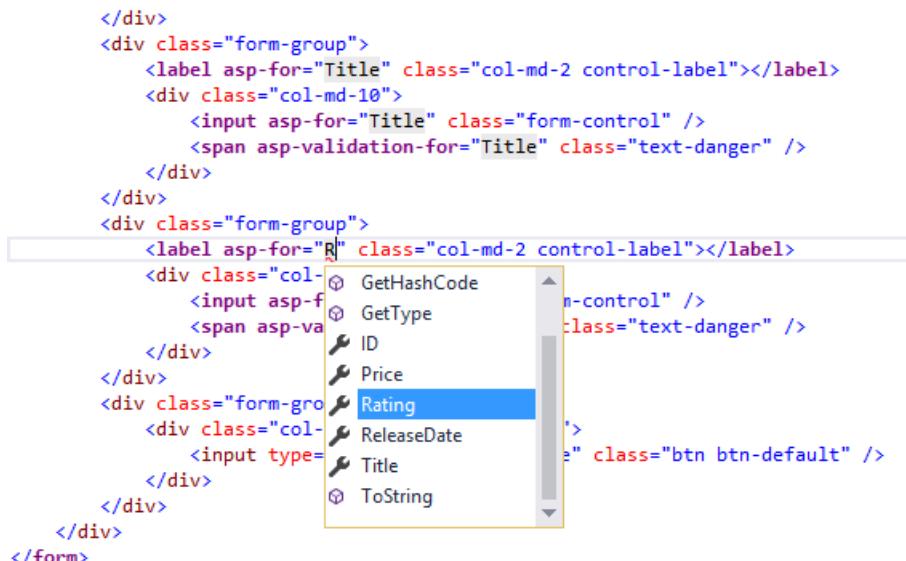
编辑 /Views/Movies/Index.cshtml 文件并添加 `Rating` 字段:

```


| @Html.DisplayNameFor(model => model.movies[0].Title) | @Html.DisplayNameFor(model => model.movies[0].ReleaseDate) | @Html.DisplayNameFor(model => model.movies[0].Genre) | @Html.DisplayNameFor(model => model.movies[0].Price) | @Html.DisplayNameFor(model => model.movies[0].Rating) |
|------------------------------------------------------|------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------|-------------------------------------------------------|
| @Html.DisplayFor(modelItem => item.Title)            | @Html.DisplayFor(modelItem => item.ReleaseDate)            | @Html.DisplayFor(modelItem => item.Genre)            | @Html.DisplayFor(modelItem => item.Price)            | @Html.DisplayFor(modelItem => item.Rating)            |


```

使用 `Rating` 字段更新 /Views/Movies/Create.cshtml。可以复制/粘贴之前的“窗体组”，并让 intelliSense 帮助更新字段。IntelliSense 适用于 [标记帮助程序](#)。请注意：在 Visual Studio 2017 的 RTM 版本中，需要安装 Razor intelliSense 的 [Razor 语言服务](#)。此问题将在下一版本中修复。



The screenshot shows a portion of the `Create.cshtml` file from the `Movies` view. A tooltip is displayed over the `Rating` field in the third column of a form group. The tooltip lists several properties and methods available for the `Rating` type, including `GetHashCode`, `GetType`, `ID`, `Price`, `Rating` (which is highlighted in blue), `ReleaseDate`, `Title`, and `ToString`.

```

</div>
&ltdiv class="form-group">
    <label asp-for="Title" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger" />
    </div>
</div>
&ltdiv class="form-group">
    <label asp-for="R" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="R" class="form-control" />
        <span asp-validation-for="R" class="text-danger" />
    </div>
</div>
&ltdiv class="form-group">
    <div class="col-md-2">
        <input type="button" value="Create" class="btn btn-default" />
    </div>
</div>
</div>
</form>

```

在将 DB 更新为包括新字段之前，应用不会正常工作。如果现在运行，会出现以下 `SqlException`：

```
SqlException: Invalid column name 'Rating'.
```

看到此错误是因为更新的 Movie 模型类与现有数据库的 Movie 表架构不同。（数据库表中没有“Rating”列。）

可通过几种方法解决此错误：

1. 让 Entity Framework 自动丢弃，并基于新的模型类架构重新创建数据库。在测试数据库上进行开发时，此方法在开发周期早期很方便；通过它可以一起快速改进模型和数据库架构。但其缺点是会丢失数据库中的现有数据 - 因此请勿对生产数据库使用此方法！使用初始值设定项，以使用测试数据自动设定数据库种子，这通常是开发应用程序的有效方式。
2. 对现有数据库架构进行显式修改，使它与模型类相匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。
3. 使用 Code First 迁移更新数据库架构。

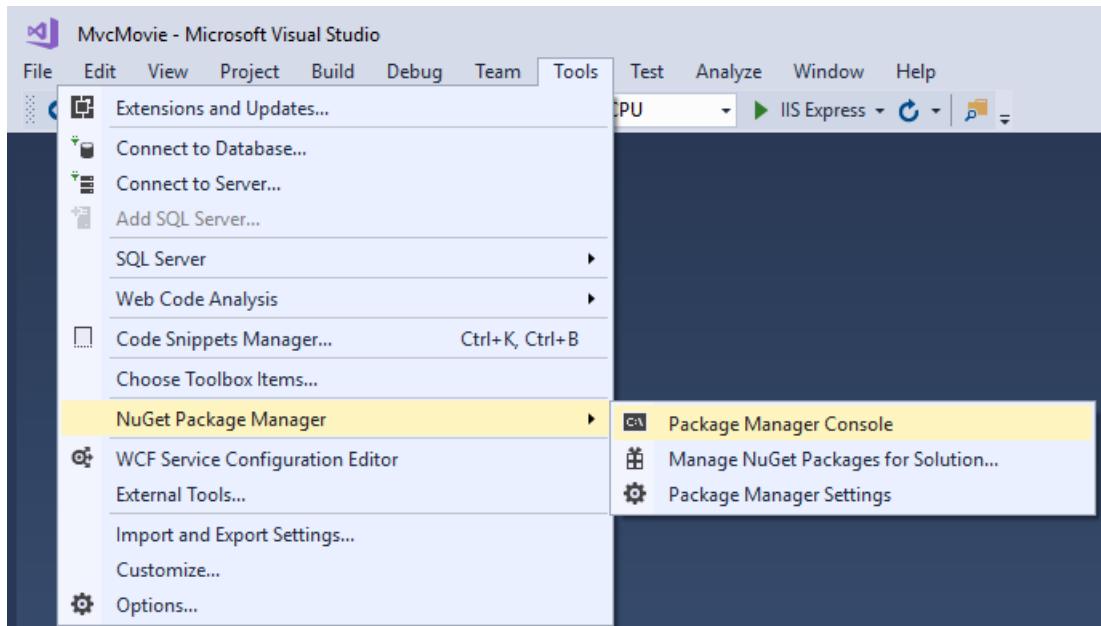
本教程使用 Code First 迁移。

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 做出此更改。

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

生成解决方案。

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。



在 PMC 中，输入以下命令：

```
Add-Migration Rating
Update-Database
```

`Add-Migration` 命令会通知迁移框架使用当前 `Movie` DB 架构检查当前 `Movie` 模型，并创建必要的代码，将 DB

迁移到新模型。名称“Rating”是任意的，用于对迁移文件进行命名。为迁移文件使用有意义的名称是有帮助的。

如果删除 DB 中的所有记录，初始化会设定 DB 种子，并将包括 `Rating` 字段。可以使用浏览器中的删除链接，也可从 SSOX 执行此操作。

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。还应向 `Edit`、`Details` 和 `Delete` 视图模板添加 `Rating` 字段。

[上一页](#) [下一页](#)

添加验证

2018/5/8 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Movie` 模型添加验证逻辑, 并确保每当用户创建或编辑电影时, 都会强制执行验证规则。

坚持 DRY 原则

MVC 的设计原则之一是 [DRY](#) (“不要自我重复”)。ASP.NET MVC 支持你仅指定一次功能或行为, 然后使它应用到整个应用中。这可以减少所需编写的代码量, 并使编写的代码更少出错, 更易于测试和维护。

MVC 和 Entity Framework Core Code First 提供的验证支持是 DRY 原则在实际操作中的极佳示例。可以在一个位置(模型类中)以声明方式指定验证规则, 并且在应用中的所有位置强制执行。

将验证规则添加到电影模型

打开 `Movie.cs` 文件。DataAnnotations 提供一组内置验证特性, 可通过声明方式应用于任何类或属性。(它还包含 `DataType` 等格式特性, 这些特性可帮助进行格式设置, 但不提供任何验证。)

更新 `Movie` 类以使用内置的 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'''\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性指定要对应用这些特性的模型属性强制执行的行为。`Required` 和 `MinimumLength` 特性表示属性必须有值; 但用户可输入空格来满足此验证。`RegularExpression` 特性用于限制可输入的字符。在上述代码中, `Genre` 和 `Rating` 仅可使用字母(禁用空格、数字和特殊字符)。`Range` 特性将值限制在指定范围内。`StringLength` 特性使你能够设置字符串属性的最大长度, 以及可选的最小长度。从本质上来说, 需要值类型(如 `decimal`、`int`、`float`、`DateTime`)，但不需要 `[Required]` 特性。

让 ASP.NET 强制自动执行验证规则有助于提升应用的可靠性。同时它能确保你无法忘记验证某些内容, 并防止你

无意中将错误数据导入数据库。

MVC 中的验证错误 UI

运行应用并导航到电影控制器。

点击“新建”连接添加新电影的链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

The screenshot shows a browser window with the title "Create - Movie App". The address bar displays "localhost:5000/Movies/Create". The main content area is titled "Create" and contains a "Movie" form. The form fields and their validation errors are:

- Title**: The field must be a string with a minimum length of 3 and a maximum length of 60.
- Release Date**: The field must be a date.
- Genre**: The Genre field is required.
- Price**: The field Price must be a number.
- Rating**: The field Rating must match the regular expression '^([A-Z]+[a-zA-Z-\\s]*\$)'.

At the bottom of the form are two buttons: "Create" and "Back to List". The footer of the page says "© 2017 - MvcMovie".

注意

可能无法在 `Price` 字段中输入十进制逗号。若要使 `jQuery` 验证支持使用逗号（“,”）表示小数点的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 [GitHub 问题 4076](#)。

请注意表单如何自动呈现每个包含无效值的字段中相应的验证错误消息。客户端（使用 JavaScript 和 jQuery）和服务器端（若用户禁用 JavaScript）都必定会遇到这些错误。

明显的好处在于不需要在 `MoviesController` 类或 `Create.cshtml` 视图中更改单个代码行来启用此验证 UI。在本教程前面创建的控制器和视图会自动选取验证规则，这些规则是通过在 `Movie` 模型类的属性上使用验证特性所指定的。使用 `Edit` 操作方法测试验证后，即已应用相同的验证。

存在客户端验证错误时，不会将表单数据发送到服务器。可通过使用 [Fiddler 工具](#)或 [F12 开发人员工具](#)在 `HTTP Post` 方法中设置断点来对此进行验证。

验证工作原理

你可能想知道在不对控制器或视图中的代码进行任何更新的情况下，验证 UI 是如何生成的。下列代码显示两种 `Create` 方法。

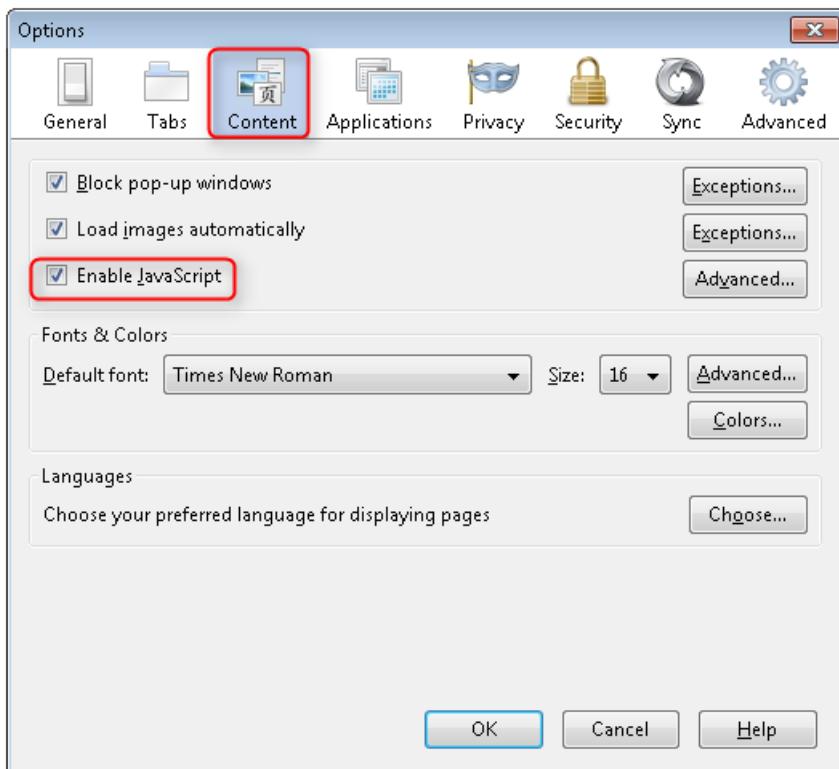
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

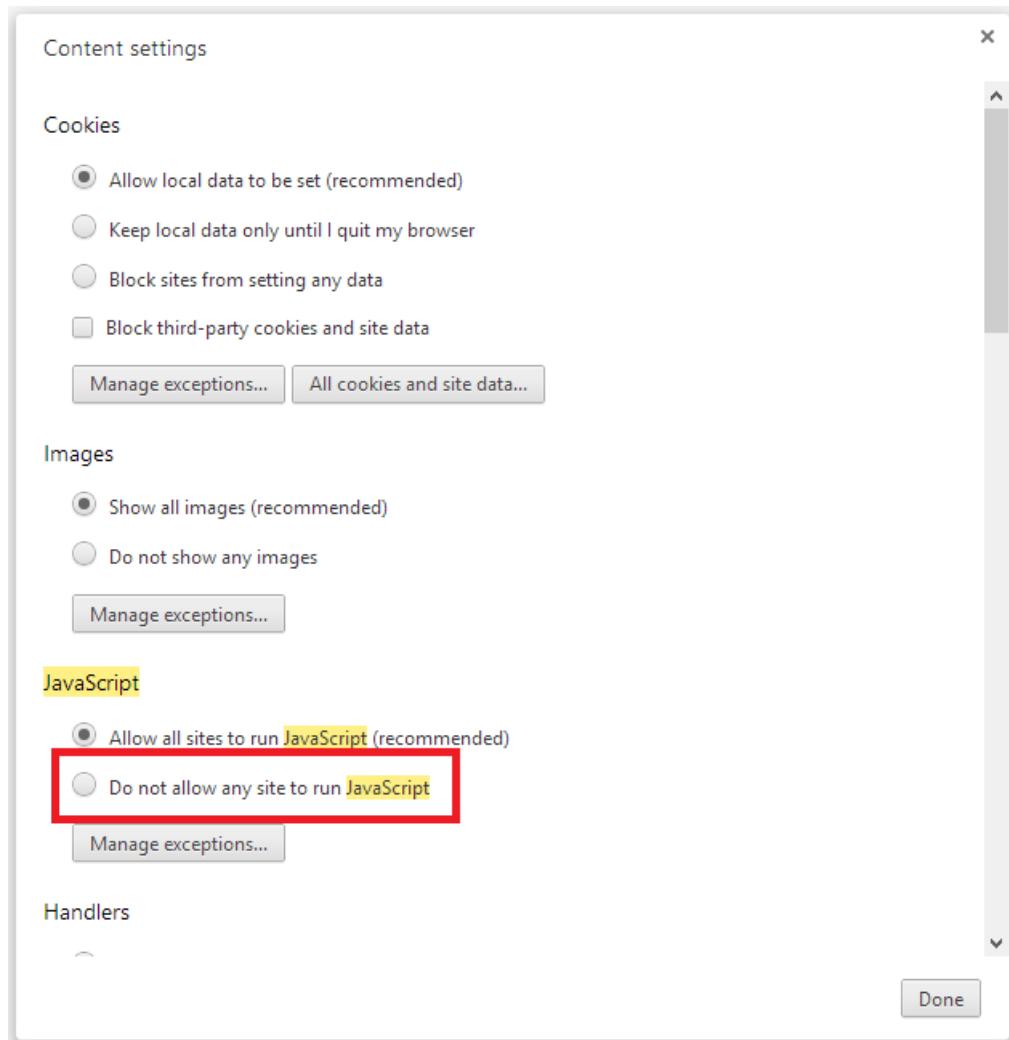
第一个 (HTTP GET) `Create` 操作方法显示初始的“创建”表单。第二个 (`[HttpPost]`) 版本处理表单发布。第二个 `Create` 方法 (`[HttpPost]` 版本) 调用 `ModelState.IsValid` 以检查电影是否有任何验证错误。调用此方法将评估已应用于对象的任何验证特性。如果对象有验证错误，则 `Create` 方法会重新显示此表单。如果没有错误，此方法则将新电影保存在数据库中。在我们的电影示例中，在检测到客户端上存在验证错误时，表单不会发布到服务器。当存在客户端验证错误时，第二个 `Create` 方法永远会被调用。如果在浏览器中禁用 JavaScript，客户端验证将被禁用，而你可以测试 HTTP POST `Create` 方法 `ModelState.IsValid` 检测任何验证错误。

可以在 `[HttpPost] Create` 方法中设置断点，并验证方法从未被调用，客户端验证在检测到存在验证错误时不会提交表单数据。如果在浏览器中禁用 JavaScript，然后提交错误的表单，将触发断点。在没有 JavaScript 的情况下仍然可以进行完整的验证。

以下图片显示如何在 FireFox 浏览器中禁用 JavaScript。



以下图片显示如何在 Chrome 浏览器中禁用 JavaScript。



禁用 JavaScript 后，发布无效数据并单步执行调试程序。

```
74         // POST: Movies/Create
75         // To protect from overposting attacks, please enable t
76         // more details see http://go.microsoft.com/fwlink/?LinkID=142072
77         [HttpPost]
78         [ValidateAntiForgeryToken]
79         public async Task<ActionResult> Create([Bind("ID,Title,
80                                         Description,ReleaseDate,Price")]
81                                         false)
82         {
83             if (ModelState.IsValid)
84             {
85                 _context.Add(movie);
86                 await _context.SaveChangesAsync();
87                 return RedirectToAction("Index");
88             }
89             return View(movie);
90         }
91     }
```

以下是之前在本教程中已搭建基架的 Create.cshtml 视图模板的一部分。以上所示的操作方法使用它来显示初始表单，并在发生错误时重新显示此表单。

```
<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>
        </div>

        @*Markup removed for brevity.*@
    </div>
</form>
```

输入标记帮助程序使用 [DataAnnotations](#) 特性，并在客户端上生成 jQuery 验证所需的 HTML 特性。[验证标记帮助程序](#)用于显示验证错误。有关详细信息，请参阅[验证](#)。

此方法真正好的一点是：无论是控制器还是 `Create` 视图模板都不知道强制实施的实际验证规则或显示的特定错误消息。仅可在 `Movie` 类中指定验证规则和错误字符串。这些相同的验证规则自动应用于 `Edit` 视图和可能创建用于编辑模型的任何其他视图模板。

需要更改验证逻辑时，可以通过将验证特性添加到模型在同一个位置实现此操作。（在此示例中为 `Movie` 类）。无需担心对应用程序的不同部分所强制执行规则的方式不一致 - 所有验证逻辑都将定义在一个位置并用于整个应用程序。这使代码非常简洁，并且更易于维护和改进。这意味着对 DRY 原则的完全遵守。

使用 `DataType` 特性

打开 `Movie.cs` 文件并检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。我们已经在发布日期和价格字段中应用了 `DataType` 枚举值。以下代码显示具有适当 `DataType` 特性的 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 属性仅提供相关提示来帮助视图引擎设置数据格式（并提供元素/属性，例如向 URL 提供 `<a>` 和向电子邮件提供 ``）。可以使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型，它们不是验证属性。在此示例中，我们只想跟踪日期，而不是时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`（读作 `data dash`）特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。（你可能不想为某些字段执行此操作—例如对于货币值，你可能不希望文本框中的货币符号可编辑。）

可以单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能（例如显示日历控件、区域设置适用的货币符号、电子邮件链接等）
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- `DataType` 特性使 MVC 能够选择正确的字段模板来呈现数据（如果 `DisplayFormat` 由自身使用，则使用的字符串模板）。

注意

jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

需要禁用 jQuery 日期验证才能使用具有 `DateTime` 的 `Range` 特性。通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

在本系列的下一部分中，我们将回顾应用程序，并对自动生成的 `Details` 和 `Delete` 方法进行一些改进。

其他资源

- [使用表单](#)
- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)

[上一页](#) [下一页](#)

检查 ASP.NET Core 应用的 Details 和 Delete 方法

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

打开电影控制器，并检查 `Details` 方法：

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

创建此操作方法的 MVC 基架引擎添加显示调用方法的 HTTP 请求的注释。在此情况下，它是包含三个 URL 段的 GET 请求，这三个段为 `Movies` 控制器、`Details` 方法和 `id` 值。回顾这些在 `Startup.cs` 中定义的段。

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

EF 可以使用 `SingleOrDefaultAsync` 方法轻松搜索数据。该方法中内置的一个重要安全功能是，代码会先验证搜索方法已经找到电影，然后再执行操作。例如，一个黑客可能通过将链接创建的 URL 从 `http://localhost:xxxx/Movies/Details/1` 更改为类似 `http://localhost:xxxx/Movies/Details/12345` 的值（或者不代表任何实际电影的其他值）将错误引入站点。如果未检查是否有空电影，则应用可能引发异常。

检查 `Delete` 和 `DeleteConfirmed` 方法。

```

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

请注意，`HTTP GET Delete` 方法不删除指定的电影，而是返回可在其中提交 (HttpPost) 删除的电影视图。执行删除操作以响应 GET 请求(或者说，执行编辑操作、创建操作或更改数据的任何其他操作)会打开安全漏洞。

删除数据的 `[HttpPost]` 方法命名为 `DeleteConfirmed`，以便为 HTTP POST 方法提供一个唯一的签名或名称。下面显示了两个方法签名：

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

公共语言运行时 (CLR) 需要重载方法拥有唯一的参数签名(相同的方法名称但不同的参数列表)。但是，这里需要两个 `Delete` 方法 -- 一个用于 GET，另一个用于 POST -- 这两个方法拥有相同的参数签名。(它们都需要接受单个整数作为参数。)

可通过两种方法解决此问题，一种是为方法提供不同的名称。这正是前面的示例中的基架机制进行的操作。但是，这会造成一个小问题：ASP.NET 按名称将 URL 段映射到操作方法，如果重命名方法，则路由通常无法找到该方法。该示例中也提供了解决方案，即向 `DeleteConfirmed` 方法添加 `ActionName("Delete")` 属性。该属性对路由系统执行映射，以便包括 POST 请求的 /Delete/ 的 URL 可找到 `DeleteConfirmed` 方法。

对于名称和签名相同的方法，另一个常用解决方法是手动更改 POST 方法的签名以包括额外(未使用)的参数。这正是前面的文章中添加 `notUsed` 参数时进行的操作。这里为了 `[HttpPost] Delete` 方法可以执行同样的操作：

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

发布到 Azure

有关如何使用 Visual Studio 将该应用发布到 Azure 的说明, 请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。此外, 还可以从[命令行](#)发布应用。

感谢读完这篇 ASP.NET Core MVC 简介。我们期待你的意见。[MVC 和 EF Core 入门](#)是本教程的优选后续教程。

[上一篇](#)

使用 ASP.NET Core 构建 Web API

2018/5/14 • 3 min to read • [Edit Online](#)

作者: Scott Addie

[查看或下载示例代码\(如何下载\)](#)

本文档说明如何在 ASP.NET Core 中构建 Web API 以及每项功能的最佳适用场景。

从 ControllerBase 派生类

从控制器(旨在用作 Web API)中的 `ControllerBase` 类继承。例如:

```
[!code-csharp]
```

```
[!code-csharp]
```

通过 `ControllerBase` 类可使用大量属性和方法。前面的示例中包含某些此类方法, 如 `BadRequest` 和 `CreatedAtAction`。将在操作方法中调用这些方法以分别返回 HTTP 400 和 201 状态代码。将使用 `ModelState` 属性(还可由 `ControllerBase` 提供)执行请求模型验证。

使用 ApiControllerAttribute 批注类

ASP.NET Core 2.1 引入了 `[ApiController]` 特性, 用于批注 Web API 控制器类。例如:

```
[!code-csharp]
```

此特性通常与 `ControllerBase` 配合使用以获得其他有用的方法和属性。通过 `ControllerBase` 可使用 `NotFound` 和 `File` 等方法。

另一种方法是创建使用 `[ApiController]` 特性进行批注的自定义基本控制器类:

```
[!code-csharp]
```

以下各部分说明该特性添加的便利功能。

自动 HTTP 400 响应

验证错误会自动触发 HTTP 400 响应。操作中不需要以下代码:

```
[!code-csharp]
```

在 `Startup.ConfigureServices` 中使用以下代码将禁用此默认行为:

```
[!code-csharp]
```

绑定源参数推理

绑定源特性定义可找到操作参数值的位置。存在以下绑定源特性:

特性	绑定源
<code>[FromBody]</code>	请求正文
<code>[FromForm]</code>	请求正文中的表单数据
<code>[FromHeader]</code>	请求标头

特性	绑定源
<code>[FromQuery]</code>	请求查询字符串参数
<code>[FromRoute]</code>	当前请求中的路由数据
<code>[FromServices]</code>	作为操作参数插入的请求服务

注意

如果值可能包含 `%2f` (即 `/`)，请不要使用 `[FromRoute]`，因为 `%2f` 不会非转义为 `/`。如果值可能包含 `%2f`，则使用 `[FromQuery]`。

没有 `[ApiController]` 特性时，将显式定义绑定源特性。在下面的示例中，`[FromQuery]` 特性指示 `discontinuedOnly` 参数值在请求 URL 的查询字符串中提供：

[!code-csharp]

推理规则应用于操作参数的默认数据源。这些规则将配置绑定资源，否则你可以手动应用操作参数。绑定源特性的行为如下：

- **[FromBody]**，针对复杂类型参数进行推断。此规则不适用于具有特殊含义的任何复杂的内置类型，如 `IFormCollection` 和 `CancellationToken`。绑定源推理代码将忽略这些特殊类型。当操作中的多个参数为显式指定（通过 `[FromBody]`）或在请求正文作为绑定进行推断时，将会引发异常。例如，下面的操作签名会导致异常：

[!code-csharp]

- **[FromForm]**，针对 `IFormFile` 和 `IFormFileCollection` 类型的操作参数进行推断。该特性不针对任何简单类型或用户定义类型进行推断。
- **[FromRoute]**，针对与路由模板中的参数相匹配的任何操作参数名称进行推断。当多个路由与一个操作参数匹配时，任何路由值都视为 `[FromRoute]`。
- **[FromQuery]**，针对任何其他操作参数进行推断。

在 `Startup.ConfigureServices` 中使用以下代码将禁用默认推理规则：

[!code-csharp]

Multipart/form-data 请求推理

使用 `[FromForm]` 特性批注操作参数时，将推断 `multipart/form-data` 请求内容类型。

在 `Startup.ConfigureServices` 中使用以下代码将禁用默认行为：

[!code-csharp]

特性路由要求

特性路由是必要条件。例如：

[!code-csharp]

不能通过在 `UseMvc` 中定义的 `传统路由` 或通过 `Startup.Configure` 中的 `UseMvcWithDefaultRoute` 访问操作。

其他资源

- [控制器操作返回类型](#)
- [自定义格式化程序](#)
- [格式化响应数据](#)

- 使用 Swagger 的帮助页
- 路由到控制器操作

使用 ASP.NET Core 和 Visual Studio Code 创建 Web API

2018/5/17 • 17 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Mike Wasson](#)

本教程将生成一个用于管理“待办事项”列表的 Web API。不构造 UI。

本教程提供 3 个版本：

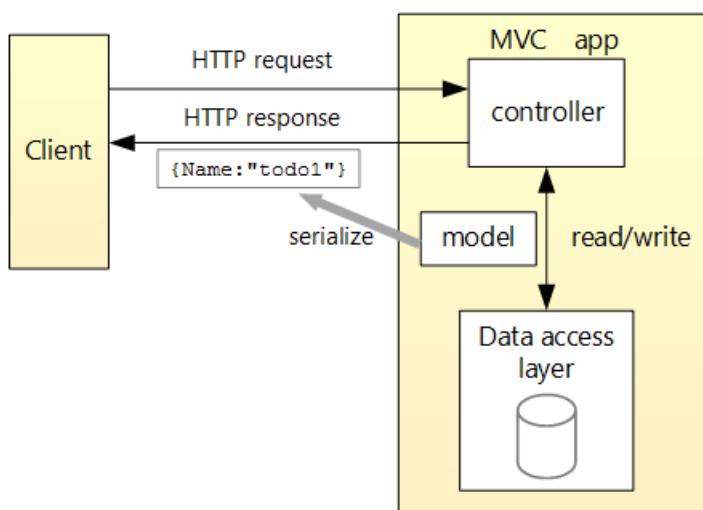
- macOS、Linux、Windows: 使用 Visual Studio Code 创建 Web API(本教程)
- macOS: [使用 Visual Studio for Mac 创建 Web API](#)
- Windows: [使用 Visual Studio for Windows 创建 Web API](#)

概述

本教程将创建以下 API：

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中，唯一的模型是待办事项。模型表示为 C# 类，也称为 Plain Old C# Object (POCO)。

- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

系统必备

Install the following:

- [.NET Core SDK 2.0 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)
- [.NET Core SDK 2.1 RC1 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)

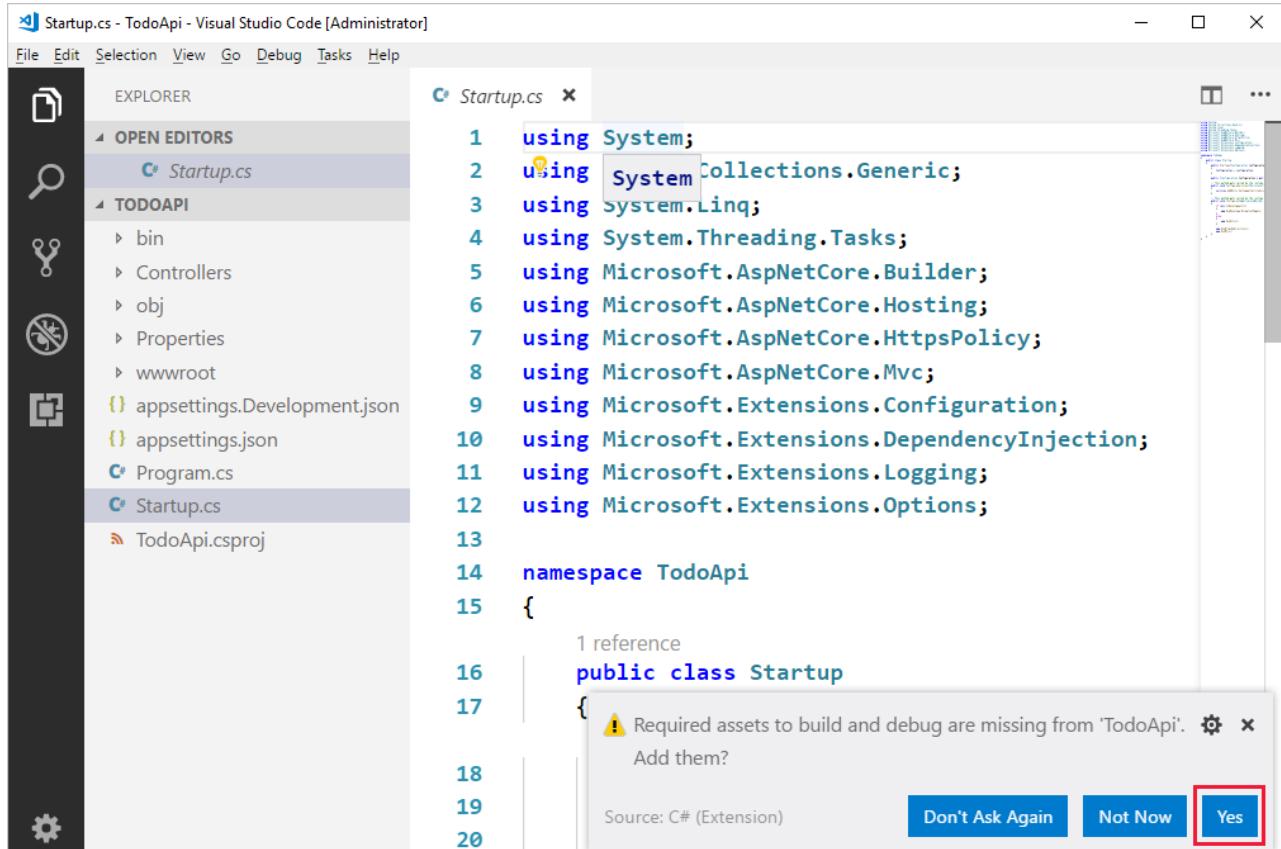
创建项目

从控制台运行以下命令：

```
dotnet new webapi -o TodoApi
code TodoApi
```

在 Visual Studio Code (VS Code) 中打开 TodoApi 文件夹。选择 Startup.cs 文件。

- 对于警告消息 -“TodoApi” 中缺少进行生成和调试所需的资产。是否添加它们？。请选择“是”。
- 对于信息性消息 -“存在未解析的依赖项”，请选择“还原”。



按“调试”(F5) 生成并运行程序。在浏览器中导航到 <http://localhost:5000/api/values>。显示以下输出：

```
["value1","value2"]
```

有关 VS Code 的使用技巧, 请参阅 [Visual Studio Code 帮助](#)。

添加对 Entity Framework Core 的支持

在 ASP.NET Core 2.0 中创建新项目会在 TodoApi.csproj 文件中添加 [Microsoft.AspNetCore.All](#) 包引用:

```
[!code-xml]
```

在 ASP.NET Core 2.1 或更改版本中创建新项目会在 TodoApi.csproj 文件中添加 [Microsoft.AspNetCore.App](#) 包引用:

```
[!code-xml]
```

无需单独安装 [Entity Framework Core InMemory](#) 数据库提供程序。此数据库提供程序允许将 Entity Framework Core 和内存数据库一起使用。

添加模型类

模型是表示应用中的数据的对象。在此示例中, 唯一的模型是待办事项。

添加名为“模型”的文件夹。可将模型类置于项目的任意位置, 但按照惯例会使用“模型”文件夹。

添加带有以下代码的 `TodoItem` 类:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时, 数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调 Entity Framework 功能的主类。将通过从 `Microsoft.EntityFrameworkCore.DbContext` 类派生的方式创建此类。

在“模型”文件夹中添加 `TodoContext` 类:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 Startup.cs 文件的内容替换为以下代码：

```
[!code-csharp]
```

```
[!code-csharp]
```

前面的代码：

- 删除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在“控制器”文件夹中，创建名为 `TodoController` 的类。用以下代码替代其内容：

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以启用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项（如果不存在）添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

```
[!code-csharp]
```

```
[!code-csharp]
```

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
 {  
   "id": 1,  
   "name": "Item1",  
   "isComplete": false  
 }  
]
```

稍后将在本教程中演示如何使用 [Postman](#) 或 [curl](#) 查看 HTTP 响应。

路由和 URL 路径

`[HttpGet]` 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板（如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

```
[!code-csharp]
```

```
[!code-csharp]
```

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。
- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 [JSON](#)，并将 JSON 写入响应消息的正文。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 [IActionResult](#) 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 `ActionResult<T>` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 VS Code 中，按 F5 启动应用。导航到 <http://localhost:5000/api/todo>（我们刚刚创建的 `Todo` 控制器）。

使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 Startup.Configure 中调用 [UseStaticFiles](#) 和 [UseDefaultFiles](#) 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 index.html 的 HTML 文件添加至项目的 wwwroot 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Edit">
            <a onclick="closeInput()" aria-label="Close">Close</a>
        </form>
    </div>
</body>
```

```

<p id="counter"></p>

<table>
  <tr>
    <th>Is Complete</th>
    <th>Name</th>
    <th></th>
    <th></th>
  </tr>
  <tbody id="todos"></tbody>
</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
        integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
        crossorigin="anonymous"></script>
<script src="site.js"></script>
</body>
</html>

```

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
  const el = $('#counter');
  let name = 'to-do';
  if (data) {
    if (data > 1) {
      name = 'to-dos';
    }
    el.text(data + ' ' + name);
  } else {
    el.html('No ' + name);
  }
}

$(document).ready(function () {
  getData();
});

function getData() {
  $.ajax({
    type: 'GET',
    url: uri,
    success: function (data) {
      $('#todos').empty();
      getCount(data.length);
      $.each(data, function (key, item) {
        const checked = item.isComplete ? 'checked' : '';
        $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
          '<td>' + item.name + '</td>' +
          '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
          '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
          '</tr>').appendTo($('#todos'));
      });
      todos = data;
    }
  });
}

function addItem() {
  const item = {
    'name': $('#add-name').val(),
    'isComplete': false
  };
}

```

```

$.ajax({
    type: 'POST',
    accepts: 'application/json',
    url: uri,
    contentType: 'application/json',
    data: JSON.stringify(item),
    error: function (jqXHR, textStatus, errorThrown) {
        alert('here');
    },
    success: function (result) {
        getData();
        $('#add-name').val('');
    }
});

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            $('#edit-name').val(item.name);
            $('#edit-id').val(item.id);
            $('#edit-isComplete').val(item.isComplete);
        }
    });
    $('#spoiler').css({ 'display': 'block' });
}

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });

    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```
$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';
                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}
```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```

$.ajax({
    url: uri + '/' + $('#edit-id').val(),
    type: 'PUT',
    accepts: 'application/json',
    contentType: 'application/json',
    data: JSON.stringify(item),
    success: function (result) {
        getData();
    }
});

```

删除待办事项

若要删除待办事项，请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```

$.ajax({
    url: uri + '/' + id,
    type: 'DELETE',
    success: function (result) {
        getData();
    }
});

```

实现其他的 CRUD 操作

在以下部分中，将 `Create`、`Update` 和 `Delete` 方法添加到控制器。

创建

添加以下 `Create` 方法：

[!code-csharp]

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

```
[!code-csharp]
```

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。MVC 从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法：

- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 [10.2.2 201 已创建](#)。
- 使用名为 `route` 的“`GetTodo`”来创建 URL。已在 `GetById` 中定义名为 `route` 的“`GetTodo`”：

```
[!code-csharp]
```

```
[!code-csharp]
```

使用 Postman 发送创建请求

- 启动该应用程序。
- 打开 Postman。

The screenshot shows the Postman application interface. The top navigation bar includes File, Edit, View, Help, New, Import, Runner, My Workspace, and IN SYNC status. The main workspace shows a POST request to `https://localhost:44375/api/todo`. The Body tab is selected, showing the raw JSON input:

```
1 [{}  
2   "name": "walk dog",  
3   "isComplete": true  
4 ]
```

The Headers section shows three entries: `Content-Type: application/json`, `Accept: */*`, and `Host: localhost:44375`. The Body section also shows the JSON input. The bottom section displays the response: Status: 201 Created, Time: 148 ms, and the JSON response body:

```
1 {  
2   "id": 6,  
3   "name": "walk dog",  
4   "isComplete": true  
5 }
```

- 更新 localhost URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 输入包含待办事项的请求正文，类似以下 JSON：

```
{  
    "name": "walk dog",  
    "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应，则禁用“SSL 证书验证”选项。在“文件”>“设置”下可以找到该选项。在禁用该设置后，再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡，然后复制位置标头值：

The screenshot shows the Postman application interface. A POST request is made to `https://localhost:44375/api/todo`. The request body is set to raw JSON:

```
1 [{}  
2     "name": "walk dog",  
3     "isComplete": true  
4 ]
```

The Headers section is highlighted with a red box. It contains the following entries:

- Content-Type → application/json; charset=utf-8
- Date → Fri, 27 Apr 2018 18:32:32 GMT
- Location → `https://localhost/api/Todo/6`

位置标头 URI 可用于访问新项。

更新

添加以下 `Update` 方法：

```
![code-csharp]
```

```
![code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 204(无内容)。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

使用 Postman 将待办事项的名称更新为“带猫出去散步”：

The screenshot shows the Postman interface with the following details:

- Method:** PUT (highlighted with a red box)
- URL:** https://localhost:44375/api/todo/1
- Body:** Body tab selected (highlighted with a red box).
 - Content Type: raw (highlighted with a red box)
 - JSON (application/json) (highlighted with a red box)
- Body Content:**

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```
- Response:**
 - Status: 204 No Content
 - Time: 70 ms

删除

添加以下 `Delete` 方法:

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

`Delete` 响应是 204(无内容)。

使用 Postman 删除待办事项:

The screenshot shows the Postman application interface. At the top, there's a header bar with the Postman logo, file menu (File, Edit, View, Help), and workspace status (My Workspace, IN SYNC). Below the header are buttons for New, Import, Runner, and a plus sign. A toolbar on the right includes icons for environment management and sync.

The main area shows three tabs for environments: https://localhost:4437, https://localhost:4437, and https://localhost:4437. A 'No Environment' button is also present. Below these tabs, a red box highlights the 'DELETE' method dropdown, followed by the URL 'https://localhost:44375/api/todo/1'. To the right are 'Params' and 'Send' buttons.

The 'Authorization' tab is selected, showing 'No Auth' selected in the dropdown. A note states: 'The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)'. To the right, another note says: 'This request does not use any authorization. [Learn more about authorization](#)'.

Below the authorization section, tabs for Body, Cookies, Headers (4), and Test Results are visible. The Headers tab is selected. The Headers section shows four entries: 'Content-Type: application/json', 'Content-Length: 1', 'Accept: */*', and 'Host: localhost:44375'. The Body tab is selected, showing a JSON response with a single item: { "id": 1 }. The response status is 204 No Content and the time taken is 735 ms.

Visual Studio Code 帮助

- [入门](#)
- [调试](#)
- [集成终端](#)
- [键盘快捷键](#)
 - [macOS 键盘快捷方式](#)
 - [Linux 键盘快捷键](#)
 - [Windows 键盘快捷键](#)

后续步骤

- 有关使用永久数据库的详细信息, 请参阅:
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)

- 有关部署 API 的信息(包括部署到 Azure 应用服务), 请参阅[托管和部署](#)。
- [查看或下载示例代码](#)。请参阅[如何下载](#)。

使用 ASP.NET Core 和 Visual Studio for Mac 创建 Web API

2018/5/14 • 17 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Mike Wasson](#)

注意

ASP.NET Core 2.1 is in preview and not recommended for production use.

本教程将生成一个用于管理“待办事项”列表的 Web API。不构造 UI。

本教程提供 3 个版本：

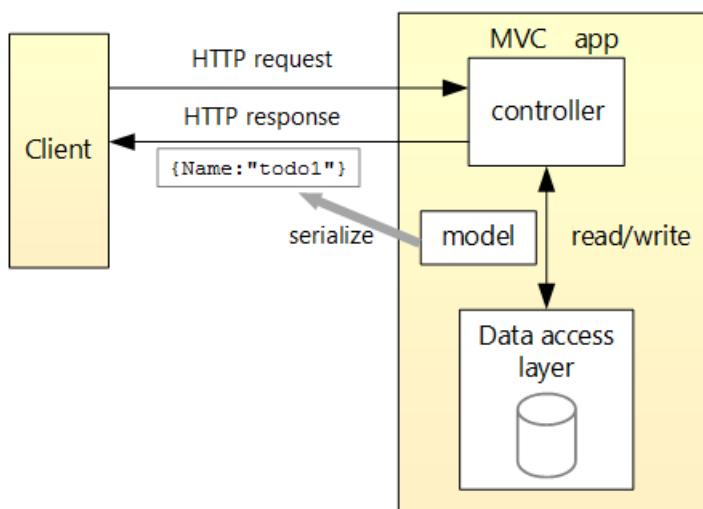
- macOS: 使用 Visual Studio for Mac 创建 Web API (本教程)
- Windows: [使用 Visual Studio for Windows 创建 Web API](#)
- macOS、Linux、Windows: [使用 Visual Studio Code 创建 Web API](#)

概述

本教程将创建以下 API：

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中，唯一的模型是待办事项。模型表示为 C# 类，也称为 Plain Old C# Object (POCO)。
- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

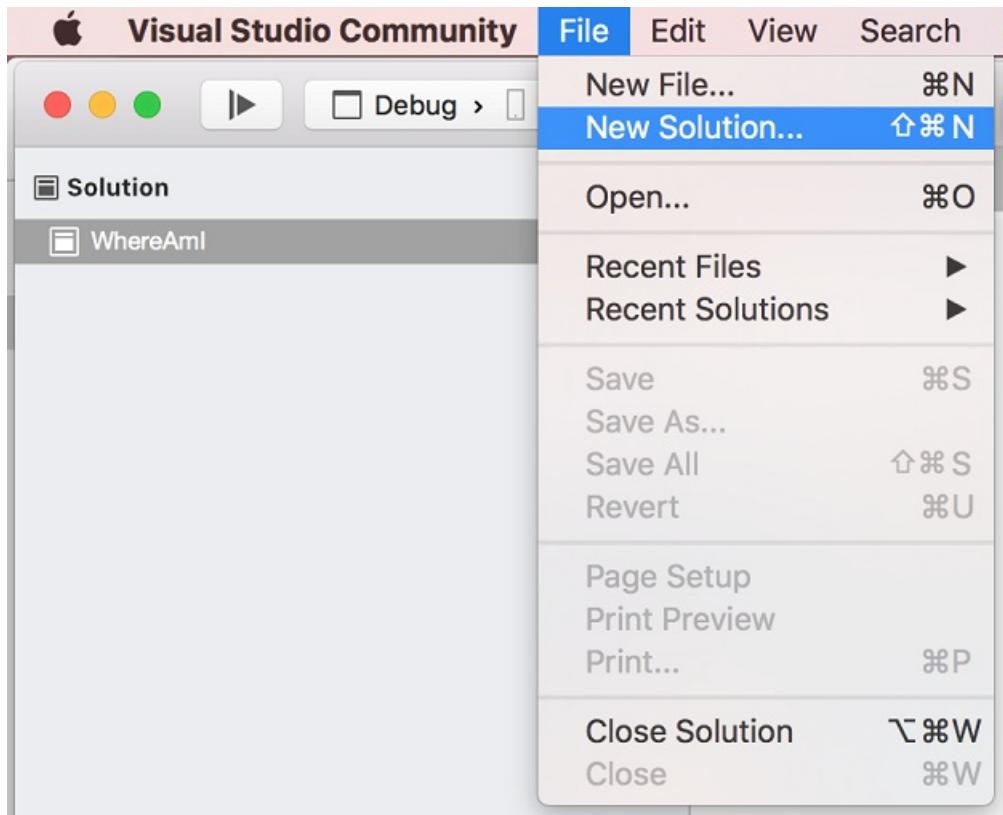
请参阅 [macOS 或 Linux 上的 ASP.NET Core MVC 介绍](#)，获取使用永久数据库的示例。

系统必备

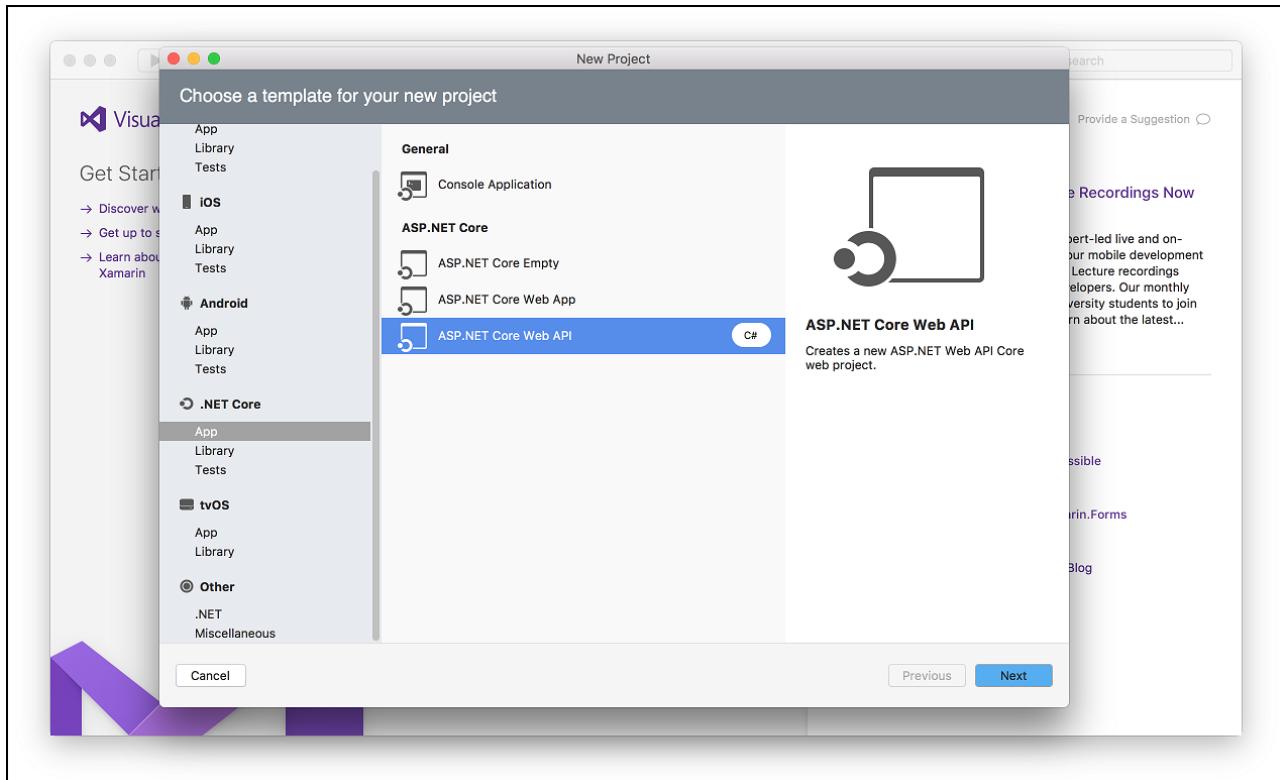
[Visual Studio for Mac](#)

创建项目

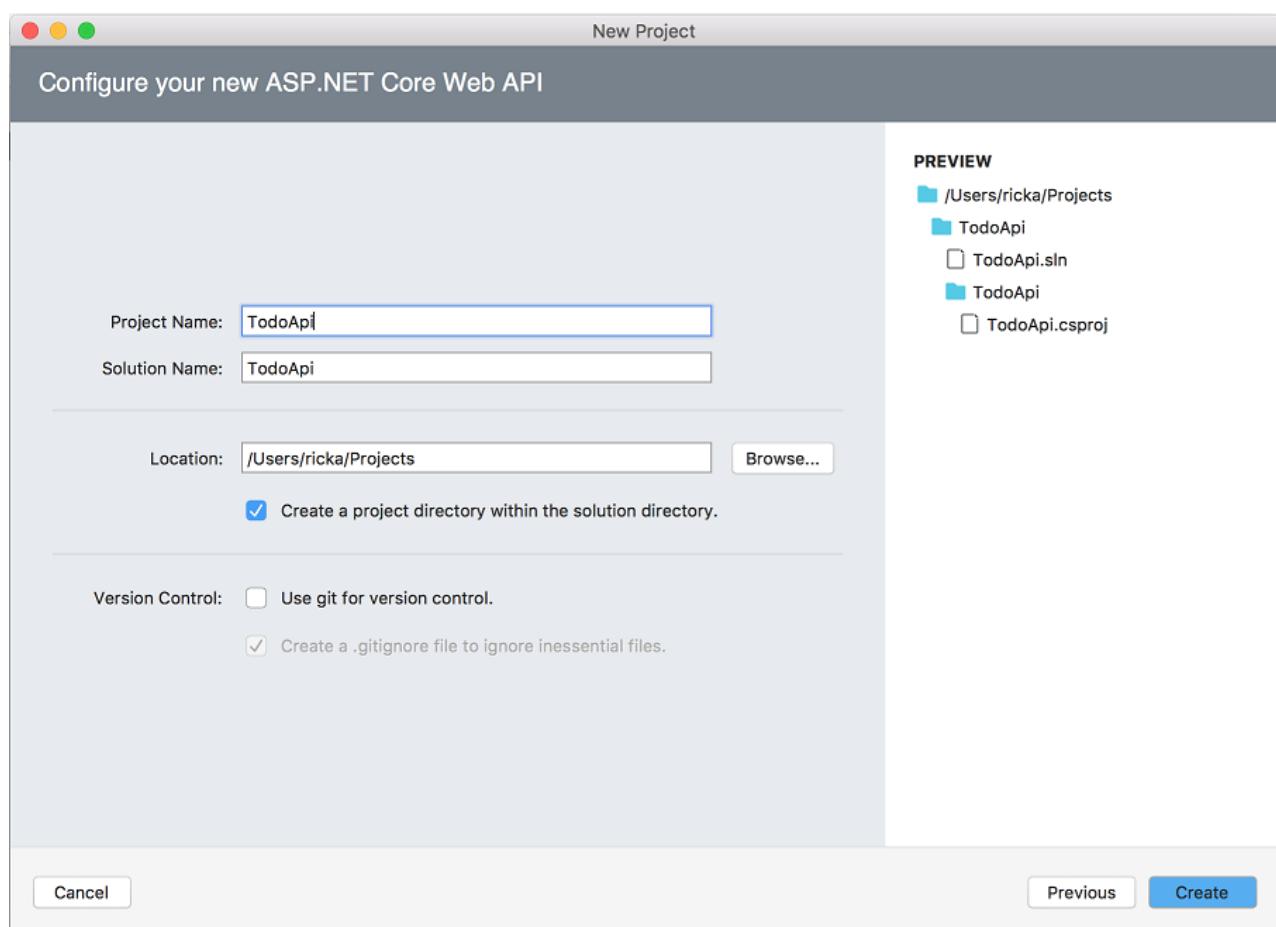
在 Visual Studio 中，选择“文件”>“新建解决方案”。



选择“.NET Core App”>“ASP.NET Core Web API”>“下一步”。



输入“TodoApi”作为“项目名称”，然后选择“创建”。



启动应用

在 Visual Studio 中，选择“运行” > “开始执行(调试)”启动应用。Visual Studio 启动浏览器并导航到 <http://localhost:5000>。将收到 HTTP 404(未找到)错误。将 URL 更改为 <http://localhost:<port>/api/values>。
ValuesController 数据已显示：

```
["value1","value2"]
```

添加对 Entity Framework Core 的支持

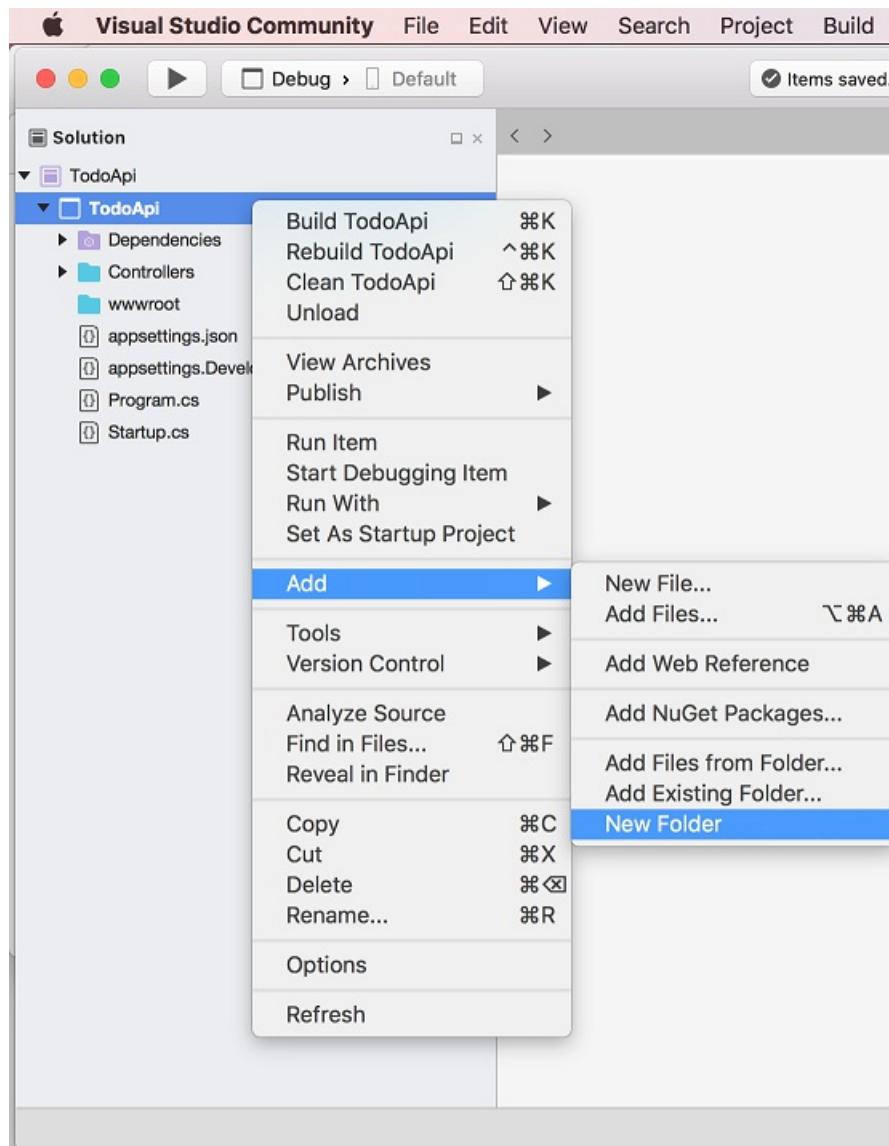
安装 [Entity Framework Core InMemory](#) 数据库提供程序。此数据库提供程序允许将 Entity Framework Core 和内存数据库一起使用。

- 从“项目”菜单中，选择“添加 NuGet 包”。
- 或者，可以右键单击“依赖项”，然后选择“添加包”。
- 在搜索框中输入 `EntityFrameworkCore.InMemory`。
- 选择 `Microsoft.EntityFrameworkCore.InMemory`，然后选择“添加包”。

添加模型类

模型是表示应用中的数据的对象。在此示例中，唯一的模型是待办事项。

在解决方案资源管理器中，右键单击项目。选择“添加”>“新建文件夹”。将文件夹命名为“Models”。



注意

可将模型类置于项目的任意位置，但按照惯例会使用“模型”文件夹。

右键单击“Models”文件夹，然后选择“添加”>“新建文件”>“常规”>“空类”。将类命名为“TodoItem”，然后单击“新

建”。

将生成的代码替换为：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时，数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调 Entity Framework 功能的主类。可以通过继承 `Microsoft.EntityFrameworkCore.DbContext` 类的方式创建此类。

将 `TodoContext` 类添加到“Models”文件夹。

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 `Startup.cs` 文件的内容替换为以下代码：

```
[!code-csharp]
```

```
[!code-csharp]
```

前面的代码：

- 删除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在解决方案资源管理器的“控制器”文件夹中，添加 `TodoController` 类。

将生成的代码替换为以下代码：

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

[!code-csharp]

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以启用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项(如果不存在)添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

[!code-csharp]

[!code-csharp]

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
 {  
   "id": 1,  
   "name": "Item1",  
   "isComplete": false  
 }  
]
```

稍后将在本教程中演示如何使用 [Postman](#) 或 [curl](#) 查看 HTTP 响应。

路由和 URL 路径

`[HttpGet]` 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

[!code-csharp]

[!code-csharp]

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板(如 `[HttpGet("/products")]`)，则将它追加到路径。此示例不使用模板。有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

[!code-csharp]

[!code-csharp]

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。

- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 JSON，并将 JSON 写入响应消息的正文 中。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 `IActionResult` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类 型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 `ActionResult<T>` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 Visual Studio 中，选择“运行”>“开始执行(调试)”启动应用。Visual Studio 启动浏览器并导航到

`http://localhost:<port>`，其中 `<port>` 是随机选择的端口号。将收到 HTTP 404(未找到)错误。将 URL 更改为 `http://localhost:<port>/api/values`。`ValuesController` 数据已显示：

```
["value1","value2"]
```

导航到位于 `http://localhost:<port>/api/todo` 的 `Todo` 控制器：

```
[{"key":1,"name":"Item1","isComplete":false}]
```

实现其他的 CRUD 操作

我们将向控制器添加 `Create`、`Update` 和 `Delete` 方法。这些方法是主题的变体，因此在这里只提供代码并突出显 示主要的差异。添加或更改代码后生成项目。

创建

```
![code-csharp]
```

上述方法响应至 HTTP POST，由 `[HttpPost]` 属性指示。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

```
![code-csharp]
```

上述方法响应至 HTTP POST，由 `[HttpPost]` 属性指示。MVC 从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法返回 201 响应。201 响应是在服务器上创建新资源的 HTTP POST 方法的标准响应。

`CreatedAtRoute` 还会向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 [10.2.2 201 已创建](#)。

使用 Postman 发送创建请求

- 启动应用(“运行”>“开始执行(调试)”)。
- 打开 Postman。

The screenshot shows the Postman application interface. At the top, there's a toolbar with 'File', 'Edit', 'View', 'Help' menus, and icons for 'New', 'Import', 'Runner', and 'My Workspace'. Below the toolbar, the URL bar shows 'https://localhost:44375' with a red dot indicating it's a secure connection. To the right of the URL bar is a 'No Environment' dropdown. The main workspace shows a 'POST' request to 'https://localhost:44375/api/todo'. The 'Body' tab is selected, indicated by a blue dot. Under 'Body', the type is set to 'raw' and the content type is 'JSON (application/json)'. The JSON payload is: { "name": "walk dog", "isComplete": true }. Below the request, the 'Body' section shows the response: { "id": 6, "name": "walk dog", "isComplete": true }. The status is '201 Created' and the time taken is '148 ms'. There are tabs for 'Pretty', 'Raw', and 'Preview'.

- 更新 localhost URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 输入包含待办事项的请求正文, 类似以下 JSON:

```
{  
  "name": "walk dog",  
  "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应, 则禁用“SSL 证书验证”选项。在“文件” > “设置”下可以找到该选项 在禁用该设置后, 再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡, 然后复制位置标头值:

The screenshot shows the Postman application interface. At the top, there's a toolbar with 'File', 'Edit', 'View', 'Help' and various icons like 'New', 'Import', 'Runner'. The main area has tabs for 'My Workspace' and 'IN SYNC'. Below that, a search bar shows 'https://localhost:44375' and a dropdown with '+ ...'. To the right, it says 'No Environment'.

The request details show a 'POST' method to 'https://localhost:44375/api/todo'. The 'Body' tab is selected, showing a JSON payload:

```
1 [{}  
2   "name": "walk dog",  
3   "isComplete": true  
4 }]
```

Below the body, the 'Headers' tab is highlighted with a red box, showing:

- Content-Type → application/json; charset=utf-8
- Date → Fri, 27 Apr 2018 18:32:32 GMT
- Location → <https://localhost/api/Todo/6>

The status bar at the bottom indicates 'Status: 201 Created' and 'Time: 148 ms'.

可以使用此位置标头 URI 访问创建的资源。`Create` 方法返回 `CreatedAtRoute`。传递至 `CreatedAtRoute` 的第一个参数代表用于生成 URL 的命名路由。请回想一下，`GetById` 方法创建的 `"GetTodo"` 命令路由：

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

更新

```
[!code-csharp]
```

```
[!code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 `204(无内容)`。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

```
{  
  "key": 1,  
  "name": "walk dog",  
  "isComplete": true  
}
```

The screenshot shows the Postman application interface. A red box highlights the 'PUT' method dropdown. Another red box highlights the 'Body' tab. A third red box highlights the 'raw' radio button under the body type dropdown. A fourth red box highlights the JSON (application/json) option in the dropdown. The request URL is https://localhost:44375/api/todo/1. The body content is:

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```

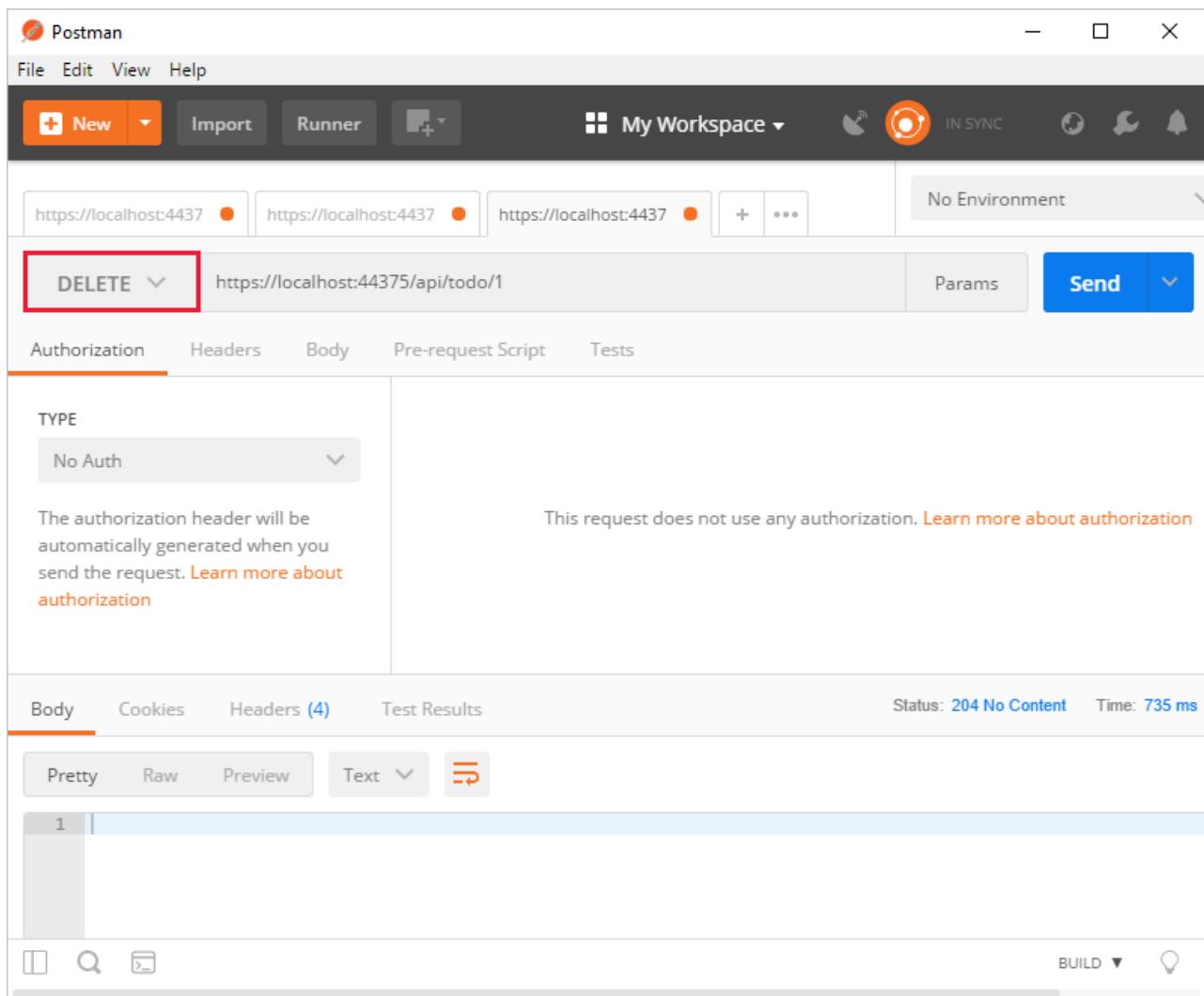
The response status is 204 No Content and the time taken is 70 ms.

删除

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

响应是 204(无内容)。



使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 `Startup.Configure` 中调用 `UseStaticFiles` 和 `UseDefaultFiles` 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 `index.html` 的 HTML 文件添加至项目的 `wwwroot` 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
```

```

        display: none;
    }

    table {
        font-family: Arial, sans-serif;
        border: 1px solid;
        border-collapse: collapse;
    }

    th {
        background-color: #0066CC;
        color: white;
    }

    td {
        border: 1px solid;
        padding: 5px;
    }

```

</style>

</head>

<body>

<h1>To-do CRUD</h1>

<h3>Add</h3>

<form action="javascript:void(0);" method="POST" onsubmit="addItem()">

<input type="text" id="add-name" placeholder="New to-do">

<input type="submit" value="Add">

</form>

<div id="spoiler">

<h3>Edit</h3>

<form class="my-form">

<input type="hidden" id="edit-id">

<input type="checkbox" id="edit-isComplete">

<input type="text" id="edit-name">

<input type="submit" value="Edit">

×

</form>

</div>

<p id="counter"></p>

<table>

<tr>

<th>Is Complete</th>

<th>Name</th>

<th></th>

<th></th>

</tr>

<tbody id="todos"></tbody>

</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
 integrity="sha256-FgpCb/KJQ1LNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
 crossorigin="anonymous"></script>

<script src="site.js"></script>

</body>

</html>

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
    const el = $('#counter');
    let name = 'to-do';
    if (data) {
        if (data > 1) {
            name = 'to-dos';
        }
        el.text(name + ' ' + data);
    } else {
        el.text('to-do');
    }
}

```

```

        name = todos ;
    }
    el.text(data + ' ' + name);
} else {
    el.html('No ' + name);
}
}

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';
                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };
    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            ...
        }
    });
}

```

```

        $('#edit-name').val(item.name);
        $('#edit-id').val(item.id);
        $('#edit-isComplete').val(item.isComplete);
    }
});

$('#spoiler').css({ 'display': 'block' });

}

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });

    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';

                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
        }
    });
}

```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```
$.ajax({
    url: uri + '/' + $('#edit-id').val(),
    type: 'PUT',
    accepts: 'application/json',
    contentType: 'application/json',
    data: JSON.stringify(item),
    success: function (result) {
        getData();
    }
});
```

删除待办事项

若要删除待办事项, 请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
    url: uri + '/' + id,
    type: 'DELETE',
    success: function (result) {
        getData();
    }
});
```

后续步骤

- 有关使用永久数据库的详细信息, 请参阅:
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)
- 有关部署 API 的信息(包括部署到 Azure 应用服务), 请参阅[托管和部署](#)。
- [查看或下载示例代码](#)。请参阅[如何下载](#)。

使用 ASP.NET Core 和 Visual Studio for Windows 创建 Web API

2018/5/17 • 17 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Mike Wasson](#)

本教程构建了用于管理“待办事项”列表的 Web API。未创建用户界面 (UI)。

本教程提供 3 个版本:

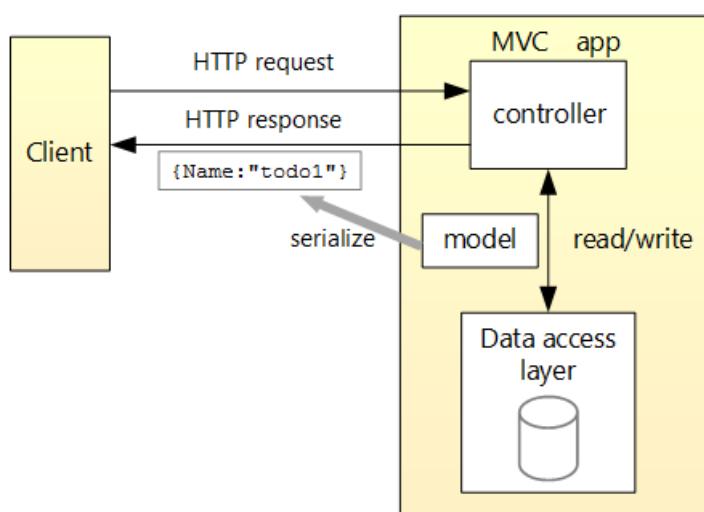
- Windows: 使用 Visual Studio for Windows 创建 Web API(本教程)
- macOS: [使用 Visual Studio for Mac 创建 Web API](#)
- macOS、Linux、Windows: [使用 Visual Studio Code 创建 Web API](#)

概述

本教程将创建以下 API:

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中, 唯一的模型是待办事项。模型表示为 C# 类, 也称为 Plain Old C# Object (POCO)。

- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

系统必备

Visual Studio for Windows

- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate

创建项目

在 Visual Studio 中执行以下步骤：

- 从“文件”菜单中选择“新建” > “项目”。
- 选择“ASP.NET Core Web 应用程序”模板。将项目命名为 TodoApi，然后单击“确定”。
- 在“新建 ASP.NET Core Web 应用程序 - TodoApi”对话框中，选择 ASP.NET Core 版本。选择“API”模板，然后单击“确定”。请不要选择“启用 Docker 支持”。

启动应用

在 Visual Studio 中，按 CTRL+F5 启动应用。Visual Studio 启动浏览器并导航到

`http://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。Chrome、Microsoft Edge 和 Firefox 将显示以下输出：

```
["value1", "value2"]
```

添加模型类

模型是表示应用中的数据的对象。在此示例中，唯一的模型是待办事项。

在解决方案资源管理器中，右键单击项目。选择“添加” > “新建文件夹”。将文件夹命名为“Models”。

注意

模型类可以出现在项目的任意位置。Models 文件夹按约定用于模型类。

在解决方案资源管理器中右键单击“模型”文件夹，然后选择“添加” > “类”。将类命名为 TodoItem，然后单击“添加”。

使用以下代码更新 `TodoItem` 类：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时，数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调实体框架功能的主类。此类由 `Microsoft.EntityFrameworkCore.DbContext` 类派生而来。

在解决方案资源管理器中右键单击“模型”文件夹，然后选择“添加”>“类”。将类命名为 `TodoContext`，然后单击“添加”。

将该类替换为以下代码：

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {

        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 `Startup.cs` 文件的内容替换为以下代码：

[!code-csharp]

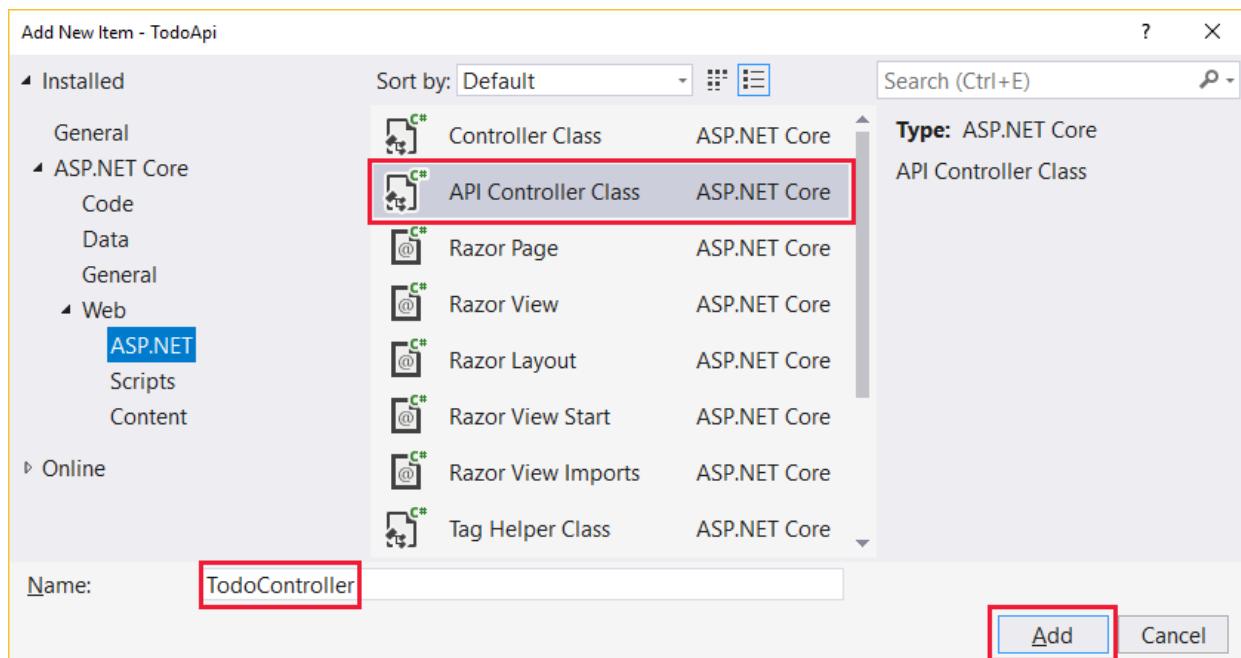
[!code-csharp]

前面的代码：

- 删除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在解决方案资源管理器中，右键单击“控制器”文件夹。选择 **添加** > **新建项**。在“添加新项”对话框中，选择“API 控制器类”模板。将类命名为 `TodoController`，然后单击“添加”。



将该类替换为以下代码：

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

```
[!code-csharp]
```

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以启用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项(如果不存在)添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

```
[!code-csharp]
```

```
[!code-csharp]
```

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
  {  
    "id": 1,  
    "name": "Item1",  
    "isComplete": false  
  }  
]
```

稍后将在本教程中演示如何使用 [Postman](#) 或 [curl](#) 查看 HTTP 响应。

路由和 URL 路径

[HttpGet] 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板（如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

```
[!code-csharp]
```

```
[!code-csharp]
```

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。
- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 `JSON`，并将 `JSON` 写入响应消息的正文中。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 `IActionResult` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 `JSON` 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 `ActionResult<T>` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 `JSON` 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 Visual Studio 中，按 `CTRL+F5` 启动应用。Visual Studio 启动浏览器并导航到

`http://localhost:<port>/api/values`，其中 `<port>` 是随机选择的端口号。导航到位于 `http://localhost:<port>/api/todo` 的 `Todo` 控制器。

实现其他的 CRUD 操作

在以下部分中，将 `Create`、`Update` 和 `Delete` 方法添加到控制器。

创建

添加以下 `Create` 方法：

```
[!code-csharp]
```

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

```
[!code-csharp]
```

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。MVC 从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法：

- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 [10.2.2 201 已创建](#)。
- 使用名为 `route` 的“`GetTodo`”来创建 URL。已在 `GetById` 中定义名为 `route` 的“`GetTodo`”：

```
[!code-csharp]
```

```
[!code-csharp]
```

使用 Postman 发送创建请求

- 启动该应用程序。
- 打开 Postman。

The screenshot shows the Postman application window. The top navigation bar includes File, Edit, View, Help, New, Import, Runner, My Workspace, and various status indicators. The main interface shows a search bar with 'https://localhost:44375' and a dropdown menu. Below it, a 'POST' method is selected for the request type, and the URL is set to 'https://localhost:44375/api/todo'. The 'Body' tab is active, and the 'raw' radio button is selected under the 'JSON (application/json)' type. The JSON payload is defined as:

```
1 {  
2   "name": "walk dog",  
3   "isComplete": true  
4 }
```

In the bottom section, the 'Body' tab is selected, showing the response status as '201 Created' and time as '148 ms'. The response body is displayed in Pretty, Raw, and Preview formats, showing the newly created todo item with id 6.

- 更新 localhost URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 输入包含待办事项的请求正文，类似以下 JSON：

```
{  
  "name": "walk dog",  
  "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应，则禁用“SSL 证书验证”选项。在“文件”>“设置”下可以找到该选项。在禁用该设置后，再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡，然后复制位置标头值：

The screenshot shows the Postman application interface. At the top, there's a navigation bar with File, Edit, View, Help, New, Import, Runner, My Workspace, and various status indicators. Below the bar, a search bar contains the URL https://localhost:44375. To the right of the search bar, it says 'No Environment'. Underneath, a request card shows a POST method pointing to https://localhost:44375/api/todo. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "name": "walk dog",  
3   "isComplete": true  
4 }
```

Below the body, the 'Headers' tab is highlighted with a red box around it, indicating it's the active tab. Other tabs shown are Body, Cookies, and Test Results. At the bottom of the response pane, the status is listed as 'Status: 201 Created Time: 148 ms'. The 'Location' header value, https://localhost/api/Todo/6, is also highlighted with a red box.

位置标头 URI 可用于访问新项。

更新

添加以下 `Update` 方法：

```
[!code-csharp]
```

```
[!code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 204(无内容)。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

使用 Postman 将待办事项的名称更新为“带猫出去散步”：

The screenshot shows the Postman application interface. At the top, there are tabs for 'File', 'Edit', 'View', and 'Help'. Below the tabs, there are buttons for 'New', 'Import', 'Runner', and 'My Workspace'. The 'My Workspace' dropdown is set to 'IN SYNC'. On the right side of the header, there are icons for environment management, sync status, and notifications.

In the main workspace, there are two tabs: 'https://localhost:4437' and 'https://localhost:44375'. A red box highlights the 'PUT' method dropdown. The URL 'https://localhost:44375/api/todo/1' is entered in the address bar. To the right of the URL, there are buttons for 'Params' and 'Send'.

Below the address bar, there are tabs for 'Authorization', 'Headers (3)', 'Body (●)', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected and highlighted with a red box. Under the 'Body' tab, there are four options: 'form-data', 'x-www-form-urlencoded', 'raw', and 'binary'. The 'raw' option is selected and highlighted with a red box. To the right of the 'raw' button, there is a dropdown menu set to 'JSON (application/json)'.

The 'Body' section contains a code editor with the following JSON data:

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```

Below the code editor, there are tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results'. The 'Body' tab is selected and highlighted with a red box. To the right, it shows 'Status: 204 No Content' and 'Time: 70 ms'. Below these status indicators, there are buttons for 'Pretty', 'Raw', 'Preview', 'Text', and a copy icon.

删除

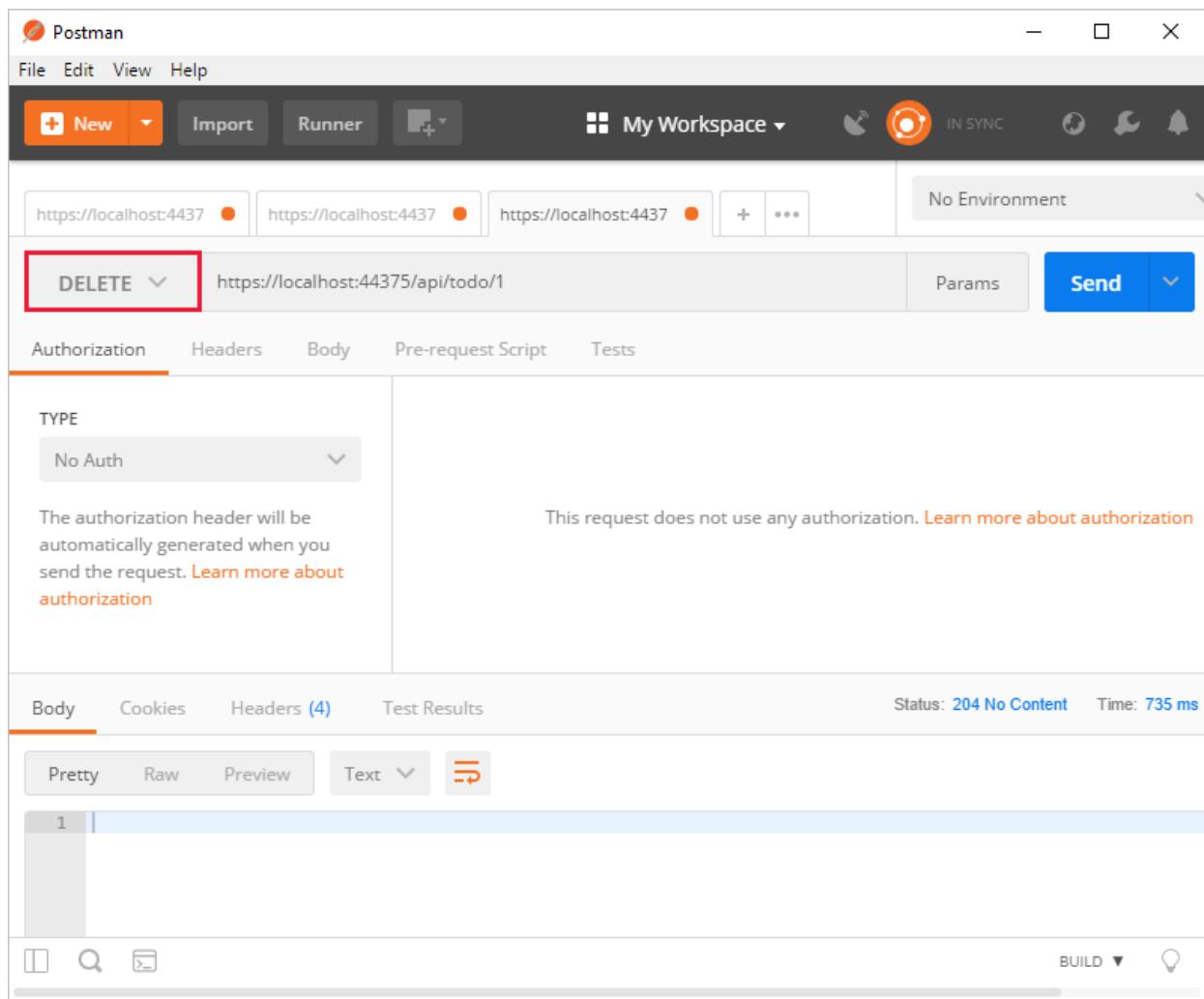
添加以下 `Delete` 方法：

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

`Delete` 响应是 204(无内容)。

使用 Postman 删除待办事项：



使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 `Startup.Configure` 中调用 [UseStaticFiles](#) 和 [UseDefaultFiles](#) 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 `index.html` 的 HTML 文件添加至项目的 `wwwroot` 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }
    </style>

```

```

        }

    table {
        font-family: Arial, sans-serif;
        border: 1px solid;
        border-collapse: collapse;
    }

    th {
        background-color: #0066CC;
        color: white;
    }

    td {
        border: 1px solid;
        padding: 5px;
    }

```

</style>

</head>

<body>

<h1>To-do CRUD</h1>

<h3>Add</h3>

<form action="javascript:void(0);" method="POST" onsubmit="addItem()">

<input type="text" id="add-name" placeholder="New to-do">

<input type="submit" value="Add">

</form>

<div id="spoiler">

<h3>Edit</h3>

<form class="my-form">

<input type="hidden" id="edit-id">

<input type="checkbox" id="edit-isComplete">

<input type="text" id="edit-name">

<input type="submit" value="Edit">

✖

</form>

</div>

<p id="counter"></p>

<table>

<tr>

<th>Is Complete</th>

<th>Name</th>

<th></th>

<th></th>

</tr>

<tbody id="todos"></tbody>

</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
 integrity="sha256-FgpcB/KJQ1LNf0u91ta32o/NMZxltwRo8QtmkMRdAu8="
 crossorigin="anonymous"></script>

<script src="site.js"></script>

</body>

</html>

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
    const el = $('#counter');
    let name = 'to-do';
    if (data) {
        if (data > 1) {
            name = 'to-dos';
        }
    }
}

```

```

        }
        el.text(data + ' ' + name);
    } else {
        el.html('No ' + name);
    }
}

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';
                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };
    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            $('#edit-name').val(item.name);
        }
    });
}

```

```

        $('#edit-id').val(item.id);
        $('#edit-isComplete').val(item.isComplete);
    }
});

$('#spoiler').css({ 'display': 'block' });

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });
    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';
                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')>Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')>Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}

```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```
$.ajax({
  url: uri + '/' + $('#edit-id').val(),
  type: 'PUT',
  accepts: 'application/json',
  contentType: 'application/json',
  data: JSON.stringify(item),
  success: function (result) {
    getData();
  }
});
```

删除待办事项

若要删除待办事项，请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
  url: uri + '/' + id,
  type: 'DELETE',
  success: function (result) {
    getData();
  }
});
```

后续步骤

- 有关使用永久数据库的详细信息，请参阅：
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)
- 有关部署 API 的信息(包括部署到 Azure 应用服务)，请参阅[托管和部署](#)。
- [查看或下载示例代码](#)。请参阅[如何下载](#)。

使用 ASP.NET Core 为本机移动应用创建后端服务

2018/5/14 • 9 min to read • [Edit Online](#)

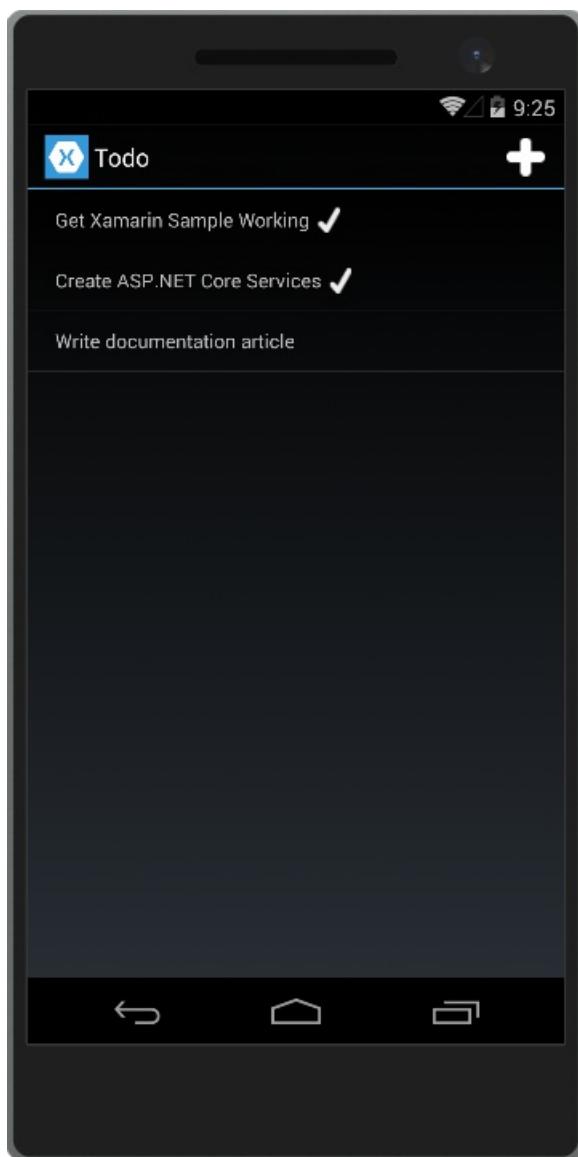
作者: [Steve Smith](#)

移动应用可与 ASP.NET Core 后端服务轻松通信。

[查看或下载后端服务代码示例](#)

本机移动应用示例

本教程演示如何创建使用 ASP.NET Core MVC 支持本机移动应用的后端服务。它使用 [Xamarin Forms ToDoRest 应用](#) 作为其本机客户端，其中包括 Android、iOS、Windows Universal 和 Window Phone 设备的单独本机客户端。你可以遵循链接中的教程来创建本机应用程序（并安装需要的免费 Xamarin 工具），以及下载 Xamarin 示例解决方案。Xamarin 示例包含一个 ASP.NET Web API 2 服务项目，使用本文中的 ASP.NET Core 应用替换（客户端无需进行任何更改）。



功能

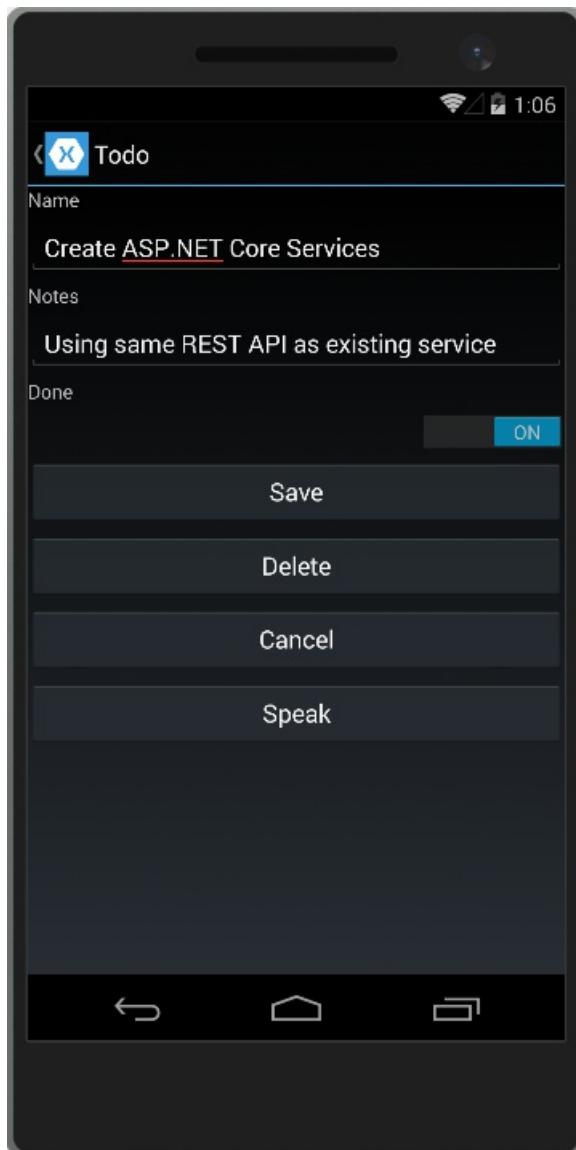
ToDoRest 应用支持列出、添加、删除和更新待办事项。每个项都有一个 ID、Name(名称)、Notes(说明)以及一个指示该项是否已完成的属性 Done。

待办事项的主视图如上所示，列出每个项的名称，并使用复选标记指示它是否已完成。

点击  图标打开“添加项”对话框：



点击主列表屏幕上的项将打开一个编辑对话框，在其中可以修改项的名称、说明以及是否完成，或删除项目：



此示例默认配置为使用托管在 developer.xamarin.com 上的后端服务，允许只读操作。若要使用在你计算机上运行的下一节创建的 ASP.NET Core 应用对其进行测试，你需要更新应用程序的 `RestUrl` 常量。导航到 `ToDoREST` 项目，然后打开 `Constants.cs` 文件。使用包含计算机 IP 的 URL 地址替换 `RestUrl`（不是 `localhost` 或 `127.0.0.1`，因为此地址用于从设备模拟器中，而不是从你的计算机中访问）。请包括端口号（`5000`）。为了测试你的服务能否在设备上正常运行，请确保没有活动的防火墙阻止访问此端口。

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

创建 ASP.NET Core 项目

在 Visual Studio 中创建一个新的 ASP.NET Core Web 应用程序。选择 Web API 模板和 No Authentication（无身份验证）。将项目命名为 `ToDoApi`。

Select a template:

ASP.NET Core Templates

Empty



Web API



Web Application

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET MVC Views and Controllers.

[Learn more](#)[Change Authentication](#)Authentication: **No Authentication**

Microsoft Azure

 Host in the cloud

App Service

OK

Cancel

对于向端口 5000 进行的请求，应用程序均需作出响应。更新 Program.cs，使其包含 `.UseUrls("http://*:5000")`，以便实现以下操作：

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

注意

请确保直接运行应用程序，而不是在 IIS Express 后运行，因为在默认情况下，后者会忽略非本地请求。从命令提示符处运行 `dotnet run`，或从 Visual Studio 工具栏中的“调试目标”下拉列表中选择应用程序名称配置文件。

添加一个模型类来表示待办事项。使用 `[Required]` 属性标记必需字段：

```
using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

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

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

        public bool Done { get; set; }
    }
}
```

API 方法需要通过某种方式处理数据。使用原始 Xamarin 示例所用的 `IToDoRepository` 接口：

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

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}
```

在此示例中，该实现仅使用一个专用项集合：

```
using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
        }
    }
}
```

```
public ToDoItem Find(string id)
{
    return _toDoList.FirstOrDefault(item => item.ID == id);
}

public void Insert(ToDoItem item)
{
    _toDoList.Add(item);
}

public void Update(ToDoItem item)
{
    var todoItem = this.Find(item.ID);
    var index = _toDoList.IndexOf(todoItem);
    _toDoList.RemoveAt(index);
    _toDoList.Insert(index, item);
}

public void Delete(string id)
{
    _toDoList.Remove(this.Find(id));
}

private void InitializeData()
{
    _toDoList = new List<ToDoItem>();

    var todoItem1 = new ToDoItem
    {
        ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
        Name = "Learn app development",
        Notes = "Attend Xamarin University",
        Done = true
    };

    var todoItem2 = new ToDoItem
    {
        ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
        Name = "Develop apps",
        Notes = "Use Xamarin Studio/Visual Studio",
        Done = false
    };

    var todoItem3 = new ToDoItem
    {
        ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
        Name = "Publish apps",
        Notes = "All app stores",
        Done = false,
    };

    _toDoList.Add(todoItem1);
    _toDoList.Add(todoItem2);
    _toDoList.Add(todoItem3);
}
}
```

在 *Startup.cs* 中配置该实现:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository, ToDoRepository>();
}
```

现可创建 `ToDoItemsController`。

提示

有关创建 Web API 的详细信息, 请参阅[使用 ASP.NET Core MVC 和 Visual Studio 生成首个 Web API](#)。

创建控制器

在项目中添加新控制器 `ToDoItemsController`。它应继承 `Microsoft.AspNetCore.Mvc.Controller`。添加 `[Route]` 属性以指示控制器将处理路径以 `api/todoitems` 开始的请求。路由中的 `[controller]` 标记会被控制器的名称代替(省略 `Controller` 后缀), 这对全局路由特别有用。详细了解 [路由](#)。

控制器需要 `IToDoRepository` 才能正常运行;通过控制器的构造函数请求该类型的实例。在运行时, 此实例将使用框架对 [依赖关系注入](#) 的支持来提供。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly IToDoRepository _ToDoRepository;

        public ToDoItemsController(IToDoRepository ToDoRepository)
        {
            _ToDoRepository = ToDoRepository;
        }
    }
}
```

此 API 支持四个不同的 HTTP 谓词来执行对数据源的 CRUD(创建、读取、更新、删除)操作。最简单的是读取操作, 它对应于 HTTP GET 请求。

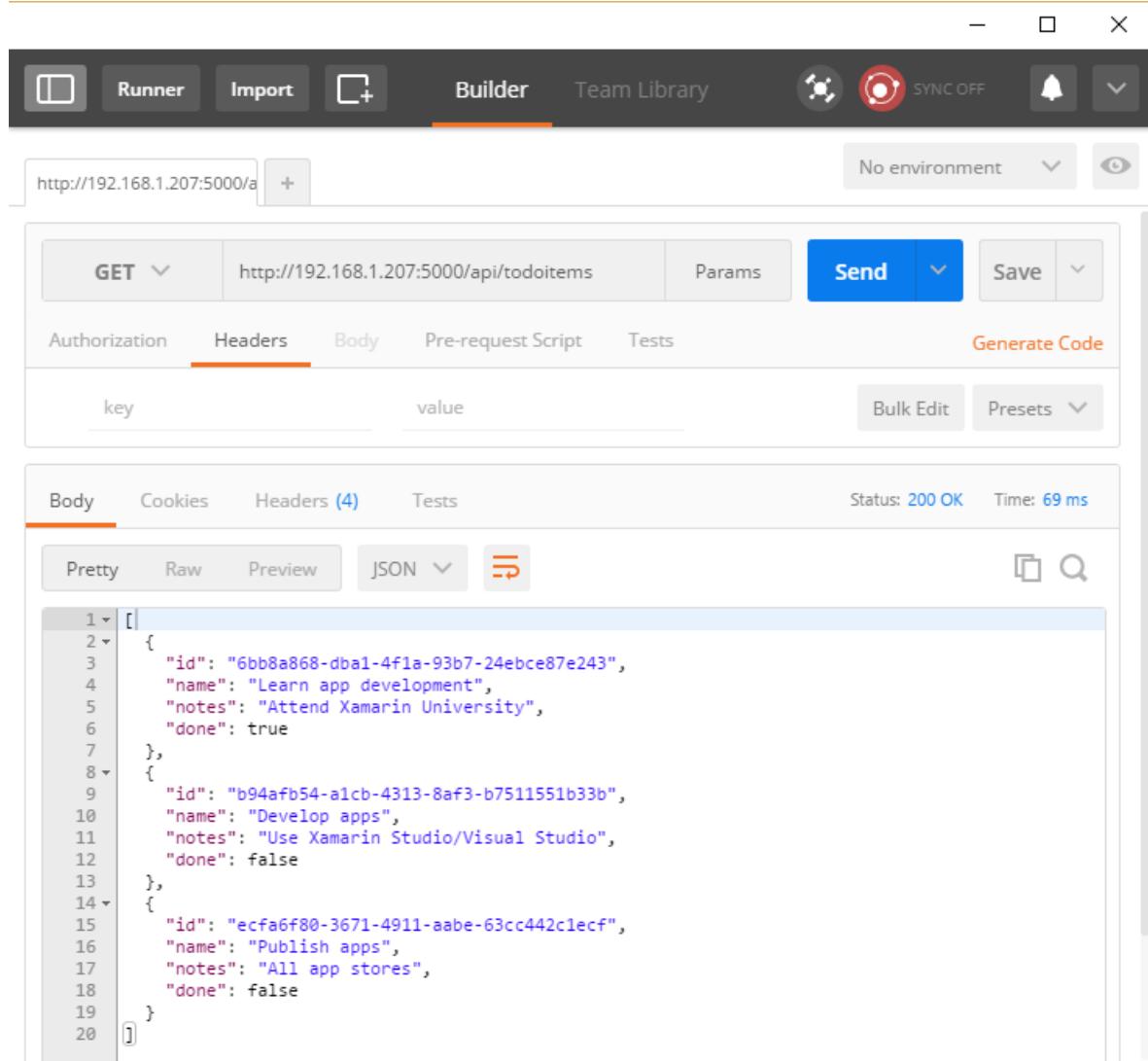
读取项目

要请求项列表, 可对 `List` 方法使用 GET 请求。`[HttpGet]` 方法的 `List` 属性指示此操作应仅处理 GET 请求。此操作的路由是在控制器上指定的路由。你不一定必须将操作名称用作路由的一部分。你只需确保每个操作都有唯一的和明确的路由。路由属性可以分别应用在控制器和方法级别, 以此生成特定的路由。

```
[HttpGet]
public IActionResult List()
{
    return Ok(_ToDoRepository.All);
}
```

`List` 方法返回 200 OK 响应代码和所有 ToDo 项, 并序列化为 JSON。

你可以使用多种工具测试新的 API 方法, 如 [Postman](#), 如此处所示:



The screenshot shows the Postman application window. At the top, there are tabs for Runner, Import, Builder (which is selected), and Team Library. On the right side, there are icons for Sync (OFF), a bell, and a dropdown menu. Below the tabs, the URL is set to <http://192.168.1.207:5000/>. The main interface shows a GET request to <http://192.168.1.207:5000/api/todoitems>. The Headers tab is selected, showing a key-value pair: 'key' and 'value'. The Body tab is also visible. The response status is 200 OK, and the time taken is 69 ms. The response body is displayed in JSON format, showing a list of three todo items:

```
1 [ 2 { 3   "id": "6bb8a868-dba1-4f1a-93b7-24ebce87e243", 4   "name": "Learn app development", 5   "notes": "Attend Xamarin University", 6   "done": true 7 }, 8 { 9   "id": "b94afb54-a1cb-4313-8af3-b7511551b33b", 10  "name": "Develop apps", 11  "notes": "Use Xamarin Studio/Visual Studio", 12  "done": false 13 }, 14 { 15   "id": "ecfa6f80-3671-4911-aabe-63cc442c1ecf", 16   "name": "Publish apps", 17   "notes": "All app stores", 18   "done": false 19 }]
```

创建项目

按照约定, 创建新数据项映射到 HTTP POST 谓词。Create 方法具有应用于该对象的 `[HttpPost]` 属性, 并接受 `ToDoItem` 实例。由于 `item` 参数将在 POST 的正文中传递, 因此该参数用 `[FromBody]` 属性修饰。

在该方法中, 会检查项的有效性和之前是否存在于数据存储, 并且如果没有任何问题, 则使用存储库添加。检查 `ModelState.IsValid` 将执行 模型验证, 应该在每个接受用户输入的 API 方法中执行此步骤。

```
[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _ToDoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TodoItemIDInUse.ToString());
        }
        _ToDoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

示例中使用一个枚举，后者包含传递到移动客户端的错误代码：

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

使用 Postman 测试添加新项，选择 POST 谓词并在请求正文中以 JSON 格式提供新对象。你还应添加一个请求标头指定 `Content-Type` 为 `application/json`。

The screenshot shows the Postman interface. At the top, there are tabs for Runner, Import, Builder (which is selected), Team Library, and various status indicators. Below the tabs, the URL is set to `http://192.168.1.207:5000/a`. The main area shows a POST request to `http://192.168.1.207:5000/api/todoitems`. The Body tab is selected, showing a raw JSON payload:

```

1 {
2   "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
3   "Name": "A Test Item",
4   "Notes": "asdf",
5   "Done": false
6 }

```

The response section shows a 200 OK status with a time of 227 ms. The Body tab is selected, displaying the same JSON object with the ID updated to a new GUID.

该方法返回在响应中新建的项。

更新项目

通过使用 HTTP PUT 请求来修改记录。除了此更改之外, `Edit` 方法几乎与 `Create` 完全相同。请注意, 如果未找到记录, 则 `Edit` 操作将返回 `NotFound` (404) 响应。

```

[HttpPut]
public IActionResult Edit([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _ToDoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _ToDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}

```

若要使用 Postman 进行测试，将谓词更改为 PUT。在请求正文中指定要更新的对象数据。

The screenshot shows the Postman application window. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), 'Team Library', and various status indicators like 'SYNC OFF'. The main area has a URL input field containing 'http://192.168.1.207:5000/api/todolist' and a 'Send' button. Below the URL, there's a table for parameters with columns 'key' and 'value'. Under the 'Body' tab, the 'raw' option is selected, and the JSON payload is:

```
1 {  
2     "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",  
3     "Name": "An UPDATED Test Item",  
4     "Notes": "Some updated notes",  
5     "Done": true  
6 }
```

Below the body, the response section shows a status of 204 No Content and a time of 91 ms. The 'Pretty' view of the response is empty.

为了与预先存在的 API 保持一致，此方法在成功时返回 `NoContent` (204) 响应。

删除项目

删除记录可以通过向服务发出 DELETE 请求并传递要删除项的 ID 来完成。与更新一样，请求的项不存在时会收到 `NotFound` 响应。请求成功会得到 `NoContent` (204) 响应。

```
[HttpDelete("{id}")]  
public IActionResult Delete(string id)  
{  
    try  
    {  
        var item = _todoRepository.Find(id);  
        if (item == null)  
        {  
            return NotFound(ErrorCode.RecordNotFound.ToString());  
        }  
        _todoRepository.Delete(id);  
    }  
    catch (Exception)  
    {  
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());  
    }  
    return NoContent();  
}
```

请注意，在测试删除功能时，请求正文中不需要任何内容。

The screenshot shows the Postman application interface. At the top, there are tabs for Runner, Import, Builder (which is selected), and Team Library. There are also icons for sync, notifications, and a dropdown menu. The URL bar shows 'http://192.168.1.207:5000/a'. Below the URL bar, there are buttons for 'Send' and 'Save'. The main area shows a 'DELETE' request to 'http://192.168.1.207:5000/api/todoitems/6bb8b8'. The 'Body' tab is selected, showing options for form-data, x-www-form-urlencoded, raw, binary, and JSON (application/json). The 'JSON (application/json)' option is selected. The 'Body' section contains a single item with key '1'. Below this, the response section shows 'Status: 204 No Content' and 'Time: 91 ms'. The 'Body' tab is selected here as well, with options for Pretty, Raw, Preview, and HTML.

常见的 Web API 约定

开发应用程序的后端服务时，你将想要使用一组一致的约定或策略来处理横切关注点。例如，在上面所示服务中，针对不存在的特定记录的请求会收到 `NotFound` 响应，而不是 `BadRequest` 响应。同样，对于此服务，传递模型绑定类型的命令始终检查 `ModelState.IsValid` 并为无效的模型类型返回 `BadRequest`。

一旦为 API 指定通用策略，一般可以将其封装在 `Filter(筛选器)`。详细了解 [如何封装 ASP.NET Core MVC 应用程序中的通用 API 策略](#)。

带有 Swagger / 开放 API 的 ASP.NET Core Web API 帮助页

2018/5/14 • 3 min to read • [Edit Online](#)

作者: [Christoph Nienaber](#) 和 [Rico Suter](#)

使用 Web API 时, 了解其各种方法对开发人员来说可能是一项挑战。[Swagger](#) 也称为开放 API, 解决了为 Web API 生成有用文档和帮助页的问题。它具有诸如交互式文档、客户端 SDK 生成和 API 可发现性等优点。

本文展示了 [Swashbuckle.AspNetCore](#) 和 [NSwag](#) .NET Swagger 实现:

- **Swashbuckle.AspNetCore** 是一个开源项目, 用于生成 ASP.NET Core Web API 的 Swagger 文档。
- **NSwag** 是另一个用于将 [Swagger UI](#) 或 [ReDoc](#) 集成到 ASP.NET Core Web API 中的开源项目。它提供了为 API 生成 C# 和 TypeScript 客户端代码的方法。

什么是 Swagger / 开放 API?

Swagger 是一个与语言无关的规范, 用于描述 REST API。Swagger 项目已捐赠给 [OpenAPI 计划](#), 现在它被称为开放 API。这两个名称可互换使用, 但开放 API 是首选。它允许计算机和人员了解服务的功能, 而无需直接访问实现(源代码、网络访问、文档)。其中一个目标是尽量减少连接取消关联的服务所需的工作量。另一个目标是减少准确记录服务所需的时间。

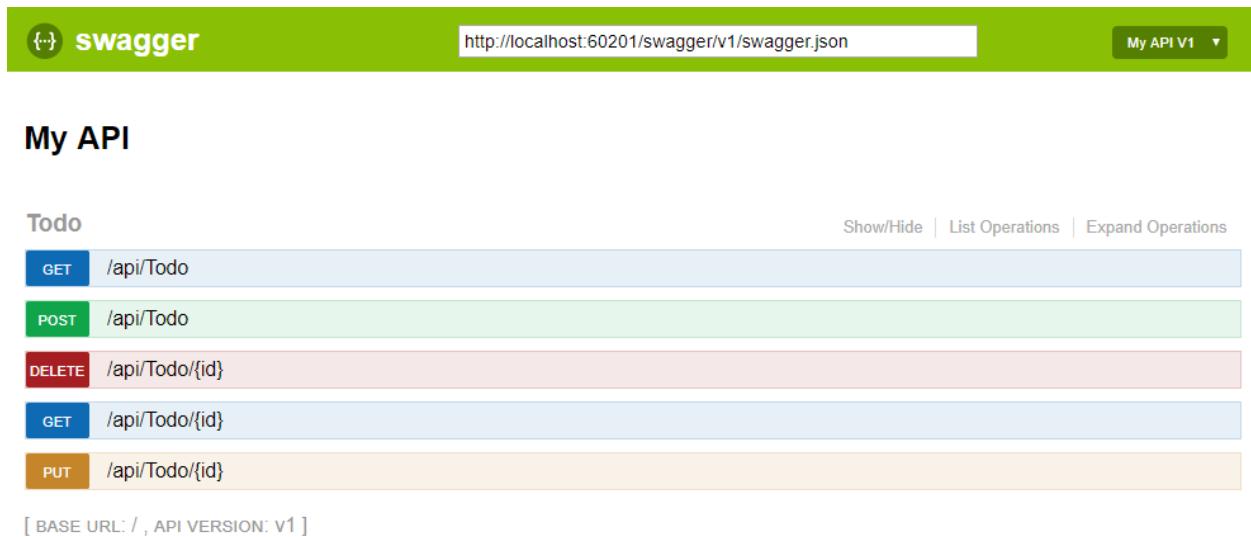
Swagger 规范 (swagger.json)

Swagger 流的核心是 Swagger 规范, 默认情况下是名为 swagger.json 的文档。它由 Swagger 工具链(或其第三方实现)根据你的服务生成。它描述了 API 的功能以及使用 HTTP 对其进行访问的方式。它驱动 Swagger UI, 并由工具链用来启用发现和客户端代码生成。下面是为简洁起见而缩减的 Swagger 规范的示例:

```
{
    "swagger": "2.0",
    "info": {
        "version": "v1",
        "title": "API V1"
    },
    "basePath": "/",
    "paths": {
        "/api/Todo": {
            "get": {
                "tags": [
                    "Todo"
                ],
                "operationId": "ApiTodoGet",
                "consumes": [],
                "produces": [
                    "text/plain",
                    "application/json",
                    "text/json"
                ],
                "responses": {
                    "200": {
                        "description": "Success",
                        "schema": {
                            "type": "array",
                            "items": {
                                "$ref": "#/definitions/TodoItem"
                            }
                        }
                    }
                }
            },
            "post": {
                ...
            }
        },
        "/api/Todo/{id)": {
            "get": {
                ...
            },
            "put": {
                ...
            },
            "delete": {
                ...
            },
            "definitions": {
                "TodoItem": {
                    "type": "object",
                    "properties": {
                        "id": {
                            "format": "int64",
                            "type": "integer"
                        },
                        "name": {
                            "type": "string"
                        },
                        "isComplete": {
                            "default": false,
                            "type": "boolean"
                        }
                    }
                }
            }
        },
        "securityDefinitions": {}
    }
}
```

Swagger UI

Swagger UI 提供了基于 Web 的 UI, 它使用生成的 Swagger 规范提供有关服务的信息。Swashbuckle 和 NSwag 均包含 Swagger UI 的嵌入式版本, 因此可使用中间件注册调用将该嵌入式版本托管在 ASP.NET Core 应用中。Web UI 如下所示:



The screenshot shows the Swagger UI interface for a Todo API. At the top, there's a green header bar with the title "swagger" and a URL input field containing "http://localhost:60201/swagger/v1/swagger.json". On the right of the header is a dropdown menu labeled "My API V1". Below the header, the main content area has a title "My API" followed by "Todo". To the right of the title are three buttons: "Show/Hide", "List Operations", and "Expand Operations". The main content area lists five API operations under the "Todo" category:

- GET /api/Todo** (blue button)
- POST /api/Todo** (green button)
- DELETE /api/Todo/{id}** (red button)
- GET /api/Todo/{id}** (blue button)
- PUT /api/Todo/{id}** (orange button)

At the bottom left, there's a note: "[BASE URL: / , API VERSION: V1]".

控制器中的每个公共操作方法都可以从 UI 中进行测试。单击方法名称可以展开该部分。添加所有必要的参数, 然后单击“试试看!”。

GET /api/Todo

Response Class (Status 200)

Success

[Model](#) [Example Value](#)

```
[  
  {  
    "id": 0,  
    "name": "string",  
    "isComplete": false  
  }  
]
```

Response Content Type [text/plain](#)[Try it out!](#) [Hide Response](#)

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:60201/api/Todo'
```

Request URL

```
http://localhost:60201/api/Todo
```

Response Body

```
[  
  {  
    "id": 1,  
    "name": "Item1",  
    "isComplete": false  
  }  
]
```

Response Code

```
200
```

Response Headers

```
{  
  "date": "Thu, 31 Aug 2017 17:29:04 GMT",  
  "server": "Kestrel",  
  "transfer-encoding": "chunked",  
  "content-type": "application/json; charset=utf-8"  
}
```

注意

用于屏幕截图的 Swagger UI 版本是版本 2。有关版本 3 的示例, 请参阅 [Petstore 示例](#)。

后续步骤

- [Swashbuckle 入门](#)
- [NSwag 入门](#)

NSwag 和 ASP.NET Core 入门

2018/5/14 • 5 min to read • [Edit Online](#)

作者: [Christoph Nienaber](#) 和 [Rico Suter](#)

在 ASP.NET Core 中间件中使用 [NSwag](#) 需要 [NSwag.AspNetCore](#) NuGet 包。该包由 Swagger 生成器、Swagger UI(v2 和 v3)和 [ReDoc UI](#) 组成。

强烈建议使用 NSwag 的代码生成功能。请为代码生成选择下列选项之一：

- 使用 [NSwagStudio](#), 这是一款 Windows 桌面应用, 用于在 C# 和 TypeScript 中为 API 生成客户端代码
- 使用 [NSwag.CodeGeneration.CSharp](#) 或 [NSwag.CodeGeneration.TypeScript](#) NuGet 包在项目中执行代码生成
- 使用[命令行](#)中的 NSwag
- 使用 [NSwag.MSBuild](#) NuGet 包

功能

使用 NSwag 的主要原因是它不仅能引入 Swagger UI 和 Swagger 生成器, 还能利用灵活的代码生成功能。无需现有 API — 可使用包含 Swagger 的第三方 API 并让 NSwag 生成客户端实现。无论哪种方式, 开发周期都会加快, 可以更轻松地适应 API 更改。

包安装

可使用以下方法添加 NuGet 包:

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)
- 从“程序包管理器控制台”窗口:

```
Install-Package NSwag.AspNetCore
```

- 从“管理 NuGet 程序包”对话框中:
 - 右键单击“解决方案资源管理器”>“管理 NuGet 包”中的项目
 - 将“包源”设置为“nuget.org”
 - 在搜索框中输入“NSwag.AspNetCore”
 - 从“浏览”选项卡中选择“NSwag.AspNetCore”包, 然后单击“安装”

添加并配置 Swagger 中间件

在 `Info` 类中导入以下命名空间:

```
using NSwag.AspNetCore;
using System.Reflection;
using Newtonsoft.Json;
```

在 `Startup.Configure` 方法中, 启用中间件为生成的 Swagger 规范和 Swagger UI 提供服务:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable the Swagger UI middleware and the Swagger generator
    app.UseSwaggerUi(typeof(Startup).GetTypeInfo().Assembly, settings =>
    {
        settings.GeneratorSettings.DefaultPropertyNameHandling = PropertyNameHandling.CamelCase;
    });

    app.UseMvc();
}
```

启动应用。导航到 [/swagger](#) 查看 Swagger UI。导航到 [/swagger/v1/swagger.json](#) 查看 Swagger 规范。

代码生成

通过 NSwagStudio

- 从官方 [GitHub 存储库](#) 安装 [NSwagStudio](#)。
- 启动 NSwagStudio。输入 swagger.json 的位置或直接复制它：



- 指示所需的客户端输出类型。选项包括“TypeScript 客户端”、“CSharp 客户端”或“CSharp Web API 控制器”。使用 Web API 控制器基本上是一种反向生成。它使用服务的规范来重新生成服务。
- 单击“生成输出”。
- 可在此处看到 C# 中的 TodoApi.NSwag 示例的完整客户端实现：



- 将该文件放入客户端项目（例如，[Xamarin.Forms](#) 应用）。开始使用 API：

```
var todoClient = new TodoClient();

// Gets all Todos from the Api
var allTodos = await todoClient.GetAllAsync();

// Create a new TodoItem and save it in the Api
var createdTodo = await todoClient.CreateAsync(new TodoItem());

// Get a single Todo by Id
var foundTodo = await todoClient.GetByIdAsync(1);
```

注意

可将基 URL 和/或 HTTP 客户端注入 API 客户端。最佳做法是始终 [重复使用 HttpClient](#)。

现在可开始轻松地将 API 实施到客户端项目中。

生成客户端代码的其他方法

可通过其他更适合你的工作流的方式生成代码：

- [MSBuild](#)

- 在代码中生成
- 通过 T4 模板生成

自定义

XML 注释

可使用以下方法启用 XML 注释：

- [Visual Studio](#)
 - [Visual Studio for Mac](#)
 - [Visual Studio Code](#)
- 右键单击“解决方案资源管理器”中的项目，然后选择“属性”
 - 查看“生成”选项卡的“输出”部分下的“XML 文档文件”框：



数据注释

NSwag 使用[反射](#)，Web API 操作的最佳做法是返回 [IActionResult](#)。因此，NSwag 无法推断正在执行的操作和返回的结果。请看下面的示例：

```
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

上述操作返回 `IActionResult`，但在操作内部返回 `CreatedAtRoute` 或 `BadRequest`。使用数据注释告知客户端此操作返回的 HTTP 响应。使用以下属性修饰该操作：

```
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)] // Created
[ProducesResponseType(typeof(TodoItem), 400)] // BadRequest
public IActionResult Create([FromBody] TodoItem item)
```

Swagger 生成器现在可准确地描述此操作，且生成的客户端知道调用终结点时收到的内容。强烈建议使用这些属性来修饰所有操作。有关 API 操作应返回的 HTTP 响应的指导原则，请参阅 [RFC 7231 规范](#)。

Swashbuckle 和 ASP.NET Core 入门

2018/5/14 • 10 min to read • [Edit Online](#)

作者:Shayne Boyer 和 Scott Addie

Swashbuckle 有三个主要组成部分:

- [Swashbuckle.AspNetCore.Swagger](#): 将 `SwaggerDocument` 对象公开为 JSON 终结点的 Swagger 对象模型和中间件。
- [Swashbuckle.AspNetCore.SwaggerGen](#): 从路由、控制器和模型直接生成 `SwaggerDocument` 对象的 Swagger 生成器。它通常与 Swagger 终结点中间件结合, 以自动公开 Swagger JSON。
- [Swashbuckle.AspNetCore.SwaggerUI](#): Swagger UI 工具的嵌入式版本。它解释 Swagger JSON 以构建描述 Web API 功能的可自定义的丰富体验。它包括针对公共方法的内置测试工具。

包安装

可以使用以下方法来添加 Swashbuckle:

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)
- 从“程序包管理器控制台”窗口:

```
Install-Package Swashbuckle.AspNetCore
```

- 从“管理 NuGet 程序包”对话框中:

- 右键单击“解决方案资源管理器”>“管理 NuGet 包”中的项目
- 将“包源”设置为“nuget.org”
- 在搜索框中输入“Swashbuckle.AspNetCore”
- 从“浏览”选项卡中选择“Swashbuckle.AspNetCore”包, 然后单击“安装”

添加并配置 Swagger 中间件

将 Swagger 生成器添加到 `Startup.ConfigureServices` 方法中的服务集合中:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining one or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}
```

导入以下命名空间以使用 `Info` 类：

```
using Swashbuckle.AspNetCore.Swagger;
```

在 `Startup.Configure` 方法中，启用中间件为生成的 JSON 文档和 Swagger UI 提供服务：

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.), specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

启动应用，并导航到 `http://localhost:<random_port>/swagger/v1/swagger.json`。生成的描述终结点的文档显示在 [Swagger 规范 \(swagger.json\)](#) 中。

可在 `http://localhost:<random_port>/swagger` 找到 Swagger UI。通过 Swagger UI 浏览 API，并将其合并其他计划中。

提示

要在应用的根 (`http://localhost:<random_port>/`) 处提供 Swagger UI，请将 `RoutePrefix` 属性设置为空字符串：

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    c.RoutePrefix = string.Empty;
});
```

自定义和扩展

Swagger 提供了为对象模型进行归档和自定义 UI 以匹配你的主题的选项。

API 信息和说明

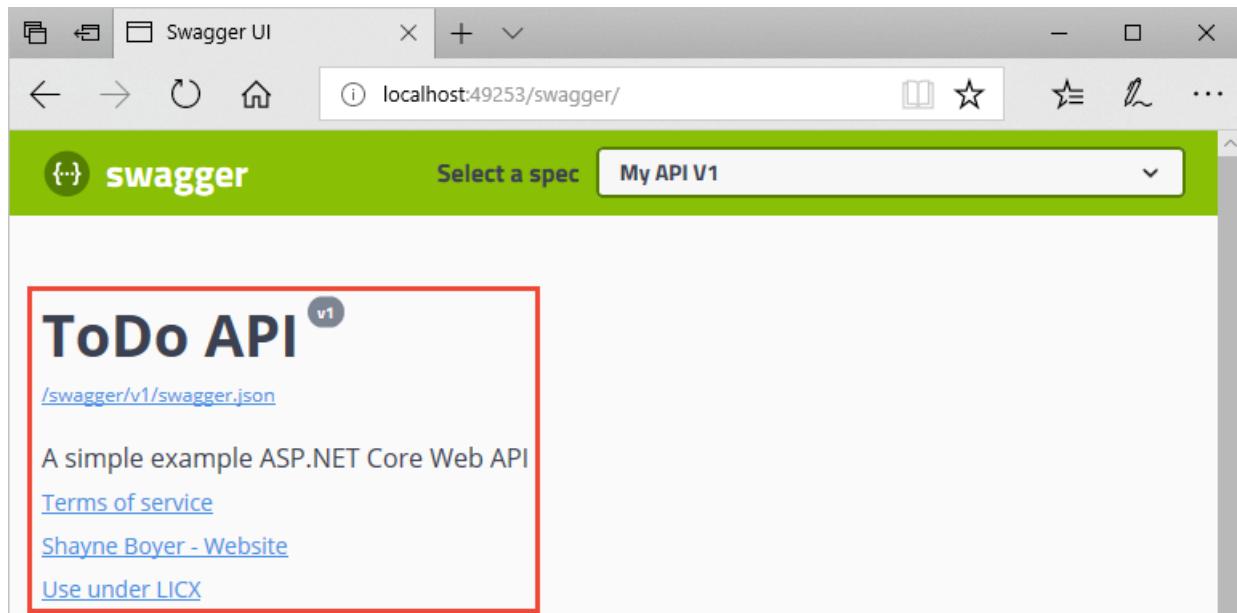
传递给 `AddSwaggerGen` 方法的配置操作会添加诸如作者、许可证和说明的信息：

```

services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = "None",
        Contact = new Contact
        {
            Name = "Shayne Boyer",
            Email = string.Empty,
            Url = "https://twitter.com/spboyer"
        },
        License = new License
        {
            Name = "Use under LICX",
            Url = "https://example.com/license"
        }
    });
});

```

Swagger UI 显示版本的信息：



XML 注释

可使用以下方法启用 XML 注释：

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- 右键单击“解决方案资源管理器”中的项目，然后选择“属性”
- 查看“生成”选项卡的“输出”部分下的“XML 文档文件”框：

启用 XML 注释，为未记录的公共类型和成员提供调试信息。警告消息指示未记录的类型和成员。例如，以下消息指示违反警告代码 1591：

```
warning CS1591: Missing XML comment for publicly visible type or member 'TodoController.GetAll()'
```

定义要在 .csproj 文件中忽略的以分号分隔的警告代码列表，以取消警告：

```
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'>
<DocumentationFile>bin\Debug\$(TargetFramework)\$(MSBuildProjectName).xml</DocumentationFile>
<NoWarn>1701;1702;1705;1591</NoWarn>
</PropertyGroup>
```

配置 Swagger 以使用生成的 XML 文件。对于 Linux 或非 Windows 操作系统，文件名和路径区分大小写。例如，“TodoApi.XML”文件在 Windows 上有效，但在 CentOS 上无效。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining one or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = "None",
            Contact = new Contact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = "https://twitter.com/spboyer"
            },
            License = new License
            {
                Name = "Use under LICX",
                Url = "https://example.com/license"
            }
        });
    });

    // Set the comments path for the Swagger JSON and UI.
    var xmlFile = $"{Assembly.GetEntryAssembly().GetName().Name}.xml";
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
});
}
```

在上述代码中，[反射](#)用于生成与 Web API 项目相匹配的 XML 文件名。此方法可确保生成的 XML 文件名与项目名称匹配。[AppContext.BaseDirectory](#)属性用于构造 XML 文件的路径。

通过向节标题添加说明，将三斜杠注释添加到操作增强了 Swagger UI。执行 [Delete](#) 操作前添加 [`<summary>`](#) 元素：

```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);

    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();

    return new NoContentResult();
}
```

Swagger UI 显示上述代码的 `<summary>` 元素的内部文本：

The screenshot shows the Swagger UI interface for a DELETE API endpoint. The endpoint is defined as `/api/Todo/{id}` with the description "Deletes a specific TodoItem.". The parameters section shows a required parameter `id` of type integer (path). The responses section shows a successful response (200) with the description "Success". A "Try it out" button is also visible.

Name	Description
<code>id</code> * required integer (path)	

Responses	Response content type
200	application/json

Code	Description
200	Success

生成的 JSON 架构驱动 UI：

```

"delete": {
    "tags": [
        "Todo"
    ],
    "summary": "Deletes a specific TodoItem.",
    "operationId": "ApiTodoByIdDelete",
    "consumes": [],
    "produces": [],
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "",
            "required": true,
            "type": "integer",
            "format": "int64"
        }
    ],
    "responses": {
        "200": {
            "description": "Success"
        }
    }
}

```

将 `<remarks>` 元素添加到 `Create` 操作方法文档。它可以补充 `<summary>` 元素中指定的信息，并提供更可靠的信息。Swagger UI。`<remarks>` 元素内容可包含文本、JSON 或 XML。

```

/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///         "id": 1,
///         "name": "Item1",
///         "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}

```

请注意带这些附加注释的 UI 增强功能：

POST

/api/Todo Creates a TodoItem.

Sample request:

```
POST /Todo
{
    "id": 1,
    "name": "Item1",
    "isComplete": true
}
```

数据注释

使用 [System.ComponentModel.DataAnnotations](#) 命名空间中的属性来修饰模型，以帮助驱动 Swagger UI 组件。

将 `[Required]` 属性添加到 `TodoItem` 类的 `Name` 属性：

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

namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }

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

        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}
```

此属性的状态更改 UI 行为并更改基础 JSON 架构：

```
"definitions": {
    "TodoItem": {
        "required": [
            "name"
        ],
        "type": "object",
        "properties": {
            "id": {
                "format": "int64",
                "type": "integer"
            },
            "name": {
                "type": "string"
            },
            "isComplete": {
                "default": false,
                "type": "boolean"
            }
        }
    }
},
```

将 `[Produces("application/json")]` 属性添加到 API 控制器。这样做的目的是声明控制器的操作支持 `application/json` 的响应内容类型：

```
namespace TodoApi.Controllers
{
    [Produces("application/json")]
    [Route("api/[controller]")]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;
```

“响应内容类型”下拉列表选此内容类型作为控制器的默认 GET 操作：

The screenshot shows the Swagger UI interface for a `GET /api/Todo` operation. In the `Responses` section, there is a dropdown menu labeled "Response content type" with the value "application/json" selected. This dropdown is highlighted with a red box.

随着 Web API 中的数据注释的使用越来越多, UI 和 API 帮助页变得更具说明性和更为有用。

描述响应类型

消费应用程序开发人员最关心的问题是返回的内容 — 具体的响应类型和错误代码(如果不标准)。在 XML 注释和数据注释中表示响应类型和错误代码。

`Create` 操作成功后返回 HTTP 201 状态代码。发布的请求正文为 NULL 时, 将返回 HTTP 400 状态代码。如果 Swagger UI 中没有提供合适的文档, 那么使用者会缺少对这些预期结果的了解。在以下示例中, 通过添加突出显示的行解决此问题：

```

///<summary>
/// Creates a TodoItem.
///</summary>
///<remarks>
/// Sample request:
///
/// POST /Todo
/// {
///     "id": 1,
///     "name": "Item1",
///     "isComplete": true
/// }
///
///</remarks>
///<param name="item"></param>
///<returns>A newly created TodoItem</returns>
///<response code="201">Returns the newly created item</response>
///<response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), 201)]
[ProducesResponseType(400)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}

```

Swagger UI 现在清楚地记录预期的 HTTP 响应代码：

The screenshot shows the 'Responses' section of the Swagger UI. At the top, it says 'Responses' and 'Response content type application/json'. Below that, there are two rows for 'Code' and 'Description'.

- Code:** 201
- Description:** Returns the newly created item
- Code:** 400
- Description:** If the item is null

Both the 'Description' and '400' sections are highlighted with a red border. In the 'Description' section, there is also a red border around the entire row. Below the descriptions, there are 'Example Value' and 'Model' buttons, and a JSON snippet for the 201 response.

自定义 UI

股票 UI 既实用又可呈现。但是，API 文档页应代表品牌或主题。将 Swashbuckle 组件标记为需要添加资源以提供静态文件，并构建文件夹结构以托管这些文件。

如果以 .NET Framework 或 .NET Core 1.x 为目标，请将 [Microsoft.AspNetCore.StaticFiles](#) NuGet 包添加到项目：

```
<PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
```

如果以 .NET Core 2.x 为目标并使用[元包](#)，则已安装上述 NuGet 包。

启用静态文件中间件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.), specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

从 [Swagger UI GitHub 存储库](#) 中获取 dist 文件夹的内容。此文件夹包含 Swagger UI 页必需的资产。

创建 wwwroot/swagger/ui 文件夹，然后将 dist 文件夹的内容复制到其中。

使用以下 CSS 在 wwwroot/swagger/ui 中创建 custom.css 文件，以自定义页面标题：

```
.swagger-ui .topbar {
    background-color: #000;
    border-bottom: 3px solid #547f00;
}
```

引用其他 CSS 文件后，引用 index.html 文件中的“custom.css”：

```
<link href="https://fonts.googleapis.com/css?family=Open+Sans:400,700|Source+Code+Pro:300,600|Titillium+Web:400,600,700" rel="stylesheet">
<link rel="stylesheet" type="text/css" href="./swagger-ui.css">
<link rel="stylesheet" type="text/css" href="custom.css">
```

浏览到 `http://localhost:<random_port>/swagger/ui/index.html` 中的 index.html 页。在标题文本框中输入 `http://localhost:<random_port>/swagger/v1/swagger.json`，然后单击“浏览”按钮。生成的页面如下所示：

ToDo API v1

<http://localhost:49253/swagger/v1/swagger.json>

A simple example ASP.NET Core Web API

[Terms of service](#)

[Shayne Boyer - Website](#)

[Use under LICX](#)

Todo

GET /api/Todo

POST /api/Todo Creates a TodoItem.

GET /api/Todo/{id}

PUT /api/Todo/{id}

DELETE /api/Todo/{id} Deletes a specific TodoItem.

Models

TodoItem >

还可以对页面执行更多操作。在 [Swagger UI GitHub 存储库](#) 中查看 UI 资源的完整功能。

在 ASP.NET Core 中使用数据

2018/4/10 • 1 min to read • [Edit Online](#)

- [通过 Visual Studio 开始使用 Razor 页面和 Entity Framework Core](#)
 - [Razor 页面及 EF 入门](#)
 - [创建、读取、更新和删除操作](#)
 - [排序、筛选器、页和组](#)
 - [迁移](#)
 - [创建复杂数据模型](#)
 - [读取相关数据](#)
 - [更新相关数据](#)
 - [处理并发冲突](#)
- [ASP.NET Core MVC 和 Entity Framework Core 入门\(使用 Visual Studio\)](#)
 - [入门](#)
 - [创建、读取、更新和删除操作](#)
 - [排序、筛选器、页和组](#)
 - [迁移](#)
 - [创建复杂数据模型](#)
 - [读取相关数据](#)
 - [更新相关数据](#)
 - [处理并发冲突](#)
 - [继承](#)
 - [高级主题](#)
- [ASP.NET Core 和 EF Core - 新数据库](#)(Entity Framework Core 文档站点)
- [ASP.NET Core 和 EF Core - 现有数据库](#)(Entity Framework Core 文档站点)
- [开始使用 ASP.NET Core 和 Entity Framework 6](#)
- [Azure 存储](#)
 - [使用 Visual Studio 连接服务添加 Azure 存储](#)
 - [开始使用 Azure Blob 存储和 Visual Studio 连接服务](#)
 - [开始使用队列存储和 Visual Studio 连接服务](#)
 - [开始使用 Azure 表存储和 Visual Studio 连接服务](#)

ASP.NET Core Razor 页面和 EF Core - 教程系列

2018/4/10 • 1 min to read • [Edit Online](#)

下面的一系列教程介绍如何创建使用 Entity Framework (EF) Core 进行数据访问的 ASP.NET Core Razor 页面 Web 应用。这些教程需要 Visual Studio 2017。

1. [入门](#)
2. [创建、读取、更新和删除操作](#)
3. [排序、筛选、分页和分组](#)
4. [迁移](#)
5. [创建复杂数据模型](#)
6. [读取相关数据](#)
7. [更新相关数据](#)
8. [处理并发冲突](#)

ASP.NET Core 中的 Razor 页面和 Entity Framework Core - 第 1 个教程(共 8 个)

2018/5/14 • 21 min to read • [Edit Online](#)

作者: [Tom Dykstra](#) 和 [Rick Anderson](#)

Contoso University 示例 Web 应用演示了如何使用 Entity Framework (EF) Core 2.0 和 Visual Studio 2017 创建 ASP.NET Core 2.0 MVC Web 应用程序。

该示例应用是一个虚构的 Contoso University 的网站。其中包括学生录取、课程创建和讲师分配等功能。本页是介绍如何构建 Contoso University 示例应用系列教程中的第一部分。

[下载或查看已完成的应用。下载说明。](#)

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

熟悉 [Razor 页面](#)。新程序员在开始学习本系列之前，应先完成 [Razor 页面入门](#)。

疑难解答

如果遇到无法解决的问题，可以通过与[已完成的阶段](#)对比代码来查找解决方案。常见错误以及对应的解决方案，请参阅[最新教程中的故障排除](#)。如果在该处找不到所需内容，可以在[StackOverflow.com](#) 上发表有关 [ASP.NET Core](#) 或 [EF Core](#) 的问题。

提示

本系列教程以之前教程中已完成的工作为基础。每次成功完成教程后，请考虑保存项目副本。如果遇到问题，便可以从上一教程重新开始，而无需从头开始。或者可以下载[已完成阶段](#)并使用已完成阶段重新开始。

Contoso University Web 应用

这些教程中所构建的应用是一个基本的大学网站。

用户可以查看和更新学生、课程和讲师信息。以下是在本教程中创建的几个屏幕。

The screenshot shows a web browser window titled "Index - Contoso University". The address bar displays "localhost:1234/Students". The main content area is titled "Contoso University" and features a large "Index" heading. Below it is a "Create New" link and a search bar with a placeholder "Find by name:" and a "Search" button. A horizontal table header row contains "Last Name", "First Name", and "Enrollment Date". Three student records are listed:

Last Name	First Name	Enrollment Date	
Smith	Joe	2017-10-31	Edit Details Delete
Alexander	Carson	2010-09-01	Edit Details Delete
Alonso	Meredith	2012-09-01	Edit Details Delete

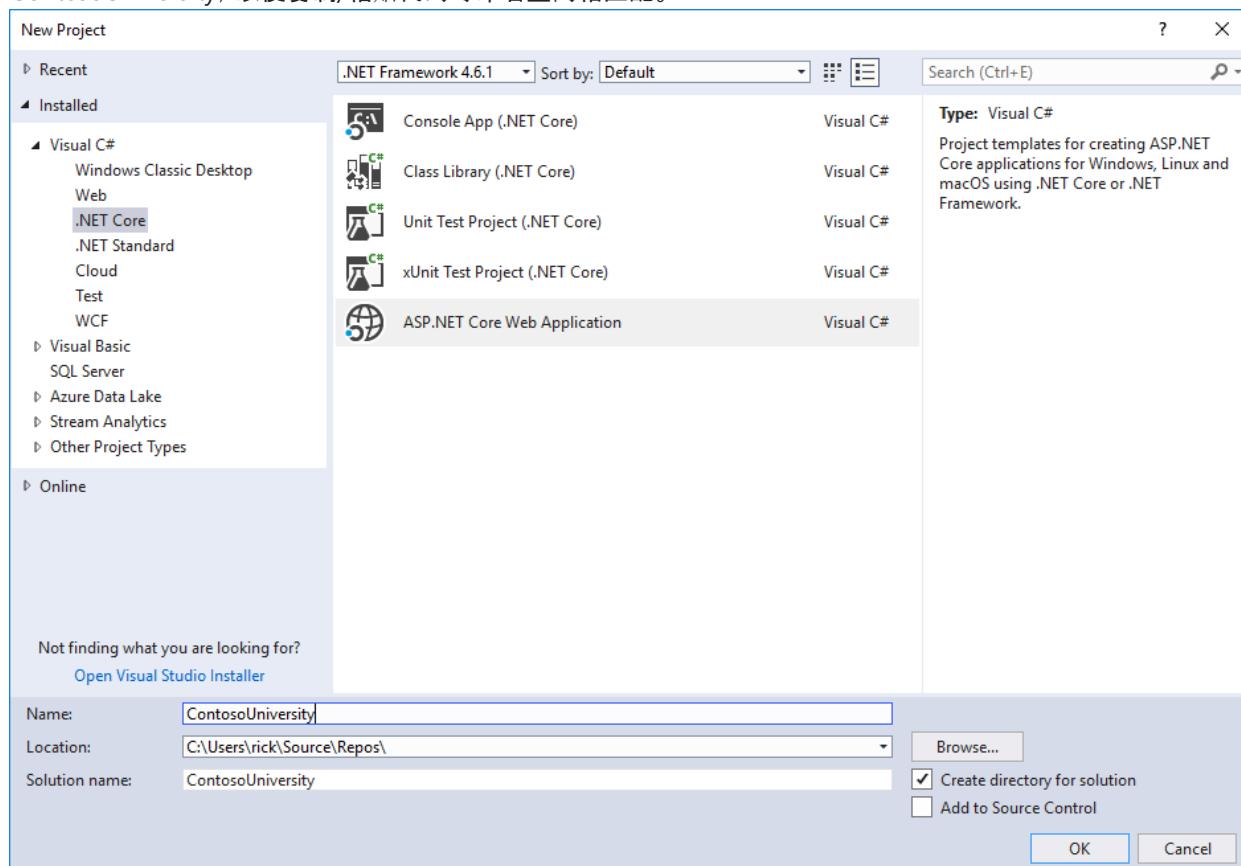
Below the table are "Previous" and "Next" navigation buttons. At the bottom of the page is a copyright notice: "© 2017 - Contoso University".

The screenshot shows a web browser window titled "Edit - Contoso University". The address bar displays "localhost:1234/Students/Edit/1". The main content area is titled "Contoso University" and features a large "Edit" heading. Below it is a "Student" title. The page contains three form fields: "Last Name" (value: Alexander), "First Name" (value: Carson), and "Enrollment Date" (value: 09/01/2010). A "Save" button is located at the bottom left. At the bottom of the page is a copyright notice: "© 2017 - Contoso University".

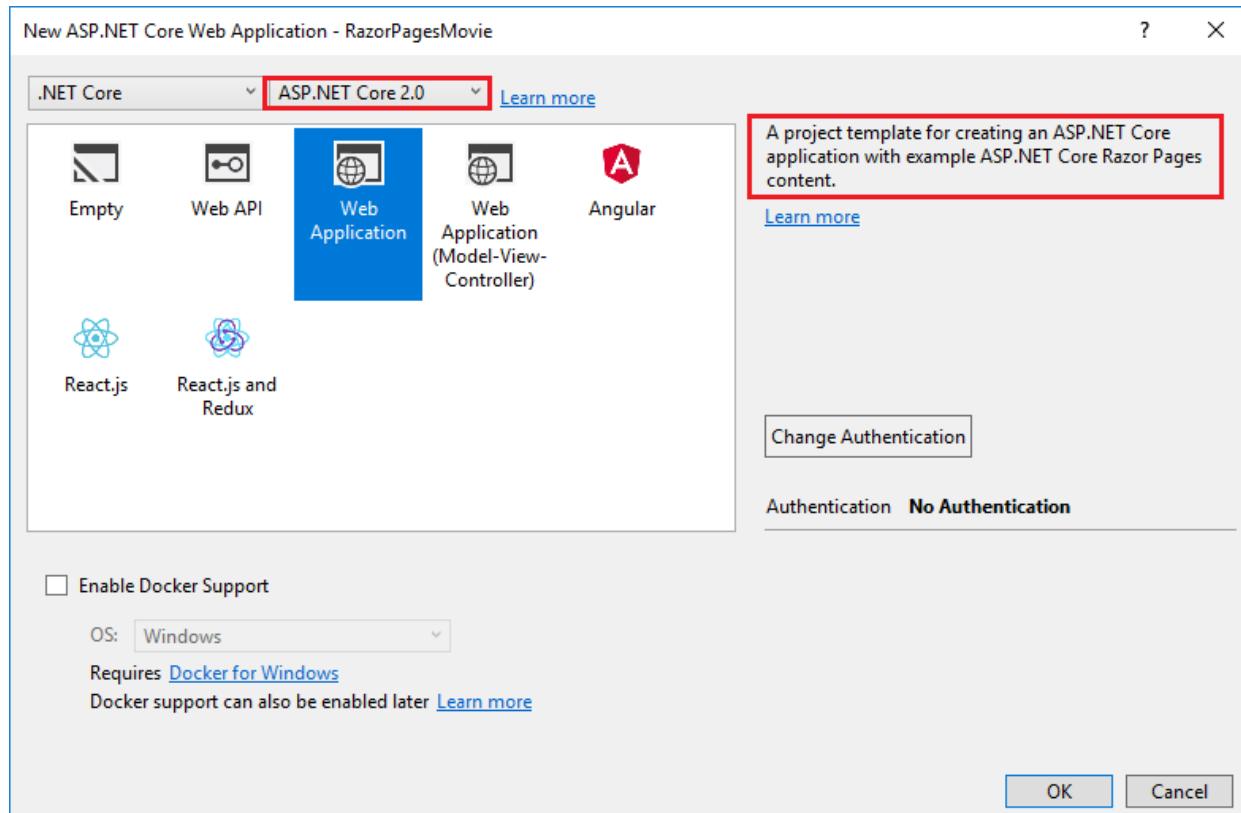
此网站的 UI 样式与内置模板生成的 UI 样式类似。教程的重点是 EF Core 和 Razor 页面，而非 UI。

创建 Razor 页面 Web 应用

- 从 Visual Studio“文件”菜单中选择“新建” > “项目”。
- 创建新的 ASP.NET Core Web 应用程序。将该项目命名为 ContosoUniversity。务必确保该项目命名为 ContosoUniversity，以便复制/粘贴代码时命名空间相匹配。



- 在下拉列表中选择“ASP.NET Core 2.0”，然后选择“Web 应用程序”。



按 F5 在调试模式下运行应用，或按 Ctrl-F5 在运行(不附加调试器)

设置网站样式

设置网站菜单、布局和主页时需作少量更改。

打开 Pages/_Layout.cshtml 并进行以下更改：

- 将“ContosoUniversity”的每个匹配项都改为“Contoso University”。共有三个匹配项。
- 添加菜单项 **Students, Courses, Instructors, 和 Department**，并删除 **Contact** 菜单项。

突出显示所作更改。(所有标记均不显示。)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href("~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
              value="absolute" />
        <link rel="stylesheet" href "~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Students/Index">Students</a></li>
                    <li><a asp-page="/Courses/Index">Courses</a></li>
                    <li><a asp-page="/Instructors/Index">Instructors</a></li>
                    <li><a asp-page="/Departments/Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2017 - Contoso University</p>
        </footer>
    </div>
```

在 Pages/Index.cshtml 中，将文件内容替换为以下代码，以将有关 ASP.NET 和 MVC 的文本替换为有关本应用的文

本：

```
@page
@model IndexModel
 @{
    ViewData["Title"] = "Home page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default"
            href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
            See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
            href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-rp/intro/samples/cu-final">
            See project source code &raquo;</a></p>
    </div>
</div>
```

按 Ctrl+F5 运行项目。将显示主页以及后续教程中创建的标签：

The screenshot shows a web browser window titled "Home page - Contoso U" with the URL "localhost:1234". The page content is as follows:

Contoso University

Welcome to Contoso University

Contoso University is a sample application that demonstrates how to use Entity Framework Core in an ASP.NET Core Razor Pages web app.

Build it from scratch

You can build the application by following the steps in a series of tutorials.

[See the tutorial »](#)

Download it

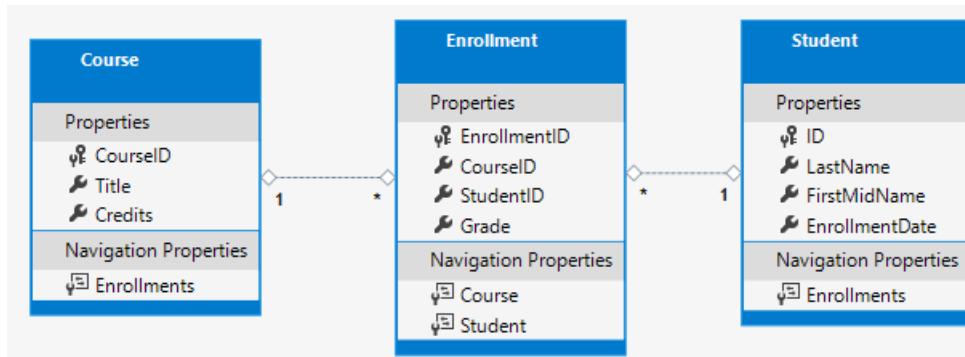
You can download the completed project from GitHub.

[See project source code »](#)

© 2017 - Contoso University

创建数据模型

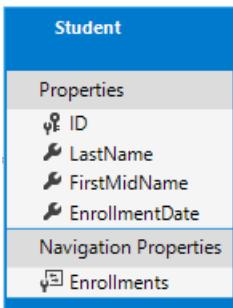
创建 Contoso University 应用的实体类。从以下三个实体开始：



Student 和 **Enrollment** 实体之间存在一对多关系。**Course** 和 **Enrollment** 实体之间存在一对多关系。一名学生可以报名参加任意数量的课程。一门课程中可以包含任意数量的学生。

以下部分将为这几个实体中的每一个实体创建一个类。

Student 实体



创建 Models 文件夹。在 Models 文件夹中，使用以下代码创建一个名为 Student.cs 的类文件：

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

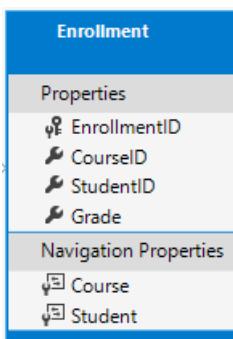
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`ID` 属性成为此类对应的数据库 (DB) 表的主键列。默认情况下，EF Core 将名为 `ID` 或 `classnameID` 的属性视为主键。`classnameID` 中的 `classname` 是类的名称，例如上述示例中的 `Student`。

`Enrollments` 属性是导航属性。导航属性链接到与此实体相关的其他实体。在这种情况下，`Student entity` 的 `Enrollments` 属性包含与该 `Student` 相关的所有 `Enrollment` 实体。例如，如果数据库中的 `Student` 行有两个相关的 `Enrollment` 行，则 `Enrollments` 导航属性包含这两个 `Enrollment` 实体。相关的 `Enrollment` 行是 `StudentID` 列中包含该学生的主键值的行。例如，假设 `ID=1` 的学生在 `Enrollment` 表中有两行。`Enrollment` 表中有两行的 `StudentID = 1`。`StudentID` 是 `Enrollment` 表中的外键，用于指定 `Student` 表中的学生。

如果导航属性包含多个实体，则导航属性必须是列表类型，例如 `ICollection<T>`。可以指定 `ICollection<T>` 或诸如 `List<T>` 或 `HashSet<T>` 的类型。使用 `ICollection<T>` 时，EF Core 会默认创建 `HashSet<T>` 集合。包含多个实体的导航属性来自于多对多和一对多关系。

Enrollment 实体



在 Models 文件夹中，使用以下代码创建 Enrollment.cs：

```

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}

```

`EnrollmentID` 属性为主键。`Student` 实体使用的是 `ID` 模式，而本实体使用的是 `classnameID` 模式。通常情况下，开发者会选择一种模式并在整个数据模型中都使用该模式。下一个教程将介绍如何使用不带类名的 ID，以便更轻松地在数据模型中实现集成。

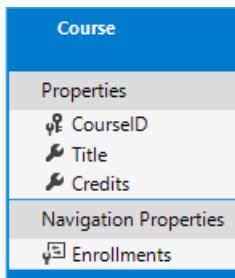
`Grade` 属性为 `enum`。`Grade` 声明类型后的 `?` 表示 `Grade` 属性可以为 null。评级为 null 和评级为零是有区别的 -- null 意味着评级未知或者尚未分配。

`StudentID` 属性是外键，其对应的导航属性为 `Student`。`Enrollment` 实体与一个 `Student` 实体相关联，因此该属性只包含一个 `Student` 实体。`Student` 实体与 `Student.Enrollments` 导航属性不同，后者包含多个 `Enrollment` 实体。

`CourseID` 属性是外键，其对应的导航属性为 `Course`。`Enrollment` 实体与一个 `Course` 实体相关联。

如果属性命名为 `<navigation property name><primary key property name>`，EF Core 会将其视为外键。例如 `Student` 导航属性的 `StudentID`，因为 `Student` 实体的主键为 `ID`。还可以将外键属性命名为 `<primary key property name>`。例如 `CourseID`，因为 `Course` 实体的主键为 `CourseID`。

Course 实体



在 Models 文件夹中，使用以下代码创建 Course.cs：

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

`Enrollments` 属性是导航属性。`Course` 实体可与任意数量的 `Enrollment` 实体相关。

应用可以通过 `DatabaseGenerated` 特性指定主键，而无需靠数据库生成。

创建 SchoolContext 数据库上下文

数据库上下文类是为给定数据模型协调 EF Core 功能的主类。数据上下文派生自 `Microsoft.EntityFrameworkCore.DbContext`。数据上下文指定数据模型中包含哪些实体。在此项目中，类命名为 `SchoolContext`。

在项目文件夹中，创建一个名为 Data 的文件夹。

在 Data 文件夹中，使用以下代码创建 `SchoolContext.cs`：

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {}

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}

```

此代码会为每个实体集创建一个 `DbSet` 属性。在 EF Core 术语中：

- 实体集通常对应一个数据库表。
- 实体对应表中的行。

`DbSet<Enrollment>` 和 `DbSet<Course>` 可以省略。EF Core 隐式包含了它们，因为 `Student` 实体引用 `Enrollment` 实体，而 `Enrollment` 实体引用 `Course` 实体。在本教程中，将 `DbSet<Enrollment>` 和 `DbSet<Course>` 保留在 `SchoolContext` 中。

创建数据库时，EF Core 会创建名称与 `DbSet` 属性名相同的表。集合的属性名通常采用复数形式（使用 `Students`，而不使用 `Student`）。开发者对表名称是否应为复数形式意见不一。在这些教程中，在 `DbContext` 中指定单数形式的表名称会覆盖默认行为。若要指定单数形式的表名称，请添加以下突出显示的代码：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

通过依赖关系注入注册上下文

ASP.NET Core 包含[依赖关系注入](#)。服务(例如 EF Core 数据库上下文)在应用程序启动期间通过依赖关系注入进行注册。需要这些服务(如 Razor 页面)的组件通过构造函数提供相应服务。本教程的后续部分介绍了用于获取数据库上下文实例的构造函数代码。

要将 `SchoolContext` 注册为服务, 请打开 `Startup.cs`, 并将突出显示的行添加到 `ConfigureServices` 方法。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

通过调用 `DbContextOptionsBuilder` 中的一个方法将数据库连接字符串在配置文件中的名称传递给上下文对象。进行本地开发时, [ASP.NET Core 配置系统](#)会从 `appsettings.json` 文件读取连接字符串。

为 `ContosoUniversity.Data` 和 `Microsoft.EntityFrameworkCore` 命名空间添加 `using` 语句。生成项目。

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

打开 `appsettings.json` 文件, 并按以下代码所示添加连接字符串:

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Server=  
(localdb)\\mssqllocaldb;Database=ContosoUniversity;ConnectRetryCount=0;Trusted_Connection=True;MultipleActiveResultSets=true"  
    },  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    }  
}
```

上述连接字符串使用 `ConnectRetryCount=0` 来防止 [SQLClient](#) 挂起。

SQL Server Express LocalDB

连接字符串指定 SQL Server LocalDB 数据库。LocalDB 是轻型版本 SQL Server Express 数据库引擎，专门针对应用开发，而非生产使用。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 会在 `C:/Users/<user>` 目录中创建 .mdf 数据库文件。

添加代码，以使用测试数据初始化该数据库

EF Core 会创建一个空的数据库。本部分中编写了 Seed 方法来使用测试数据填充该数据库。

在 Data 文件夹中，新建一个名为 `DbInitializer.cs` 的类文件，并添加以下代码：

```
using ContosoUniversity.Models;  
using System;  
using System.Linq;  
  
namespace ContosoUniversity.Data  
{  
    public static class DbInitializer  
    {  
        public static void Initialize(SchoolContext context)  
        {  
            context.Database.EnsureCreated();  
  
            // Look for any students.  
            if (context.Students.Any())  
            {  
                return; // DB has been seeded  
            }  
  
            var students = new Student[]  
            {  
                new Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Parse("2005-09-01")},  
                new Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Parse("2003-09-01")},  
                new Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Yan", LastName="Li", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Peggy", LastName="Justice", EnrollmentDate=DateTime.Parse("2001-09-01")},  
                new Student{FirstMidName="Laura", LastName="Norman", EnrollmentDate=DateTime.Parse("2003-09-01")},  
                new Student{FirstMidName="Nino", LastName="Olivetto", EnrollmentDate=DateTime.Parse("2005-09-01")}  
            };  
            foreach (Student s in students)  
            {  
                context.Students.Add(s);  
            }  
            context.SaveChanges();  
  
            var courses = new Course[] {
```

```

var courses = new Course[]
{
    new Course{CourseID=1050,Title="Chemistry",Credits=3},
    new Course{CourseID=4022,Title="Microeconomics",Credits=3},
    new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
    new Course{CourseID=1045,Title="Calculus",Credits=4},
    new Course{CourseID=3141,Title="Trigonometry",Credits=4},
    new Course{CourseID=2021,Title="Composition",Credits=3},
    new Course{CourseID=2042,Title="Literature",Credits=4}
};
foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
    new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
    new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
    new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
    new Enrollment{StudentID=3,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
    new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
    new Enrollment{StudentID=6,CourseID=1045},
    new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
    context.Enrollments.Add(e);
}
context.SaveChanges();
}
}
}

```

该代码会检查数据库中是否存在任何学生。如果数据库中没有任何学生，则会填充测试数据。它会将测试数据加载到数组中，而不是 `List<T>` 集合中，以便优化性能。

`EnsureCreated` 方法自动为数据库上下文创建数据库。如果数据库已存在，则返回 `EnsureCreated`，并且不修改数据库。

在 `Program.cs` 中，修改 `Main` 方法以执行以下操作：

- 从依赖关系注入容器获取数据库上下文实例。
- 调用 `seed` 方法，并将上下文传递给它。
- `Seed` 方法完成时释放上下文。

下面的代码显示更新后的 `Program.cs` 文件。

```

// Unused usings removed
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while seeding the database.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

第一次运行该应用时，会使用测试数据创建并填充数据库。更新数据模型时：

- 删除数据库。
- 更新 seed 方法。
- 运行该应用，并创建新的种子数据库。

后续教程中，在更改数据模型时会更新该数据库，而不会删除并重新创建该数据库。

添加基架工具

本部分中将使用包管理器控制台 (PMC) 来添加 Visual Studio Web 代码生成包。必须添加此包才能运行基架引擎。

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。

在包管理器控制台 (PMC) 中输入以下命令：

```

Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Utils

```

前一个命令将 NuGet 包添加到 *.csproj 文件中：

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>

```

构架模型

- 打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。
- 运行以下命令：

```

dotnet restore
dotnet aspnet-codegenerator razorpage -m Student -dc SchoolContext -udl -outDir Pages\Students --referenceScriptLibraries

```

如果收到错误：

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。

生成项目。此版本生成如下错误：

```
1>Pages\Students\Index.cshtml.cs(26,38,26,45): error CS1061: 'SchoolContext' does not contain a definition for 'Student'
```

将 `_context.Student` 全局更改为 `_context.Students` (即向 `Student` 添加一个"s")。找到并更新 7 个匹配项。我们计划在下一版本中修复[此 bug](#)。

下表详细说明了 ASP.NET Core 代码生成器的参数：

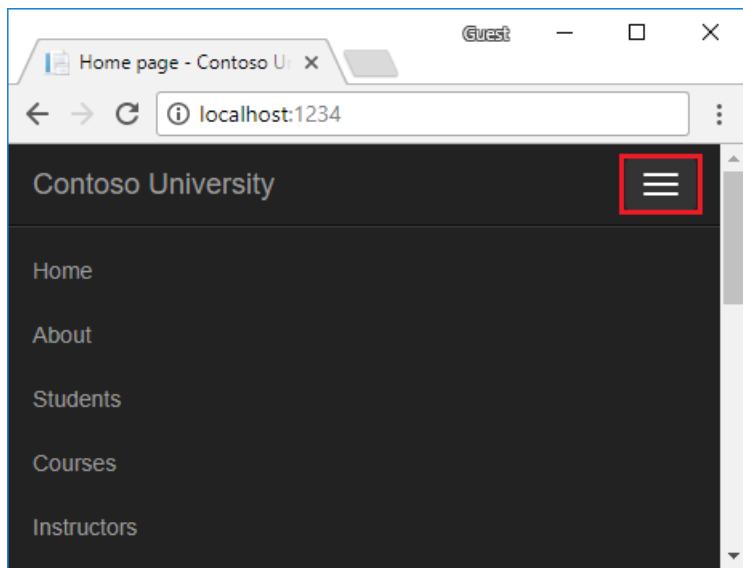
参数	描述
-m	模型的名称。
-dc	数据上下文。
-udl	使用默认布局。
-outDir	用于创建视图的相对输出文件夹路径。
--referenceScriptLibraries	向“编辑”和“创建”页面添加 <code>_ValidationScriptsPartial</code>

使用 `h` 开关获取 `aspnet-codegenerator razorpage` 命令方面的帮助：

```
dotnet aspnet-codegenerator razorpage -h
```

测试应用

运行应用并选择“学生”链接。“学生”链接将显示在页面顶部，具体取决于浏览器宽度。如果看不到“学生”链接，请单击右上角的导航图标。



测试“创建”，“编辑”和“详细信息”链接。

查看数据库

启动应用时，`DbInitializer.Initialize` 会调用 `EnsureCreated`。`EnsureCreated` 检测数据库是否存在，并在必要时创建一个数据库。如果数据库中没有学生，`Initialize` 方法将添加学生。

从 Visual Studio 中的“视图”菜单打开 SQL Server 对象资源管理器 (SSOX)。在 SSOX 中，单击“(localdb)\MSSQLLocalDB”>“数据库”>“ContosoUniversity1”。

展开“表”节点。

右键单击 Student 表，然后单击“查看数据”，以查看创建的列和插入到表中的行。

.mdf 和 .ldf 数据库文件位于 C:\Users\ 文件夹中。

启动应用时会调用 `EnsureCreated`，以进行以下工作流：

- 删除数据库。
- 更改数据库架构（例如添加一个 `EmailAddress` 字段）。
- 运行应用。

`EnsureCreated` 创建一个带有 `EmailAddress` 列的数据库。

约定

因为使用了约定或 EF Core 所做的假设，为使 EF Core 创建完整数据库而编写的代码量是最小的。

- 使用 `DbSet` 属性的名称作为表名。如果实体未被 `DbSet` 属性引用，实体类名称用作表名称。
- 使用实体属性名作为列名。
- 以 ID 或 `classnameID` 命名的实体属性被视为主键属性。
- 如果属性命名为，将视为外键属性（例如 `Student` 导航属性的 `StudentID`，因为 `Student` 实体的主键为 `ID`）。可将外键属性命名为（例如 `EnrollmentID`，因为 `Enrollment` 实体的主键为 `EnrollmentID`）。

可以重写常规行为。例如，可以显式指定表名，如本教程中前面部分所示。可以显式设置列名。可以显式设置主键

和外键。

异步代码

异步编程是 ASP.NET Core 和 EF Core 的默认模式。

Web 服务器的可用线程是有限的，而在高负载情况下的可能所有线程都被占用。当发生这种情况的时候，服务器就无法处理新请求，直到线程被释放。使用同步代码时，可能会出现多个线程被占用但不能执行任何操作的情况，因为它们正在等待 I/O 完成。使用异步代码时，当进程正在等待 I/O 完成，服务器可以将其线程释放用于处理其他请求。因此，使用异步代码可以更有效地利用服务器资源，并且可以让服务器在没有延迟的情况下处理更多流量。

异步代码会在运行时引入少量开销。流量较低时，对性能的影响可以忽略不计，但流量较高时，潜在的性能改善非常显著。

在以下代码中，`async` 关键字、`Task<T>` 返回值、`await` 关键字和 `ToListAsync` 方法让代码异步执行。

```
public async Task OnGetAsync()
{
    Student = await _context.Students.ToListAsync();
}
```

- `async` 关键字让编译器执行以下操作：
 - 为方法主体的各部分生成回调。
 - 自动创建返回的 `Task` 对象。有关详细信息，请参阅[任务返回类型](#)。
- 隐式返回类型 `Task` 表示正在进行的工作。
- `await` 关键字让编译器将该方法拆分为两个部分。第一部分是以异步方式结束已启动的操作。第二部分是当操作完成时注入调用回调方法的地方。
- `ToListAsync` 是 `ToList` 扩展方法的异步版本。

编写使用 EF Core 的异步代码时需要注意的一些事项：

- 只会异步执行导致查询或命令被发送到数据库的语句。这包括 `ToListAsync`、`SingleOrDefaultAsync`、`FirstOrDefaultAsync` 和 `SaveChangesAsync`。不包括只会更改 `IQueryable` 的语句，例如

```
var students = context.Students.Where(s => s.LastName == "Davolio")。
```
- EF Core 上下文并非线程安全：请勿尝试并行执行多个操作。
- 若要利用异步代码的性能优势，请验证在调用向数据库发送查询的 EF Core 方法时，库程序包（如用于分页）是否使用异步。

有关在 .NET 中进行异步编程的详细信息，请参阅[异步概述](#)。

下一个教程将介绍基本的 CRUD（创建、读取、更新、删除）操作。

[下一篇](#)

ASP.NET Core 中的 Razor 页面和 EF Core - CRUD - 第 2 个教程(共 8 个)

2018/5/14 • 14 min to read • [Edit Online](#)

作者:Tom Dykstra、Jon P Smith 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

本教程将介绍和自定义已搭建基架的 CRUD (创建、读取、更新、删除)代码。

请注意:为最大程度降低复杂性并让这些教程集中介绍 EF Core, “Razor 页面”页面模型中将使用 EF Core 代码。某些开发人员使用服务层或存储库模式在 UI(Razor 页面)和数据访问层之间创建抽象层。

本教程将修改“学生”文件夹中的“创建”、“编辑”、“删除”和“详细信息”Razor 页面。

基架代码将以下模式用于“创建”、“编辑”和“删除”页面:

- 使用 HTTP GET 方法 `OnGetAsync` 获取和显示请求数据。
- 使用 HTTP POST 方法 `OnPostAsync` 将更改保存到数据。

“索引”和“详细信息”页面使用 HTTP GET 方法 `OnGetAsync` 获取和显示请求数据

将 `SingleOrDefaultAsync` 替换为 `FirstOrDefaultAsync`

生成的代码使用 `SingleOrDefaultAsync` 提取请求的实体。`FirstOrDefaultAsync` 在提取一个实体时更为高效:

- 代码需要验证查询仅返回一个实体时除外。
- `SingleOrDefaultAsync` 会提取更多数据并执行不必要的工作。
- 如果有多个实体符合筛选部分, `SingleOrDefaultAsync` 将引发异常。
- 如果有多个实体符合筛选部分, `FirstOrDefaultAsync` 不引发异常。

将 `SingleOrDefaultAsync` 全局替换为 `FirstOrDefaultAsync`。`SingleOrDefaultAsync` 用于 5 个位置:

- “详细信息”页面中的 `OnGetAsync`。
- “编辑”和“删除”页面中的 `OnGetAsync` 和 `OnPostAsync`。

FindAsync

在大部分基架代码中, `FindAsync` 可用于替代 `FirstOrDefaultAsync` 或 `SingleOrDefaultAsync`。

`FindAsync`:

- 查找具有主键 (PK) 的实体。如果具有 PK 的实体正在由上下文跟踪, 会返回该实体且不向 DB 发出请求。
- 既简单又简洁。
- 经过优化后可查找单个实体。
- 在某些情况下可提供性能优势, 但它们很少用于普通的 Web 方案。
- 以隐式方式使用 `FirstAsync` 而不是 `SingleAsync`。如果想要“包含”其他实体, 则“查找”将不再适用。这意味着可能需要放弃“查找”并随着应用运行移动到查询。

自定义“详细信息”页

浏览到 `Pages/Students` 页面。“编辑”、“详细信息”和“删除”链接是在 `Pages/Students/Index.cshtml` 文件中由定位点

标记帮助器生成的。

```
<td>
    <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
    <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

选择“详细信息”链接。URL 的格式为 `http://localhost:5000/Students/Details?id=2`。“学生 ID”通过查询字符串 (`?id=2`) 进行传递。

更新“编辑”、“详细信息”和“删除”Razor 页面以使用 `{id:int}` 路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int}"`。

如果对具有不包含整数路由值的“`{id:int}`”路由模板的页面发起请求，则该请求将返回 HTTP 404(找不到)错误。例如，`http://localhost:5000/Students/Details` 返回 404 错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

运行应用，单击“详细信息”链接，并验证确认 URL 正在将 ID 作为路由数据 (

`http://localhost:5000/Students/Details/2`) 进行传递。

不要将 `@page` 全局更改为 `@page "{id:int}"`，执行此操作会将链接拆分为“主页”和“创建”页。

添加相关数据

“学生索引”页的基架代码不包括 `Enrollments` 属性。在本部分，`Enrollments` 集合的内容显示在“详细信息”页中。

Pages/Students/Details.cshtml.cs 的 `OnGetAsync` 方法使用 `FirstOrDefaultAsync` 方法检索单个 `Student` 实体。添加以下突出显示的代码：

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

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

`Include` 和 `ThenInclude` 方法使上下文加载 `Student.Enrollments` 导航属性，并在每个注册中加载 `Enrollment.Course` 导航属性。这些方法将在与数据读取相关的教程中进行详细介绍。

对于返回的实体未在当前上下文中更新的情况，`AsNoTracking` 方法将会提升性能。`AsNoTracking` 将在本教程的后续部分中讨论。

在“详细信息”页中显示相关注册

打开 Pages/Students/Details.cshtml。添加以下突出显示的代码以显示注册列表：

```

@page "{id:int}"
@model ContosoUniversity.Pages.Students.DetailsModel

 @{
     ViewData["Title"] = "Details";
 }

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page=".Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page=".Index">Back to List</a>
</div>

```

如果代码缩进在粘贴代码后出现错误，请按 CTRL-K-D 进行更正。

上面的代码循环通过 `Enrollments` 导航属性中的实体。它将针对每个注册显示课程标题和成绩。课程标题从 `Course` 实体中检索，该实体存储在 `Enrollments` 实体的 `Course` 导航属性中。

运行应用，选择“学生”选项卡，然后单击学生的“详细信息”链接。随即显示出所选学生的课程和成绩列表。

更新“创建”页

将 Pages/Students/Create.cshtml.cs 中的 `OnPostAsync` 方法更新为以下代码：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student", // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return null;
}
```

TryUpdateModelAsync

检查 `TryUpdateModelAsync` 代码：

```
var emptyStudent = new Student();

if (await TryUpdateModelAsync<Student>(
    emptyStudent,
    "student", // Prefix for form value.
    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
{
```

在前面的代码中，`TryUpdateModelAsync<Student>` 尝试使用 `PageModel` 的 `PageContext` 属性中已发布的表单值更新 `emptyStudent` 对象。`TryUpdateModelAsync` 仅更新列出的属性（`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`）。

在上述示例中：

- 第二个自变量 (`"student", // Prefix`) 是用于查找值的前缀。该自变量不区分大小写。
- 已发布的表单值通过 [模型绑定](#) 转换为 `Student` 模型中的类型。

过多发布

使用 `TryUpdateModel` 更新具有已发布值的字段是一种最佳的安全做法，因为这能阻止过多发布。例如，假设 `Student` 实体包含此网页不应更新或添加的 `Secret` 属性：

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

即使应用的创建/更新 Razor 页面上没有 `Secret` 字段，黑客仍可利用过多发布设置 `Secret` 值。黑客也可使用 Fiddler 等工具或通过编写某个 JavaScript 来发布 `Secret` 表单值。原始代码不会限制模型绑定器在创建“学生”实例时使用的字段。

黑客为 `Secret` 表单字段指定的任何值都会在 DB 中更新。下图显示 Fiddler 工具正在将 `Secret` 字段(值为“OverPost”)添加到已发布的表单值。

The screenshot shows the Fiddler interface with a red box highlighting the 'Execute' button in the top right. In the Request Body section, a red box highlights the '&Secret=OverPost' parameter.

值“OverPost”已成功添加到所插入行的 `Secret` 属性中。应用程序设计器绝不会在“创建”页设置 `Secret` 属性。

视图模型

视图模型通常包含应用程序所用的模型中包括的属性的子集。应用程序模型通常称为域模型。域模型通常包含 DB 中对应实体所需的全部属性。视图模型仅包含 UI 层(例如“创建”页)所需的属性。除视图模型外，某些应用使用绑定模型或输入模型在“Razor 页面”页面模型类和浏览器之间传递数据。请考虑以下 `Student` 视图模型：

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

视图模型还提供了一种防止过度发布的方法。视图模型仅包含要查看(显示)或更新的属性。

以下代码使用 `StudentVM` 视图模型创建新的学生：

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

`SetValues` 方法通过从另一个 `PropertyValues` 对象读取值来设置此对象的值。`SetValues` 使用属性名称匹配。视图模型类型不需要与模型类型相关，它只需要具有匹配的属性。

使用 `StudentVM` 时需要更新 `CreateVM.cshtml` 才能使用 `StudentVM` 而非 `Student`。

在 Razor 页面，`PageModel` 派生类就是视图模型。

更新“编辑”页

更新“编辑”页的页面模型。突出显示所作的主要更改：

```

public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }

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

        Student = await _context.Students.FindAsync(id);

        if (Student == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var studentToUpdate = await _context.Students.FindAsync(id);

        if (await TryUpdateModelAsync<Student>(
            studentToUpdate,
            "student",
            s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }

        return Page();
    }
}

```

代码更改与“创建”页类似，但有少数例外：

- `OnPostAsync` 具有可选的 `id` 参数。
- 当前学生是从 DB 提取的，而非通过创建空学生获得。
- 已将 `FirstOrDefaultAsync` 替换为 `FindAsync`。从主键中选择实体时，使用 `FindAsync` 是一个不错的选择。请参阅 [FindAsync](#) 了解详细信息。

测试“编辑”和“创建”页

创建和编辑几个学生实体。

实体状态

DB 上下文会随时跟踪内存中的实体是否已与其在 DB 中的对应行进行同步。DB 上下文同步信息可决定调用

`SaveChanges` 后的行为。例如，将新实体传递到 `Add` 方法时，该实体的状态设置为 `Added`。调用 `SaveChanges` 时，DB 上下文会发出 SQL INSERT 命令。

实体可能处于以下状态之一：

- `Added`：DB 中尚不存在实体。`SaveChanges` 方法发出 INSERT 语句。
- `Unchanged`：无需保存对该实体所作的任何更改。从 DB 中读取实体时，该实体将具有此状态。
- `Modified`：已修改实体的部分或全部属性值。`SaveChanges` 方法发出 UPDATE 语句。
- `Deleted`：已标记该实体进行删除。`SaveChanges` 方法发出 DELETE 语句。
- `Detached`：DB 上下文未跟踪该实体。

在桌面应用中，通常会自动设置状态更改。读取实体并执行更改后，实体状态自动更改为 `Modified`。调用 `SaveChanges` 会生成仅更新已更改属性的 SQL UPDATE 语句。

在 Web 应用中，读取实体并显示数据的 `DbContext` 将在页面呈现后进行处理。调用页面 `OnPostAsync` 方法时，将发出具有 `DbContext` 的新实例的 Web 请求。如果在这个新的上下文中重新读取实体，则会模拟桌面处理。

更新“删除”页

在此部分中，当对 `SaveChanges` 的调用失败时，将添加用于实现自定义错误消息的代码。添加字符串，使其包含可能的错误消息：

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }
    public string ErrorMessage { get; set; }
}
```

将 `OnGetAsync` 方法替换为以下代码：

```

public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ErrorMessage = "Delete failed. Try again";
    }

    return Page();
}

```

上述代码包含可选参数 `saveChangesError`。`saveChangesError` 指示学生对象删除失败后是否调用该方法。删除操作可能由于暂时性网络问题而失败。云端更可能出现暂时性网络错误。通过 UI 调用“删除”页 `OnGetAsync` 时，`saveChangesError` 为 `false`。当 `OnPostAsync` 调用 `OnGetAsync`（由于删除操作失败）时，`saveChangesError` 参数为 `true`。

“删除”页 `OnPostAsync` 方法

将 `OnPostAsync` 替换为以下代码：

```

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

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("./Delete",
            new { id = id, saveChangesError = true });
    }
}

```

上述代码检索所选的实体，然后调用 `Remove` 方法，将实体的状态设置为 `Deleted`。调用 `SaveChanges` 时生成

SQL DELETE 命令。如果 `Remove` 失败：

- 会捕获 DB 异常。
- 通过 `saveChangesError=true` 调用“删除”页 `OnGetAsync` 方法。

更新“删除”Razor 页面

将以下突出显示的错误消息添加到“删除”Razor 页面。

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DeleteModel

 @{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
```

测试“删除”。

常见错误

“学生/主页”或其他链接不起作用：

验证确认 Razor 页面包含正确的 `@page` 指令。例如，“学生/主页”Razor 页面不应包含路由模板：

```
@page "{id:int}"
```

每个 Razor 页面均必须包含 `@page` 指令。

[上一页](#) [下一页](#)

ASP.NET Core 中的 Razor 页面和 EF Core - 排序、筛选、分页 - 第 3 个教程(共 8 个)

2018/5/14 • 17 min to read • [Edit Online](#)

作者: [Tom Dykstra](#)、[Rick Anderson](#) 和 [Jon P Smith](#)

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

本教程将添加排序、筛选、分组和分页功能。

下图显示完整的页面。列标题是可单击的链接, 可用于对列进行排序。重复单击列标题可在升降和降序排序顺序之间切换。

The screenshot shows the 'Index' page of the Contoso University application. At the top, there's a search bar with 'Find by name:' and a 'Search' button, along with links to 'Create New' and 'Back to Full List'. Below the search bar is a table header with columns for 'Last Name', 'First Name', and 'Enrollment Date'. Underneath the table, there are three rows of student data:

Last Name	First Name	Enrollment Date	Action
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

At the bottom of the table, there are 'Previous' and 'Next' navigation buttons.

如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

向索引页添加排序

向 Students/Index.cshtml.cs `PageModel` 添加字符串, 使其包含排序参数:

```

public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

```

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync` :

```

public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                      select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}

```

上述代码接收来自 URL 中的查询字符串的 `sortOrder` 参数。该 URL (包括查询字符串) 由[定位点标记帮助器](#)生成

`sortOrder` 参数为“名称”或“日期”。`sortOrder` 参数后面可跟“_desc”以指定降序 (可选)。默认排序顺序为升序。

如果通过“学生”链接对“索引”页发起请求，则不会有任何查询字符串。学生按姓氏升序显示。按姓氏升序是 `switch` 语句中的默认顺序 (fall-through case)。用户单击列标题链接时，查询字符串值中会提供相应的 `sortOrder` 值。

Razor 页面使用 `NameSort` 和 `DateSort` 为列标题超链接配置相应的查询字符串值：

```

public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                      select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}

```

以下代码包含 C# ?: 运算符：

```

NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";

```

第一行指定当 `sortOrder` 为 NULL 或为空时，`NameSort` 设置为“name_desc”。如果 `sortOrder` 不为 NULL 或不为空，则 `NameSort` 设置为空字符串。

`?:` operator 也称为三元运算符。

通过这两个语句，视图可如下设置列标题超链接：

当前排序顺序	姓氏超链接	日期超链接
姓氏升序	descending	ascending
姓氏降序	ascending	ascending
日期升序	ascending	descending
日期降序	ascending	ascending

该方法使用 LINQ to Entities 指定要作为排序依据的列。此代码会初始化 switch 语句前面的 `IQueryable<Student>`，并在 switch 语句中对其进行修改：

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Students
                                      select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

创建或修改 `IQueryable` 时，不会向数据库发送任何查询。将 `IQueryable` 对象转换成集合后才能执行查询。通过调用 `IQueryable` 等方法可将 `ToListAsync` 转换成集合。因此，`IQueryable` 代码会生成单个查询，此查询直到出现以下语句才执行：

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` 可能获得包含大量列的详细信息。

向“学生索引”视图添加列标题超链接

将 `Students/Index.cshtml` 中的代码替换为以下突出显示的代码：

```

@page
@model ContosoUniversity.Pages.Students.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

前面的代码：

- 向 `LastName` 和 `EnrollmentDate` 列标题添加超链接。
- 使用 `NameSort` 和 `DateSort` 中的信息为超链接设置当前的排序顺序值。

若要验证排序是否生效：

- 运行应用并选择“学生”选项卡。
- 单击“姓氏”。
- 单击“注册日期”。

若要更好地了解此代码：

- 请在 Student/Index.cshtml.cs 中的 `switch (sortOrder)` 上设置断点。
- 添加对 `NameSort` 和 `DateSort` 的监视。
- 在 Student/Index.cshtml 中的 `@Html.DisplayNameFor(model => model.Student[0].LastName)` 上设置断点。

单步执行调试程序。

向“学生索引”页添加搜索框

若要向“学生索引”页添加筛选：

- 需要向 Razor 页面添加一个文本框和一个提交按钮。文本框会针对名字或姓氏提供一个搜索字符串。
- 页面模型随即更新以使用文本框值。

向 Index 方法添加筛选功能

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync`：

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Students
                                         select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

前面的代码：

- 向 `OnGetAsync` 方法添加 `searchString` 参数。从下一部分中添加的文本框中所接收搜索字符串值。
- 已向 LINQ 语句添加 `Where` 子句。`Where` 子句仅选择其名字或姓氏中包含搜索字符串的学生。只有存在要搜索的值时才执行 LINQ 语句。

请注意：上述代码调用 `IQueryable` 对象上的 `Where` 方法，且在服务器上处理该筛选器。在某些情况下，应用可能会将 `Where` 方法作为内存中集合上的扩展方法进行调用。例如，假设 `_context.Students` 从 EF Core `DbSet` 更改为可返回 `IEnumerable` 集合的存储库方法。结果通常是相同的，但在某些情况下可能不同。

例如，`Contains` 的 .NET Framework 实现会默认执行区分大小写的比较。在 SQL Server 中，`Contains` 区分大小写由 SQL Server 实例的排序规则设置决定。SQL Server 默认为不区分大小写。可调用 `ToUpper`，进行不区分大小写的显式测试：

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))
```

如果上述代码改为使用 `IEnumerable`，则该代码会确保结果区分大小写。如果在 `IEnumerable` 集合上调用 `Contains`，则使用 .NET Core 实现。如果在 `IQueryable` 对象上调用 `Contains`，则使用数据库实现。从存储库返回 `IEnumerable` 可能会大幅降低性能：

1. 所有行均从 DB 服务器返回。
2. 筛选应用于应用程序中所有返回的行。

调用 `ToUpper` 不会对性能产生负面影响。`ToUpper` 代码会在 TSQL SELECT 语句的 WHERE 子句中添加一个函数。添加的函数会防止优化器使用索引。如果安装的 SQL 区分大小写，则最好避免在必要时调用 `ToUpper`。

向“学生索引”视图添加搜索框

在 Views/Student/Index.cshtml 中，添加以下突出显示的代码以创建“搜索”按钮和各种 chrome。

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}



## Index



Create New



Find by name:  

 
         | 
        Back to full List






```

上述代码使用 `<form>` 标记帮助器来添加搜索文本框和按钮。默认情况下，`<form>` 标记帮助器利用 POST 提交表单数据。借助 POST，会在 HTTP 消息正文中而不是在 URL 中传递参数。使用 HTTP GET 时，表单数据作为查询字符串在 URL 中进行传递。通过查询字符串传递数据时，用户可对 URL 添加书签。[W3C 指南](#)建议应在操作不引起更新的情况下使用 GET。

测试应用：

- 选择“学生”选项卡并输入搜索字符串。
- 选择“搜索”。

请注意，该 URL 包含搜索字符串。

```
http://localhost:5000/Students?SearchString=an
```

如果页面具有书签，该书签将包含该页面的 URL 和 `SearchString` 查询字符串。`form` 标记中的 `method="get"` 会导致生成查询字符串。

目前，选中列标题排序链接时，“搜索”框中的筛选值会丢失。丢失的筛选值在下一部分进行修复。

向“学生索引”页添加分页功能

本部分将创建一个 `PaginatedList` 类来支持分页。`PaginatedList` 类使用 `Skip` 和 `Take` 语句在服务器上筛选数据，而不是检索所有表格行。下图显示了分页按钮。

The screenshot shows a web browser window with the title "Index - Contoso University". The address bar indicates the URL is "localhost:5813/Student". The main content area is titled "Contoso University" and contains the word "Index". Below this, there is a search bar with the placeholder "Find by name:" and a "Search" button. To the right of the search bar is a link "Back to Full List". The main content area displays a table with three rows of student data:

Last Name	First Name	Enrollment Date	Action
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

At the bottom of the table, there are two buttons: "Previous" and "Next".

在项目文件夹中，使用以下代码创建 `PaginatedList.cs`：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}

```

上述代码中的 `CreateAsync` 方法会提取页面大小和页码，并将相应的 `Skip` 和 `Take` 语句应用于 `IQueryable`。当在 `IQueryable` 上调用 `ToListAsync` 时，它将返回仅包含所请求页的列表。属性 `HasPreviousPage` 和 `HasNextPage` 用于启用或禁用“上一页”和“下一页”分页按钮。

`CreateAsync` 方法用于创建 `PaginatedList<T>`。构造函数不能创建 `PaginatedList<T>` 对象；构造函数不能运行异步代码。

向 Index 方法添加分页功能

在 `Students/Index.cshtml.cs` 中，将 `student` 的类型从 `IList<Student>` 更新到 `PaginatedList<Student>`：

```
public PaginatedList<Student> Student { get; set; }
```

用以下代码更新 Students/Index.cshtml.cs `OnGetAsync` :

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Students
                                         select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

上述代码会向方法签名添加页面索引、当前的 `sortOrder` 和 `currentFilter`。

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

出现以下情况时，所有参数均为 NULL：

- 从“学生”链接调用页面。
- 用户尚未单击分页或排序链接。

单击分页链接后，页面索引变量将包含要显示的页码。

`CurrentSort` 为 Razor 页面提供当前排序顺序。必须在分页链接中包含当前排序顺序才能在分页时保留排序顺序。

`CurrentFilter` 为 Razor 页面提供当前的筛选字符串。`CurrentFilter` 值：

- 必须包含在分页链接中才能在分页过程中保留筛选设置。
- 必须在重新显示页面时还原到文本框。

如果在分页时更改搜索字符串，页码会重置为 1。页面必须重置为 1，因为新的筛选器会导致显示不同的数据。输入搜索值并选择“提交”时：

- 搜索字符串将会更改。
- `searchString` 参数不为 NULL。

```
if (searchString != null)
{
    pageIndex = 1;
}
else
{
    searchString = currentFilter;
}
```

`PaginatedList.CreateAsync` 方法会将学生查询转换为支持分页的集合类型中的单个学生页面。单个学生页面会传递到 Razor 页面。

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

`PaginatedList.CreateAsync` 中的两个问号表示 [NULL 合并运算符](<https://docs.microsoft.com/dotnet/csharp/language-reference/operators/null-conditional-operator>)。NULL 合并运算符定义可为 NULL 的类型的默认值。`(pageIndex ?? 1)` 表达式表示返回 `pageIndex` 的值(若带有值)。如果 `pageIndex` 没有值，则返回 1。

向“学生”Razor 页面添加分页链接

更新 `Students/Index.cshtml` 中的标记。突出显示所作更改：

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page=".Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page=".Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
```

```

        <a asp-page=".~/Index" asp-route-sortOrder="@Model.NameSort"
           asp-route-currentFilter="@Model.CurrentFilter">
           @Html.DisplayNameFor(model => model.Student[0].LastName)
        </a>
        </th>
    <th>
        @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
    </th>
    <th>
        <a asp-page=".~/Index" asp-route-sortOrder="@Model.DateSort"
           asp-route-currentFilter="@Model.CurrentFilter">
           @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
        </a>
    </th>
    <th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model.Student) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            <a asp-page=".~/Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".~/Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".~/Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

@{
    var prevDisabled = !Model.Student.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Student.HasNextPage ? "disabled" : "";
}

<a asp-page=".~/Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@((Model.StudentPageIndex - 1))"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page=".~/Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@((Model.StudentPageIndex + 1))"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>

```

列标题链接使用查询字符串将当前搜索字符串传递到 `OnGetAsync` 方法，让用户可对筛选结果进行排序：

```

<a asp-page=".~/Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
   @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>

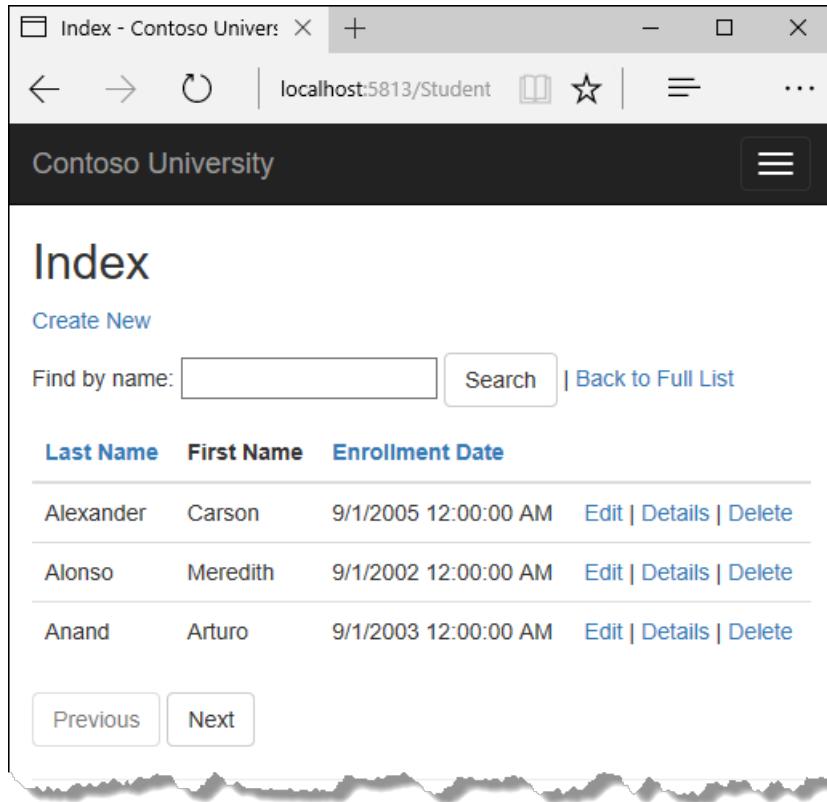
```

分页按钮由标记帮助器显示：

```
<a asp-page=".~/Index"
    asp-route-sortOrder="@Model.CurrentSort"
    asp-route-pageIndex="@Model.StudentPageIndex - 1"
    asp-route-currentFilter="@Model.CurrentFilter"
    class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page=".~/Index"
    asp-route-sortOrder="@Model.CurrentSort"
    asp-route-pageIndex="@Model.StudentPageIndex + 1"
    asp-route-currentFilter="@Model.CurrentFilter"
    class="btn btn-default @nextDisabled">
    Next
</a>
```

运行应用并导航到学生页面。

- 为确保分页生效，请单击不同排序顺序的分页链接。
- 要验证确保分页后可正确地排序和筛选，请输入搜索字符串并尝试分页。



若要更好地了解此代码：

- 请在 Student/Index.cshtml.cs 中的 `switch (sortOrder)` 上设置断点。
- 添加对 `NameSort`、`DateSort`、`CurrentSort` 和 `Model.StudentPageIndex` 的监视。
- 在 Student/Index.cshtml 中的 `@Html.DisplayNameFor(model => model.Student[0].LastName)` 上设置断点。

单步执行调试程序。

更新“关于”页以显示学生统计信息

此步骤将更新 Pages/About.cshtml，显示每个注册日期的已注册学生的数量。更新需使用分组并包括以下步骤：

- 为“关于”页使用的数据创建视图模型。

- 修改“关于”Razor 页面和页面模型。

创建视图模型

在 Models 文件夹中创建一个 SchoolViewModels 文件夹。

在 SchoolViewModels 文件夹中，使用以下代码添加 EnrollmentDateGroup.cs：

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

更新“关于”页面模型

用以下代码更新 Pages/About.cshtml.cs 文件：

```
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };
            Student = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

LINQ 语句按注册日期对学生实体进行分组，计算每组中实体的数量，并将结果存储在 EnrollmentDateGroup 视图模型对象的集合中。

注意: EF Core 当前不支持 LINQ `group` 命令。在上述代码中, 所有学生记录均从 SQL Server 返回。`group` 命令应用于 Razor 页面应用, 而不是应用于 SQL Server。EF Core 2.1 将支持此 LINQ `group` 运算符, 并在 SQL Server 上进行分组。请参阅 [Relational: Support translating GroupBy\(\) to SQL](#)(关系: 支持将 GroupBy() 转换到 SQL)。EF Core 2.1 将随 .NET Core 2.1 一起发布。有关详细信息, 请参阅 [.NET Core Roadmap](#)(.NET Core 路线图)。

修改“关于”Razor 页面

将 Views/Home/About.cshtml 文件中的代码替换为以下代码:

```
@page
@model ContosoUniversity.Pages.AboutModel

 @{
    ViewData["Title"] = "Student Body Statistics";
}

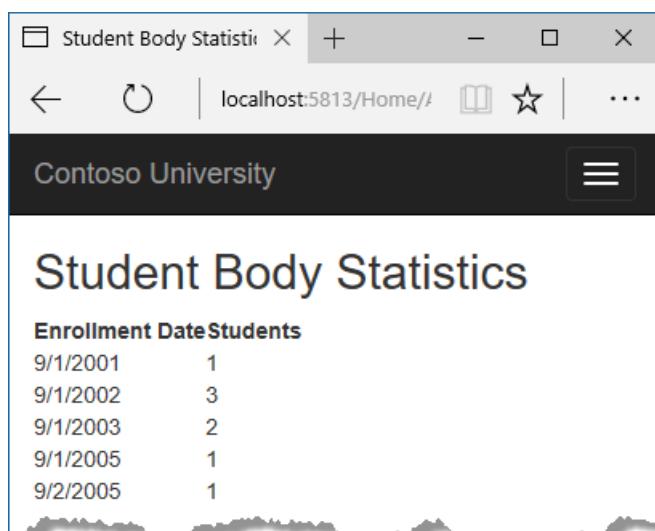
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

运行应用并导航到“关于”页面。表格中会显示每个注册日期的学生计数。

如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。



其他资源

- [调试 ASP.NET Core 2.x 源](#)

在下一教程中，应用将利用迁移更新数据模型。

[上一页](#)

[下一页](#)

ASP.NET Core 中的 Razor 页面和 EF Core - 迁移 - 第 4 个教程(共 8 个)

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Tom Dykstra](#)、[Jon P Smith](#) 和 [Rick Anderson](#)

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

本教程使用 EF Core 迁移功能管理数据模型更改。

如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

开发新应用时, 数据模型会频繁更改。每当模型发生更改时, 都无法与数据库进行同步。本教程首先配置 Entity Framework 以创建数据库(如果不存在)。每当数据模型发生更改时:

- DB 都会被删除。
- EF 都会创建一个新数据库来匹配该模型。
- 应用使用测试数据为 DB 设定种子。

这种使 DB 与数据模型保持同步的方法适用于多种情况, 但将应用部署到生产环境的情况除外。当应用在生产环境中运行时, 应用通常会存储需要保留的数据。每当发生更改(例如添加新列)时, 应用都无法在具有测试 DB 的环境下启动。EF Core 迁移功能可通过使 EF Core 更新 DB 架构而不是创建新 DB 来解决此问题。

数据模型发生更改时, 迁移将更新架构并保留现有数据, 而无需删除或重新创建 DB。

用于进行迁移的 Entity Framework Core NuGet 包

要使用迁移, 请使用“包管理器控制台”(PMC) 或命令行接口 (CLI)。以下教程演示如何使用 CLI 命令。有关 PMC 的信息, 请转到[本教程末尾](#)。

`Microsoft.EntityFrameworkCore.Tools.DotNet` 中提供了适用于命令行接口 (CLI) 的 EF Core 工具。若要安装此程序包, 请将它添加到 .csproj 文件中的 `DotNetCliToolReference` 集合, 如下所示。注意: 必须通过编辑 .csproj 文件来安装此程序包。不能使用 `install-package` 命令或包管理器 GUI 安装此程序包。要编辑 .csproj 文件, 请右键单击解决方案资源管理器中的项目名称, 然后选择“编辑 ContosoUniversity.csproj”。

以下标记显示已更新的 .csproj 文件, 其中突出显示了 EF Core CLI 工具:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  </ItemGroup>
</Project>
```

编写本教程时, 前面示例中的版本号是最新的。请对在其他程序包中发现的 EF Core CLI 工具使用相同的版本。

更改连接字符串

在 appsettings.json 文件中，将连接字符串中 DB 名称更改为 ContosoUniversity2。

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=  
      (localdb)\\mssqllocaldb;Database=ContosoUniversity2;ConnectRetryCount=0;Trusted_Connection=True;MultipleActive  
      ResultSets=true"  
  },  
}
```

更改连接字符串中的 DB 名称会导致初始迁移创建新的 DB。之所以会创建新的 DB，是因为不存在具有该名称的 DB。使用迁移无需更改连接字符串。

更改 DB 名称的另一种方法是删除 DB。使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` CLI 命令：

```
dotnet ef database drop
```

下面的部分说明如何运行 CLI 命令。

创建初始迁移

生成项目。

打开命令窗口并导航到项目文件夹。项目文件夹包含 Startup.cs 文件。

在命令窗口中输入以下内容：

```
dotnet ef migrations add InitialCreate
```

命令窗口显示类似于以下内容的信息：

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]  
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key  
      repository and Windows DPAPI to encrypt keys at rest.  
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]  
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider  
      'Microsoft.EntityFrameworkCore.SqlServer' with options: None  
Done. To undo this action, use 'ef migrations remove'
```

如果迁移失败，并出现消息“无法访问文件...ContosoUniversity.dll，因为它正被另一个进程使用。”：

- 停止 IIS Express。
 - 退出并重启 Visual Studio，或
 - 在 Windows 系统托盘中找到 IIS Express 图标。
 - 右键单击 IIS Express 图标，然后单击“ContosoUniversity”>“停止站点”。

如果出现错误消息“生成失败。”，请再次运行该命令。如果收到此错误，请在本教程底部留下说明。

了解 Up 和 Down 方法

EF Core 命令 `migrations add` 已生成用于创建 DB 的代码。此迁移代码位于 Migrations<timestamp>_InitialCreate.cs 文件中。`InitialCreate` 类的 `Up` 的方法创建与数据模型实体集相对应的 DB 表。`Down` 方法删除这些表，如下例所示：

```

public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(type: "int", nullable: false),
                Credits = table.Column<int>(type: "int", nullable: false),
                Title = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}

```

迁移调用 `Up` 方法为迁移实现数据模型更改。输入用于回退更新的命令时，迁移调用 `Down` 方法。

前面的代码适用于初始迁移。该代码是运行 `migrations add InitialCreate` 命令时创建的。迁移名称参数（本示例中为“InitialCreate”）用于指定文件名。迁移名称可以是任何有效的文件名。最好选择能概括迁移中所执行操作的字词或短语。例如，添加了系表的迁移可称为“AddDepartmentTable”。

如果创建了初始迁移并且存在 DB：

- 会生成 DB 创建代码。
- DB 创建代码不需要运行，因为 DB 已与数据模型相匹配。即使 DB 创建代码运行也不会做出任何更改，因为 DB 已与数据模型相匹配。

如果将应用部署到新环境，则必须运行 DB 创建代码才能创建 DB。

以前，需要更改连接字符串才能使用 DB 的新名称。指定的 DB 不存在，因此迁移会创建 DB。

数据模型快照

迁移在 `Migrations/SchoolContextModelSnapshot.cs` 中创建当前数据库架构的快照。添加迁移时，EF 会通过将数据模型与快照文件进行对比来确定已更改的内容。

删除迁移时，请使用 `dotnet ef migrations remove` 命令。`dotnet ef migrations remove` 删除迁移，并确保正确重置快照。

有关如何使用快照文件的详细信息，请参阅[团队环境中的 EF Core 迁移](#)。

删除 EnsureCreated

以前的开发通常使用 `EnsureCreated` 命令。本教程将使用迁移。`EnsureCreated` 具有以下限制：

- 绕过迁移并创建 DB 和架构。

- 不会创建迁移表。
- 不能与迁移一起使用。
- 专门用于在频繁删除并重新创建 DB 的情况下进行测试或快速制作原型。

删除 `DbInitializer` 中的以下行：

```
context.Database.EnsureCreated();
```

在开发过程中将迁移应用于 DB

在命令窗口中，输入以下内容以创建 DB 和表。

```
dotnet ef database update
```

注意：如果 `update` 命令返回“生成失败。”错误：

- 请再次运行该命令。
- 如果再次失败，请退出 Visual Studio，然后运行 `update` 命令。
- 请在页面底部留言。

该命令的输出与 `migrations add` 命令的输出相似。上面的命令中显示了用于设置 DB 的 SQL 命令的日志。下面的示例输出中省略了大部分日志：

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
      repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (467ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );
<logs omitted for brevity>
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES ('N'20170816151242_InitialCreate', N'2.0.0-rtm-26452');
Done.
```

要降低日志消息的详细级别，请更改 `appsettings.Development.json` 文件中的日志级别。有关详细信息，请参阅[日志记录介绍](#)。

使用 SQL Server 对象资源管理器检查 DB。请注意，增加了 `__EFMigrationsHistory` 表。`__EFMigrationsHistory` 表跟踪已应用到 DB 的迁移。查看 `__EFMigrationsHistory` 表中的数据，其中显示对应初始迁移的一行数据。上面的 CLI 输出示例中最后部分的日志显示了创建此行的 `INSERT` 语句。

运行应用并验证一切正常运行。

在生产环境中应用迁移

不建议生产应用在应用程序启动时调用 `Database.Migrate`。不应从服务器场中的应用调用 `Migrate`。例如，已将应用在云中部署为横向扩展(运行应用的多个示例)的情况。

应在部署过程中以受控的方式执行数据库迁移。生产数据库迁移方法包括：

- 使用迁移创建 SQL 脚本，并在部署过程中使用 SQL 脚本。
- 在受控的环境中运行 `dotnet ef database update`。

EF Core 使用 `_MigrationsHistory` 表查看是否需要运行任何迁移。如果 DB 已是最新，则无需运行迁移。

命令行接口 (CLI) 与包管理器控制台 (PMC)

可通过以下项使用可管理迁移的 EF Core 工具：

- .NET Core CLI 命令。
- Visual Studio 包管理器控制台 (PMC) 窗口中的 PowerShell cmdlet。

本教程演示如何使用 CLI，但某些开发人员更倾向于使用 PMC。

适用于 PMC 的 EF Core 命令位于 `Microsoft.EntityFrameworkCore.Tools` 程序包中。此包包含在 `Microsoft.AspNetCore.All` 元包中，因此无需另外安装。

重要说明：此程序包与通过编辑 `.csproj` 文件为 CLI 安装的程序包不同。此程序包的名称以 `Tools` 结尾，而 CLI 程序包的名称以 `Tools.DotNet` 结尾。

有关 CLI 命令的详细信息，请参阅 [.NET Core CLI](#)。

有关 PMC 命令的详细信息，请参阅 [包管理器控制台 \(Visual Studio\)](#)。

疑难解答

请下载[本阶段的已完成应用](#)。

应用会生成以下异常：

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.  
The login failed.  
Login failed for user 'user name'.
```

解决方案：运行 `dotnet ef database update`

如果 `update` 命令返回“生成失败。”错误：

- 请再次运行该命令。
- 请在页面底部留言。

ASP.NET Core 中的 Razor 页面和 EF Core - 数据模型 - 第 5 个教程(共 8 个)

2018/5/14 • 32 min to read • [Edit Online](#)

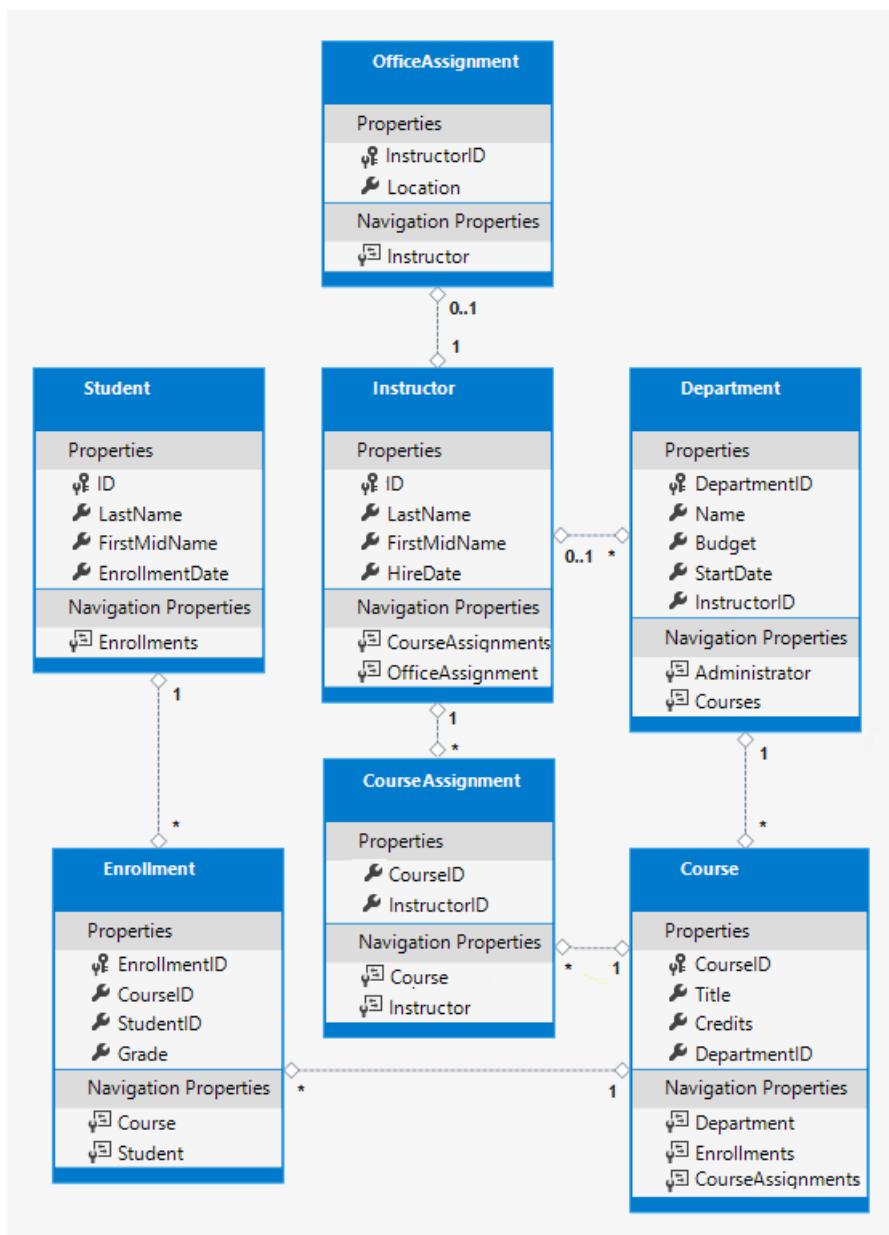
作者:Tom Dykstra 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

前面的教程介绍了由三个实体组成的基本数据模型。本教程将演示如何:

- 添加更多实体和关系。
- 通过指定格式设置、验证和数据库映射规则来自定义数据模型。

已完成数据模型的实体类如下图所示:



如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

使用特性自定义数据模型

此部分将使用特性自定义数据模型。

DataType 特性

学生页面当前显示注册日期。通常情况下，日期字段仅显示日期，不显示时间。

用以下突出显示的代码更新 *Models/Student.cs*:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

DataType 特性指定比数据库内部类型更具体的数据类型。在此情况下，应仅显示日期，而不是日期加时间。

DataType 枚举 提供多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用还可通过 **DataType** 特性自动提供类型特定的功能。例如：

- **mailto:** 链接将依据 **DataType.EmailAddress** 自动创建。
- 大多数浏览器中都提供面向 **DataType.Date** 的日期选择器。

DataType 特性发出 HTML 5 **data-** (读作 data dash) 特性供 HTML 5 浏览器使用。**DataType** 特性不提供验证。

DataType.Date 不指定显示日期的格式。默认情况下，日期字段根据基于服务器的 **CultureInfo** 的默认格式进行显示。

DisplayFormat 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

ApplyFormatInEditMode 设置指定还应对编辑 UI 应用该格式设置。某些字段不应使用 **ApplyFormatInEditMode**。例如，编辑文本框中通常不应显示货币符号。

DisplayFormat 特性可由其本身使用。搭配使用 **DataType** 特性和 **DisplayFormat** 特性通常是很好的做法。

DataType 特性按照数据在屏幕上的呈现方式传达数据的语义。**DataType** 特性可提供 **DisplayFormat** 中所不具有的以下优点：

- 浏览器可启用 HTML5 功能。例如，显示日历控件、区域设置适用的货币符号、电子邮件链接、客户端输入验证等。
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。

有关详细信息，请参阅 [<input> 标记帮助器文档](#)。

运行应用。导航到学生索引页。将不再显示时间。使用 **Student** 模型的每个视图将显示日期，不显示时间。

Last Name	First Name	Enrollment Date	
Alexander	Carson	2005-09-01	Edit Details Delete
Alonso	Meredith	2002-09-01	Edit Details Delete
Anand	Arturo	2003-09-01	Edit Details Delete

StringLength 特性

可使用特性指定数据验证规则和验证错误消息。StringLength 特性指定数据字段中允许的字符的最小长度和最大长度。StringLength 特性还提供客户端和服务器端验证。最小值对数据库架构没有任何影响。

使用以下代码更新 Student 模型：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }

        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]  
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

上面的代码将名称限制为不超过 50 个字符。StringLength 特性不会阻止用户在名称中输入空格。

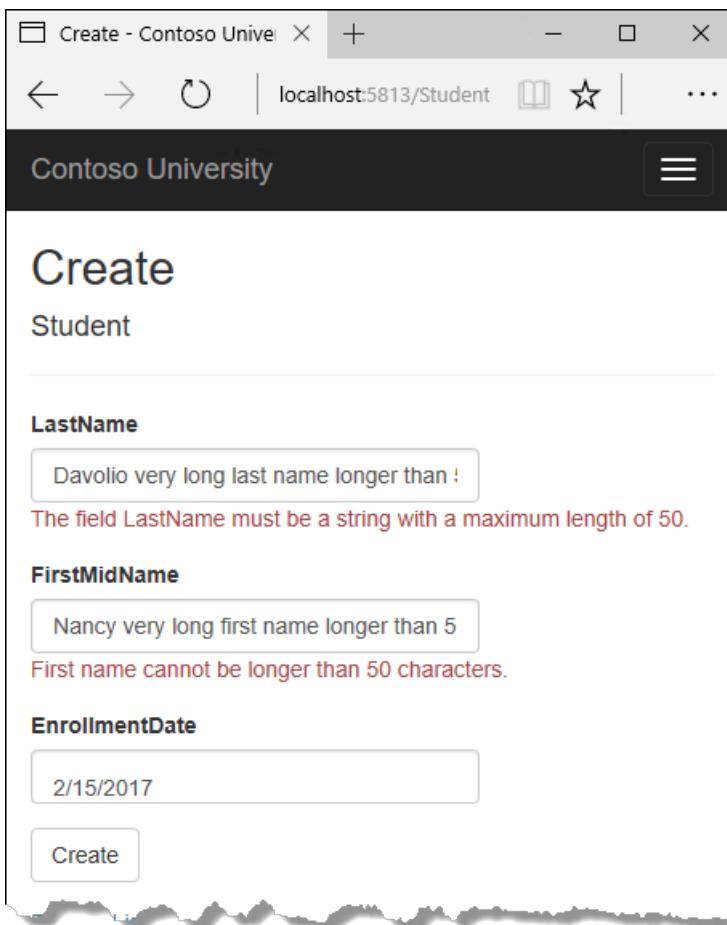
RegularExpression 特性用于向输入应用限制。例如，以下代码要求第一个字符为大写，其余字符按字母顺序排列：

```
[RegularExpression(@"^([A-Z][a-zA-Z]*$)")]
```

运行应用：

- 导航到学生页。

- 选择“新建”并输入不超过 50 个字符的名称。
- 选择“创建”时，客户端验证会显示一条错误消息。



在“SQL Server 对象资源管理器”(SSOX) 中，双击 Student 表，打开 Student 表设计器。

	Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>	
EnrollmentDate	datetime2(7)	<input type="checkbox"/>	
FirstMidName	nvarchar(MAX)	<input checked="" type="checkbox"/>	
LastName	nvarchar(MAX)	<input checked="" type="checkbox"/>	

Keys (1)
PK_Student (Primary Key, Clustered)

Check Constraints (0)

Indexes (0)

Foreign Keys (0)

Triggers (0)

```

CREATE TABLE [dbo].[Student] (
    [ID] INT IDENTITY (1, 1) NOT NULL,
    [EnrollmentDate] DATETIME2 (7) NOT NULL,
    [FirstMidName] NVARCHAR (MAX) NULL,
    [LastName] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_Student] PRIMARY KEY CLUSTERED ([ID] ASC)
);

```

上图显示 Student 表的架构。名称字段的类型为 nvarchar(MAX)，因为数据库上尚未运行迁移。稍后在本教程中运行迁移时，名称字段将变成 nvarchar(50)。

Column 特性

特性可以控制类和属性映射到数据库的方式。在本部分，Column 特性用于将 FirstMidName 属性的名称映射到数据库中的“FirstName”。

创建数据库后，模型上的属性名将用作列名(使用 Column 特性时除外)。

`Student` 模型使用 `FirstMidName` 作为名字字段，因为该字段也可能包含中间名。

用以下突出显示的代码更新 `Student.cs` 文件：

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]  
[Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

进行上述更改后，应用中的 `Student.FirstMidName` 将映射到 `Student` 表的 `FirstName` 列。

添加 `Column` 特性后，`SchoolContext` 的支持模型会发生改变。`SchoolContext` 的支持模型将不再与数据库匹配。

如果在执行迁移前运行应用，则会生成如下异常：

```
SqlException: Invalid column name 'FirstName'.
```

若要更新数据库：

- 生成项目。
- 在项目文件夹中打开命令窗口。输入以下命令以创建新迁移并更新数据库：

```
dotnet ef migrations add ColumnFirstName  
dotnet ef database update
```

`dotnet ef migrations add ColumnFirstName` 命令将生成如下警告消息：

```
An operation was scaffolded that may result in the loss of data.  
Please review the migration for accuracy.
```

生成警告的原因是名称字段现已限制为 50 个字符。如果数据库中的名称超过 50 个字符，则第 51 个字符及后面的所有字符都将丢失。

- 测试应用。

在 SSOX 中打开 `Student` 表：

	Name	Data Type	Allow Nulls
1	ID	int	<input type="checkbox"/>
2	EnrollmentDate	datetime2(7)	<input type="checkbox"/>
3	FirstName	nvarchar(50)	<input checked="" type="checkbox"/>
4	LastName	nvarchar(50)	<input checked="" type="checkbox"/>

Keys (1)
PK_Students (Primary Key, Clustered)

Check Constraints (0)

Indexes (0)

Foreign Keys (0)

Triggers (0)

Design T-SQL

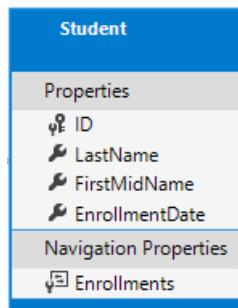
```
1 CREATE TABLE [dbo].[Students] (
2     [ID] INT IDENTITY (1, 1) NOT NULL,
3     [EnrollmentDate] DATETIME2 (7) NOT NULL,
4     [FirstName] NVARCHAR (50) NULL,
5     [LastName] NVARCHAR (50) NULL,
6     CONSTRAINT [PK_Students] PRIMARY KEY CLUSTERED ([ID] ASC)
7 );
8
```

执行迁移前，**名称**列的类型为 `nvarchar(MAX)`。名称列现在的类型为 `nvarchar(50)`。列名已从 `FirstMidName` 更改为 `FirstName`。

注意

在下一部分中，在某些阶段生成应用会生成编译器错误。说明用于指定生成应用的时间。

Student 实体更新



用以下代码更新 `Models/Student.cs`:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

Required 特性

`Required` 特性使名称属性成为必填字段。值类型(`DateTime`、`int`、`double`)等不可为 NULL 的类型不需要 `Required` 特性。系统会将不可为 NULL 的类型自动视为必填字段。

不能用 `StringLength` 特性中的最短长度参数替换 `Required` 特性：

```

[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }

```

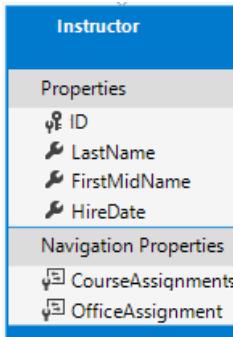
Display 特性

`Display` 特性指定文本框的标题栏应为“`FirstName`”、“`LastName`”、“`FullName`”和“`EnrollmentDate`”。标题栏默认不使用空格分隔词语，如“`Lastname`”。

FullName 计算属性

`FullName` 是计算属性，可返回通过串联两个其他属性创建的值。`FullName` 不能设置并且仅具有一个 `get` 访问器。数据库中不会创建任何 `FullName` 列。

创建 Instructor 实体



用以下代码创建 Models/Instructor.cs：

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

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

请注意，`Student` 和 `Instructor` 实体中具有几个相同属性。本系列后面的“实现继承”教程将重构此代码以消除冗余。

一行可包含多个特性。可按如下方式编写 `HireDate` 特性：

```
[DataType(DataType.Date), Display(Name = "Hire Date"), DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

CourseAssignments 和 OfficeAssignment 导航属性

`CourseAssignments` 和 `OfficeAssignment` 是导航属性。

一名讲师可以教授任意数量的课程，因此 `CourseAssignments` 定义为集合。

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

如果导航属性包含多个实体：

- 它必须是可在其中添加、删除和更新实体的列表类型。

导航属性类型包括：

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

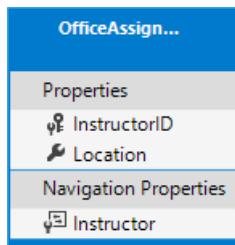
如果指定了 `ICollection<T>`，EF Core 会默认创建 `HashSet<T>` 集合。

`CourseAssignment` 实体在“多对多关系”部分进行介绍。

Contoso University 业务规则规定一名讲师最多可获得一间办公室。`OfficeAssignment` 属性包含一个 `OfficeAssignment` 实体。如果未分配办公室，则 `OfficeAssignment` 为 NULL。

```
public OfficeAssignment OfficeAssignment { get; set; }
```

创建 OfficeAssignment 实体



用以下代码创建 Models/OfficeAssignment.cs：

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

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

Key 特性

`[Key]` 特性用于在属性名不是 `classnameID` 或 `ID` 时将属性标识为主键 (PK)。

`Instructor` 和 `OfficeAssignment` 实体之间存在一对零或一关系。仅当与分配到办公室的讲师之间建立关系时才存在办公室分配。`OfficeAssignment` PK 也是其到 `Instructor` 实体的外键 (FK)。EF Core 无法自动将

`InstructorID` 识别为 `OfficeAssignment` 的 PK, 因为:

- `InstructorID` 不遵循 ID 或 `classnameID` 命名约定。

因此, `Key` 特性用于将 `InstructorID` 识别为 PK:

```
[Key]  
public int InstructorID { get; set; }
```

默认情况下, EF Core 将键视为非数据库生成, 因为该列面向的是识别关系。

Instructor 导航属性

`Instructor` 实体的 `OfficeAssignment` 导航属性可以为 NULL, 因为:

- 引用类型(例如, 类可以为 NULL)。
- 一名讲师可能没有办公室分配。

`OfficeAssignment` 实体具有不可为 NULL 的 `Instructor` 导航属性, 因为:

- `InstructorID` 不可为 NULL。
- 没有讲师则不可能存在办公室分配。

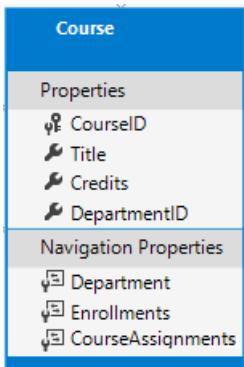
当 `Instructor` 实体具有相关 `OfficeAssignment` 实体时, 每个实体都具有对其导航属性中的另一个实体的引用。

`[Required]` 特性可以应用于 `Instructor` 导航属性:

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

上面的代码指定必须存在相关的讲师。上面的代码没有必要, 因为 `InstructorID` 外键(也是 PK)不可为 NULL。

修改 Course 实体



用以下代码更新 `Models/Course.cs`:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}

```

`Course` 实体具有外键 (FK) 属性 `DepartmentID`。`DepartmentID` 指向相关的 `Department` 实体。`Course` 实体具有 `Department` 导航属性。

当数据模型具有相关实体的导航属性时，EF Core 不要求此模型具有 FK 属性。

EF Core 可在数据库中的任何所需位置自动创建 FK。EF Core 为自动创建的 FK 创建**阴影属性**。数据模型中包含 FK 后可使更新更简单和更高效。例如，假设某个模型中不包含 FK 属性 `DepartmentID`。当提取 `Course` 实体进行编辑时：

- 如果未显式加载 `Department` 实体，则该实体将为 NULL。
- 若要更新 `Course` 实体，则必须先提取 `Department` 实体。

如果数据模型中包含 FK 属性 `DepartmentID`，则无需在更新前提取 `Department` 实体。

DatabaseGenerated 特性

`[DatabaseGenerated(DatabaseGeneratedOption.None)]` 特性指定 PK 由应用程序提供而不是由数据库生成。

```

[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }

```

默认情况下，EF Core 假定 PK 值由数据库生成。由数据库生成 PK 值通常是最佳方法。`Course` 实体的 PK 由用户指定。例如，对于课程编号，数学系可以使用 1000 系列的编号，英语系可以使用 2000 系列的编号。

`DatabaseGenerated` 特性还可用于生成默认值。例如，数据库可以自动生成日期字段以记录数据行的创建或更新日期。有关详细信息，请参阅[生成的属性](#)。

外键和导航属性

`Course` 实体中的外键 (FK) 属性和导航属性可反映以下关系：

课程将分配到一个系，因此将存在 `DepartmentID` FK 和 `Department` 导航属性。

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

参与一门课程的学生数量不定，因此 `Enrollments` 导航属性是一个集合：

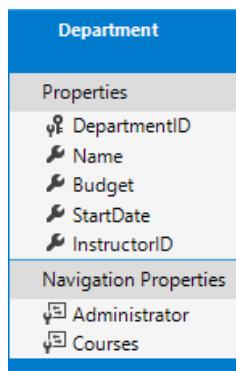
```
public ICollection<Enrollment> Enrollments { get; set; }
```

一门课程可能由多位讲师讲授，因此 `CourseAssignments` 导航属性是一个集合：

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` 在[后文](#)介绍。

创建 Department 实体



用以下代码创建 Models/Department.cs：

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

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

Column 特性

`Column` 特性以前用于更改列名映射。在 `Department` 实体的代码中, `Column` 特性用于更改 SQL 数据类型映射。

`Budget` 列通过数据库中的 SQL Server 货币类型进行定义:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

通常不需要列映射。EF Core 通常基于属性的 CLR 类型选择相应的 SQL Server 数据类型。CLR `decimal` 类型会映射到 SQL Server `decimal` 类型。`Budget` 用于货币, 但货币数据类型更适合货币。

外键和导航属性

FK 和导航属性可反映以下关系:

- 一个系可能有也可能没有管理员。
- 管理员始终由讲师担任。因此, `InstructorID` 属性作为到 `Instructor` 实体的 FK 包含在其中。

导航属性名为 `Administrator`, 但其中包含 `Instructor` 实体:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

上面代码中的问号 (?) 指定属性可以为 NULL。

一个系可以有多门课程, 因此存在 `Course` 导航属性:

```
public ICollection<Course> Courses { get; set; }
```

注意:按照约定, EF Core 能针对不可为 NULL 的 FK 和多对多关系启用级联删除。级联删除可能导致形成循环级联删除规则。循环级联删除规则会在添加迁移时引发异常。

例如, 如果未将 `Department.InstructorID` 属性定义为可以为 NULL:

- EF Core 会配置将在删除系时删除讲师的级联删除规则。
- 在删除系时删除讲师并不是预期行为。

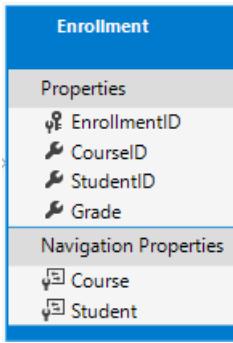
如果业务规则要求 `InstructorID` 属性不可为 NULL, 请使用以下 Fluent API 语句:

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

上面的代码会针对“系-讲师”关系禁用级联删除。

更新 Enrollment 实体

一份注册记录面向一名学生所注册的一门课程。



用以下代码更新 *Models/Enrollment.cs*:

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

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

外键和导航属性

FK 属性和导航属性可反映以下关系:

注册记录面向一门课程, 因此存在 `CourseID` FK 属性和 `Course` 导航属性:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

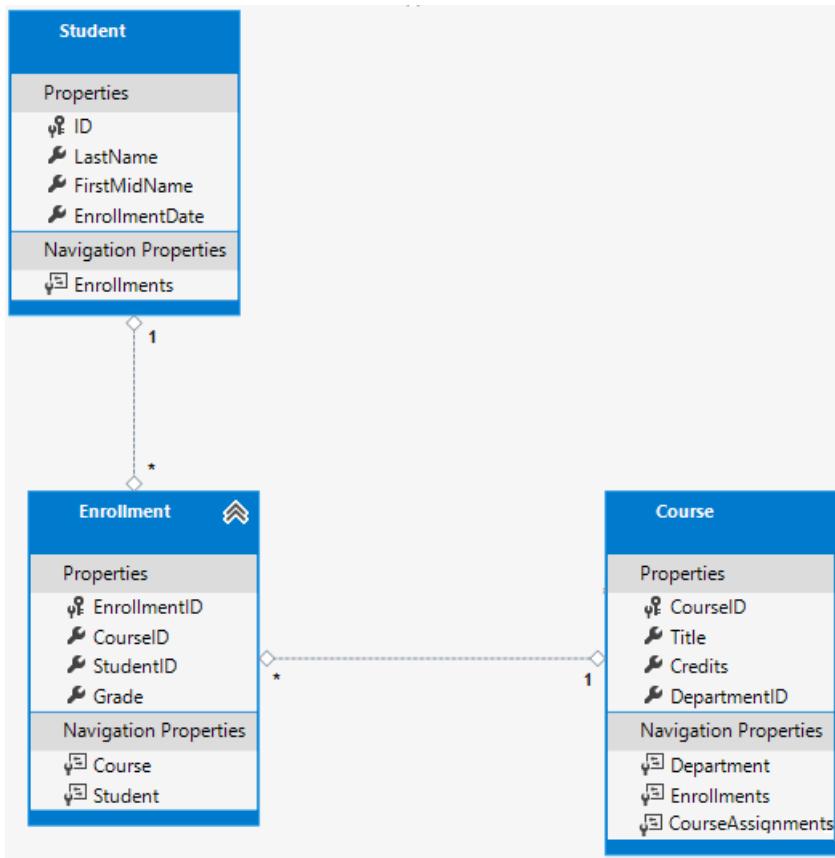
一份注册记录面向一名学生, 因此存在 `StudentID` FK 属性和 `Student` 导航属性:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

多对多关系

`Student` 和 `Course` 实体之间存在多对多关系。`Enrollment` 实体充当数据库中“具有有效负载”的多对多联接表。“具有有效负载”表示 `Enrollment` 表除了联接表的 FK 外还包含其他数据(本教程中为 PK 和 `Grade`)。

下图显示这些关系在实体关系图中的外观。(此关系图通过适用于 EF 6.x 的 EF Power Tools 生成。本教程不介绍如何创建此关系图。)



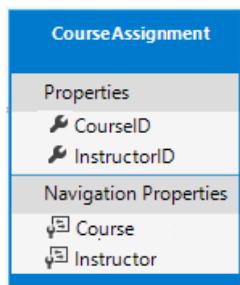
每条关系线的一端显示 1，另一端显示星号 (*)，这表示一对多关系。

如果 **Enrollment** 表不包含年级信息，则它只需包含两个 FK(**CourseID** 和 **StudentID**)。无有效负载的多对多联接表有时称为纯联接表 (PJT)。

Instructor 和 **Course** 实体具有使用纯联接表的多对多关系。

注意：EF 6.x 支持多对多关系的隐式联接表，但 EF Core 不支持。有关详细信息，请参阅 [EF Core 2.0 中的多对多关系](#)。

CourseAssignment 实体



用以下代码创建 Models/CourseAssignment.cs：

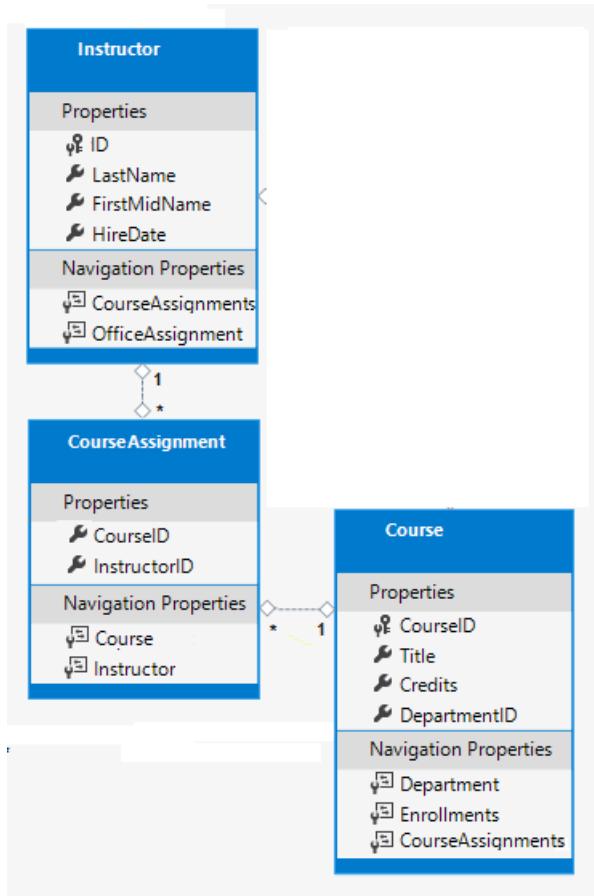
```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}

```

讲师-课程



讲师-课程的多对多关系：

- 要求必须用实体集表示联接表。
- 为纯联接表(无有效负载的表)。

常规做法是将联接实体命名为 `EntityName1EntityName2`。例如，使用此模式的“讲师-课程”联接表是 `CourseInstructor`。但是，我们建议使用可描述关系的名称。

数据模型开始时很简单，其内容会逐渐增加。无有效负载联接 (PJT) 通常会发展为包含有效负载。该名称以描述性实体名称开始，因此不需要随联接表更改而更改。理想情况下，联接实体在业务域中可能具有自己的自带名称(可能是单个字)。例如，可以使用名为“比率”的联接实体链接“账目”和“客户”。对于“讲师-课程”多对多关系，建议使用 `CourseAssignment` 而不是 `CourseInstructor`。

组合键

FK 不能为 NULL。`CourseAssignment` 中的两个 FK(`InstructorID` 和 `CourseID`)共同唯一标识 `CourseAssignment`

表的每一行。`CourseAssignment` 不需要专用的 PK。`InstructorID` 和 `CourseID` 属性充当组合 PK。使用 Fluent API 是向 EF Core 指定组合 PK 的唯一方法。下一部分演示如何配置组合 PK。

组合键可确保：

- 允许一门课程对应多行。
- 允许一名讲师对应多行。
- 不允许相同的讲师和课程对应多行。

`Enrollment` 联接实体定义其自己的 PK，因此可能会出现此类重复。若要防止此类重复：

- 请在 FK 字段上添加唯一索引，或
- 配置具有主要组合键(与 `CourseAssignment` 类似)的 `Enrollment`。有关详细信息，请参阅[索引](#)。

更新数据库上下文

将以下突出显示的代码添加到 Data/SchoolContext.cs：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

上面的代码添加新实体并配置 `CourseAssignment` 实体的组合 PK。

用 Fluent API 替代特性

上面代码中的 `OnModelCreating` 方法使用 Fluent API 配置 EF Core 行为。API 称为“Fluent”，因为它通常在将一系列方法调用连接成单个语句后才能使用。[下面的代码](#) 是 Fluent API 的示例：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

在本教程中，Fluent API 仅用于不能通过特性完成的数据库映射。但是，Fluent API 可以指定可通过特性完成的大多数格式设置、验证和映射规则。

`MinimumLength` 等特性不能通过 Fluent API 应用。`MinimumLength` 不会更改架构，它仅应用最小长度验证规则。

某些开发者倾向于仅使用 Fluent API 以保持实体类的“纯净”。特性和 Fluent API 可以相互混合。某些配置只能通过 Fluent API 完成（指定组合 PK）。有些配置只能通过特性完成（`MinimumLength`）。使用 Fluent API 或特性的建议做法：

- 选择以下两种方法之一。
- 尽可能以前后一致的方法使用所选的方法。

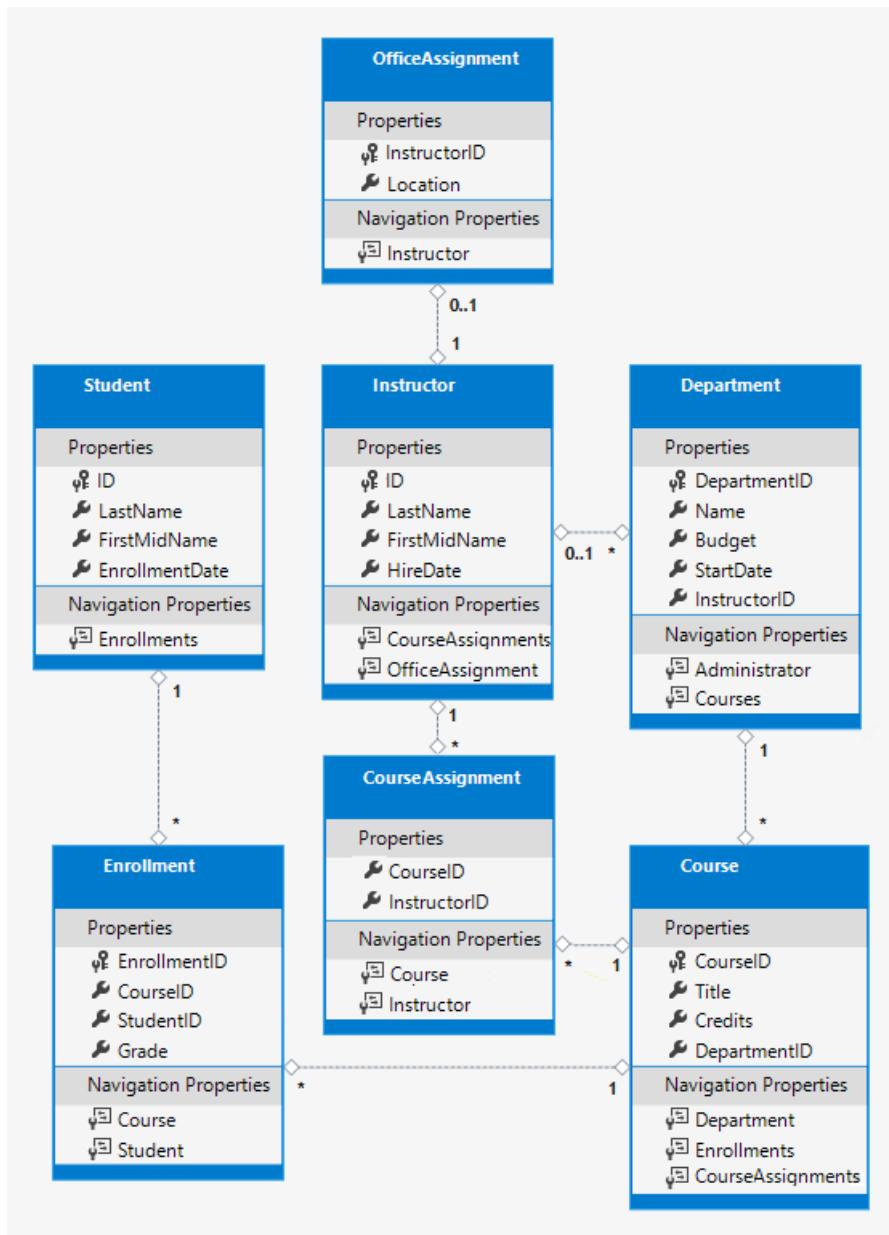
本教程中使用的某些特性可用于：

- 仅限验证（例如，`MinimumLength`）。
- 仅限 EF Core 配置（例如，`HasKey`）。
- 验证和 EF Core 配置（例如，`[StringLength(50)]`）。

有关特性和 Fluent API 的详细信息，请参阅[配置方法](#)。

显示关系的实体关系图

下图显示 EF Power Tools 针对已完成的学校模型创建的关系图。



上面的关系图显示：

- 几条一对多关系线(1 到 *)。
- `Instructor` 和 `OfficeAssignment` 实体之间的一对零或一关系线(1 到 0..1)。
- `Instructor` 和 `Department` 实体之间的零或一到多关系线(0..1 到 *)。

使用测试数据为数据库设定种子

更新 Data/DbInitializer.cs 中的代码：

```

using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
        }
    }
}
  
```

```

// Look for any students.
if (context.Students.Any())
{
    return; // DB has been seeded
}

var students = new Student[]
{
    new Student { FirstMidName = "Carson", LastName = "Alexander",
        EnrollmentDate = DateTime.Parse("2010-09-01") },
    new Student { FirstMidName = "Meredith", LastName = "Alonso",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Arturo", LastName = "Anand",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Gytis", LastName = "Barzdukas",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Yan", LastName = "Li",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Peggy", LastName = "Justice",
        EnrollmentDate = DateTime.Parse("2011-09-01") },
    new Student { FirstMidName = "Laura", LastName = "Norman",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Nino", LastName = "Olivetto",
        EnrollmentDate = DateTime.Parse("2005-09-01") }
};

foreach (Student s in students)
{
    context.Students.Add(s);
}
context.SaveChanges();

var instructors = new Instructor[]
{
    new Instructor { FirstMidName = "Kim", LastName = "Abercrombie",
        HireDate = DateTime.Parse("1995-03-11") },
    new Instructor { FirstMidName = "Fadi", LastName = "Fakhouri",
        HireDate = DateTime.Parse("2002-07-06") },
    new Instructor { FirstMidName = "Roger", LastName = "Harui",
        HireDate = DateTime.Parse("1998-07-01") },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
        HireDate = DateTime.Parse("2001-01-15") },
    new Instructor { FirstMidName = "Roger", LastName = "Zheng",
        HireDate = DateTime.Parse("2004-02-12") }
};

foreach (Instructor i in instructors)
{
    context.Instructors.Add(i);
}
context.SaveChanges();

var departments = new Department[]
{
    new Department { Name = "English", Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Abercrombie").ID },
    new Department { Name = "Mathematics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID },
    new Department { Name = "Engineering", Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID },
    new Department { Name = "Economics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)

```

```

{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
};

```

```

        '',
new CourseAssignment {
    CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
    InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
},
new CourseAssignment {
    CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
    InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
},
new CourseAssignment {
    CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
    InstructorID = instructors.Single(i => i.LastName == "Harui").ID
},
new CourseAssignment {
    CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
    InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
},
new CourseAssignment {
    CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
    InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
},
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        Grade = Grade.A
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        Grade = Grade.C
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
        Grade = Grade.B
    }
}

```

```

        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Li").ID,
            CourseID = courses.Single(c => c.Title == "Composition").CourseID,
            Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Justice").ID,
            CourseID = courses.Single(c => c.Title == "Literature").CourseID,
            Grade = Grade.B
        }
    }

    foreach (Enrollment e in enrollments)
    {
        var enrollmentInDataBase = context.Enrollments.Where(
            s =>
                s.Student.ID == e.StudentID &&
                s.Course.CourseID == e.CourseID).SingleOrDefault();
        if (enrollmentInDataBase == null)
        {
            context.Enrollments.Add(e);
        }
    }
    context.SaveChanges();
}
}

```

前面的代码为新实体提供种子数据。大多数此类代码会创建新实体对象并加载示例数据。示例数据用于测试。前面的代码将创建以下多对多关系：

- Enrollments
 - CourseAssignment

注意:EF Core 2.1 将支持数据种子设定。

添加迁移

生成项目。在项目文件夹中打开命令窗口并输入以下命令：

```
dotnet ef migrations add ComplexDataModel
```

前面的命令显示可能存在数据丢失的相关警告。

An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'

如果运行 `database update` 命令，则会生成以下错误：

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

当将现有数据与迁移一起运行时，可能存在不满足现有数据的 FK 约束。本教程将创建新数据库，这样便不会出现任何 FK 约束冲突。请参阅[通过旧数据修复外键约束](#)，获取有关如何在当前数据库上修复 FK 冲突的说明。

更改连接字符串并更新数据库

已更新 `DbInitializer` 中的代码将为新实体添加种子数据。若要强制 EF Core 创建新的空数据库，请执行以下操作：

- 将 `appsettings.json` 中的数据库连接字符串名称更改为 `ContosoUniversity3`。新名称必须是未在计算机上使用过的名称。

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Server=  
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"  
    },  
}
```

- 或者使用以下项删除数据库：

- SQL Server 对象资源管理器 (SSOX)。**

- `database drop` CLI 命令：

```
dotnet ef database drop
```

在命令窗口中运行 `database update`：

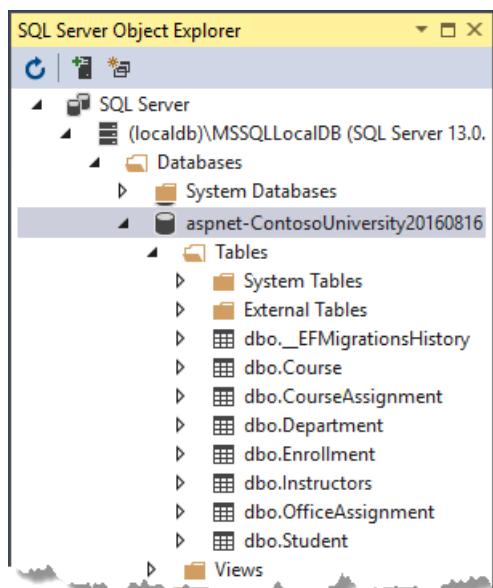
```
dotnet ef database update
```

上面的命令将运行所有迁移。

运行应用。运行应用后将运行 `DbInitializer.Initialize` 方法。`DbInitializer.Initialize` 将填充新数据库。

在 SSOX 中打开数据库：

- 展开“表”节点。随后将显示出已创建的表。
- 如果之前已打开过 SSOX，请单击“刷新”按钮。



查看 CourseAssignment 表：

- 右键单击 CourseAssignment 表，然后选择“查看数据”。
- 验证 CourseAssignment 表包含数据。

The screenshot shows the SSMS Data grid with the title bar "ContosoUniversity" and the tab "dbo.CourseAssignment [Data]". The table has two columns: "CourseID" and "InstructorID". The data rows are:

	CourseID	InstructorID
▶	2021	1
	2042	1
	1045	2
	1050	3
	3141	3
	1050	4
	4022	5
*	4041	5
	NULL	NULL

通过旧数据修复外键约束

本部分是可选的。

当将现有数据与迁移一起运行时，可能存在不满足现有数据的 FK 约束。使用生产数据时，必须采取步骤来迁移现有数据。本部分提供修复 FK 约束冲突的示例。务必在备份后执行这些代码更改。如果已完成上述部分并更新数据库，则不要执行这些代码更改。

{timestamp}_ComplexDataModel.cs 文件包含以下代码：

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

上面的代码将向 Course 表添加不可为 NULL 的 DepartmentID FK。前面教程中的数据库在 Course 中包含行，以便迁移时不会更新表。

若要使 ComplexDataModel 迁移可与现有数据搭配运行：

- 请更改代码以便为新列 (DepartmentID) 赋予默认值。
- 创建名为“临时”的虚拟系来充当默认的系。

修复外键约束

更新 ComplexDataModel 类 Up 方法：

- 打开 {timestamp}_ComplexDataModel.cs 文件。
- 对将 DepartmentID 列添加到 Course 表的代码行添加注释。

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

添加以下突出显示的代码。新代码在 `.CreateTable(name: "Department")` 块后 :[!code-csharp]

经过上面的更改, `Course` 行将在 `ComplexDataModel Up` 方法运行后与“临时”系建立联系。

生产应用可能:

- 包含用于将 `Department` 行和相关 `Course` 行添加到新 `Department` 行的代码或脚本。
- 不会使用“临时”系或 `Course.DepartmentID` 的默认值。

下一教程将介绍相关数据。

[上一页](#) [下一页](#)

ASP.NET Core 中的 Razor 页面和 EF Core - 读取相关数据 - 第 6 个教程(共 8 个)

2018/5/14 • 17 min to read • [Edit Online](#)

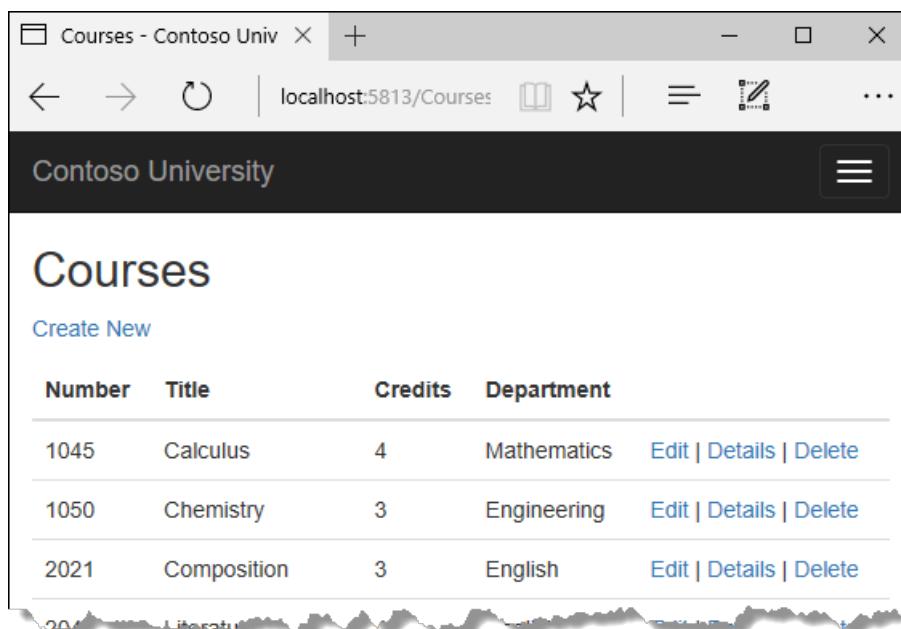
作者:Tom Dykstra、Jon P Smith 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

在本教程中, 将读取和显示相关数据。相关数据为 EF Core 加载到导航属性中的数据。

如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

下图显示了本教程中已完成的页面:



Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Contoso University

Instructors

Create New

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

相关数据的预先加载、显式加载和延迟加载

EF Core 可采用多种方式将相关数据加载到实体的导航属性中：

- **预先加载**。预先加载是指对查询某类型的实体时一并加载相关实体。读取实体时，会检索其相关数据。此时通常会出现单一联接查询，检索所有必需数据。EF Core 将针对预先加载的某些类型发出多个查询。与存在单一查询的 EF6 中的某些查询相比，发出多个查询可能更有效。预先加载通过 `Include` 和 `ThenInclude` 方法进行指定。

```

var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

Query: all Department entities
and related Course entities

当包含集合导航时，预先加载会发送多个查询：

- 一个查询用于主查询
- 一个查询用于加载树中每个集合“边缘”。
- 使用 `Load` 的单独查询：可在单独的查询中检索数据，EF Core 会“修复”导航属性。“修复”是指 EF Core 自动填充导航属性。与预先加载相比，使用 `Load` 的单独查询更像是显式加载。

```

var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

Query: all Department rows

Query: Course rows related to Department d

请注意：EF Core 会将导航属性自动“修复”为之前加载到上下文实例中的任何其他实体。即使导航属性的数据非显式包含在内，但如果先前加载了部分或所有相关实体，则仍可能填充该属性。

- 显式加载。首次读取实体时，不检索相关数据。必须编写代码才能在需要时检索相关数据。使用单独查询进行显式加载时，会向数据库发送多个查询。该代码通过显式加载指定要加载的导航属性。使用 `Load` 方法进行显式加载。例如：

```

var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

Query: all Department rows

Query: Course rows related to Department d

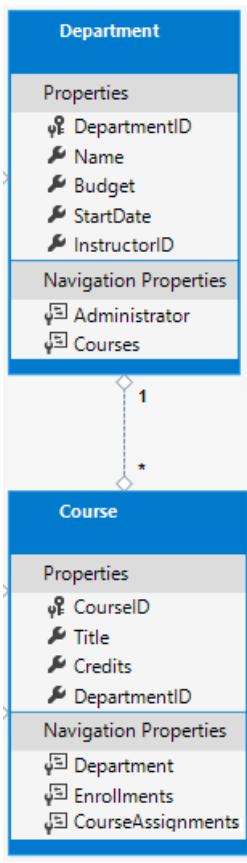
- 延迟加载。EF Core 当前不支持延迟加载。首次读取实体时，不检索相关数据。首次访问导航属性时，会自动检索该导航属性所需的数据。首次访问导航属性时，都会向数据库发送一个查询。
- `Select` 运算符仅加载所需的相关数据。

创建显示院系名称的“课程”页

课程实体包括一个带 `Department` 实体的导航属性。`Department` 实体包含要分配课程的院系。

要在课程列表中显示已分配院系的名称：

- 从 `Department` 实体中获取 `Name` 属性。
- `Department` 实体来自于 `Course.Department` 导航属性。



为课程模型创建基架

- 退出 Visual Studio。
- 打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。
- 运行下面的命令：

```
dotnet aspnet-codegenerator razorpage -m Course -dc SchoolContext -udl -outDir Pages\Courses --referenceScriptLibraries
```

上述命令为 `Course` 模型创建基架。在 Visual Studio 中打开项目。

生成项目。此版本生成如下错误：

```
1>Pages/Courses/Index.cshtml.cs(26,37,26,43): error CS1061: 'SchoolContext' does not contain a definition for 'Course' and no extension method 'Course' accepting a first argument of type 'SchoolContext' could be found (are you missing a using directive or an assembly reference?)
```

将 `_context.Course` 全局更改为 `_context.Courses` (即向 `Course` 添加一个"s")。找到并更新 7 个匹配项。

打开 Pages/Courses/Index.cshtml.cs 并检查 `OnGetAsync` 方法。基架引擎为 `Department` 导航属性指定了预先加载。`Include` 方法指定预先加载。

运行应用并选择“课程”链接。院系列表显示 `DepartmentID` (该项无用)。

使用以下代码更新 `OnGetAsync` 方法：

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

上述代码添加了 `AsNoTracking`。由于未跟踪返回的实体，因此 `AsNoTracking` 提升了性能。未跟踪实体，因为未在当前上下文中更新这些实体。

使用以下突出显示的标记更新 Views/Courses/Index.cshtml：

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel
 @{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

<p>
    <a asp-page="TestCreate">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-page=".Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page=".Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page=".Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

对基架代码进行了以下更改：

- 将标题从“索引”更改为“课程”。
- 添加了显示 `CourseID` 属性值的“数字”列。默认情况下，不针对主键进行架构，因为对最终用户而言，它们

通常没有意义。但在此情况下主键是有意义的。

- 更改“院系”列，显示院系名称。该代码显示已加载到 `Department` 导航属性中的 `Department` 实体的 `Name` 属性：

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

运行应用并选择“课程”选项卡，查看包含系名称的列表。

Number	Title	Credits	Department	
1045	Calculus	4	Mathematics	Edit Details Delete
1050	Chemistry	3	Engineering	Edit Details Delete
2021	Composition	3	English	Edit Details Delete

使用 `Select` 加载相关数据

`OnGetAsync` 方法使用 `Include` 方法加载相关数据：

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

`Select` 运算符仅加载所需的相关数据。对于单个项(如 `Department.Name`)，它使用 SQL INNER JOIN。对于集合，它使用另一个数据库访问，但集合上的 `Include` 运算符也是如此。

以下代码使用 `Select` 方法加载相关数据：

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
        .Select(p => new CourseViewModel
        {
            CourseID = p.CourseID,
            Title = p.Title,
            Credits = p.Credits,
            DepartmentName = p.Department.Name
        }).ToListAsync();
}
```

CourseViewModel :

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

有关完整示例的信息，请参阅 [IndexSelect.cshtml](#) 和 [IndexSelect.cshtml.cs](#)。

创建显示“课程”和“注册”的“讲师”页

在本部分中，将创建“讲师”页。

Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Instructors

Create New

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

该页面通过以下方式读取和显示相关数据：

- 讲师列表显示 `OfficeAssignment` 实体(上图中的办公室)的相关数据。`Instructor` 和 `OfficeAssignment` 实体之间存在一对零或一的关系。预先加载适用于 `OfficeAssignment` 实体。需要显示相关数据时，预先加载通常更有效。在此情况下，会显示讲师的办公室分配。
- 当用户选择一名讲师(上图中的 Harui)时，显示相关的 `Course` 实体。`Instructor` 和 `Course` 实体之间存在多对多关系。对 `Course` 实体及其相关的 `Department` 实体使用预先加载。这种情况下，单独查询可能更有效，因为仅需显示所选讲师的课程。此示例演示如何在位于导航实体内的实体中预先加载这些导航实体。

- 当用户选择一门课程(上图中的化学)时, 显示 `Enrollments` 实体的相关数据。上图中显示了学生姓名和成绩。`Course` 和 `Enrollment` 实体之间存在一对多的关系。

创建“讲师索引”视图的视图模型

“讲师”页显示来自三个不同表格的数据。创建一个视图模型, 该模型中包含表示三个表格的三个实体。

在 SchoolViewModels 文件夹中, 使用以下代码创建 InstructorIndexData.cs:

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

为讲师模型创建基架

- 退出 Visual Studio。
- 打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。
- 运行下面的命令:

```
dotnet aspnet-codegenerator razorpage -m Instructor -dc SchoolContext -udl -outDir Pages\Instructors --referenceScriptLibraries
```

上述命令为 `Instructor` 模型创建基架。在 Visual Studio 中打开项目。

生成项目。此版本生成错误。

将 `_context.Instructor` 全局更改为 `_context.Instructors` (即向 `Instructor` 添加一个“s”)。找到并更新 7 个匹配项。

运行应用并导航到“讲师”页。

将 Pages/Instructors/Index.cshtml.cs 替换为以下代码:

```

using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels; // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                .AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}

```

`OnGetAsync` 方法接受所选讲师 ID 的可选路由数据。

检查 Pages/Instructors/Index.cshtml 页上的查询：

```

Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

查询包括两项内容：

- `OfficeAssignment`：在 [讲师视图](#) 中显示。
- `CourseAssignments`：课程的教学内容。

更新“讲师索引”页

使用以下标记更新 Pages/Instructors/Index.cshtml：

```

@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

 @{
     ViewData["Title"] = "Instructors";
 }

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @: @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page=".Index" asp-route-id="@item.ID">Select</a> | 
                    <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> | 
                    <a asp-page=".Details" asp-route-id="@item.ID">Details</a> | 
                    <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

上述标记进行以下更改：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int?}"`。`"{id:int?}"` 是一个路由模板。路由模板将 URL 中的整数查询字符串更改为路由数据。例如，单击仅具有 `@page` 指令的讲师的“选择”链接将生成如下 URL：

```
http://localhost:1234/Instructors?id=2
```

当页面指令是 `@page "{id:int?}"` 时，之前的 URL 为：

```
http://localhost:1234/Instructors/2
```

- 页标题为“讲师”。
- 添加了仅在 `item.OfficeAssignment` 不为 null 时才显示 `item.OfficeAssignment.Location` 的“办公室”列。由于这是一对零或一的关系，因此可能没有相关的 `OfficeAssignment` 实体。

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- 添加了显示每位讲师所授课程的“课程”列。有关此 Razor 语法的详细信息，请参阅[使用 @:](#) 进行显式行转换。
- 添加了向所选讲师的 `tr` 元素中动态添加 `class="success"` 的代码。此时会使用 Bootstrap 类为所选行设置背景色。

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- 添加了标记为“选择”的新的超链接。该链接将所选讲师的 ID 发送给 `Index` 方法并设置背景色。

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

运行应用并选择“讲师”选项卡。该页显示来自相关 `OfficeAssignment` 实体的 `Location` (办公室)。如果 `OfficeAssignment`` 为 NULL，则显示空白表格单元格。

单击“选择”链接。随即更改行样式。

添加由所选讲师教授的课程

将 Pages/Instructors/Index.cshtml.cs 中的 `OnGetAsync` 方法替换为以下代码：

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

检查更新后的查询：

```

Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

先前查询添加了 `Department` 实体。

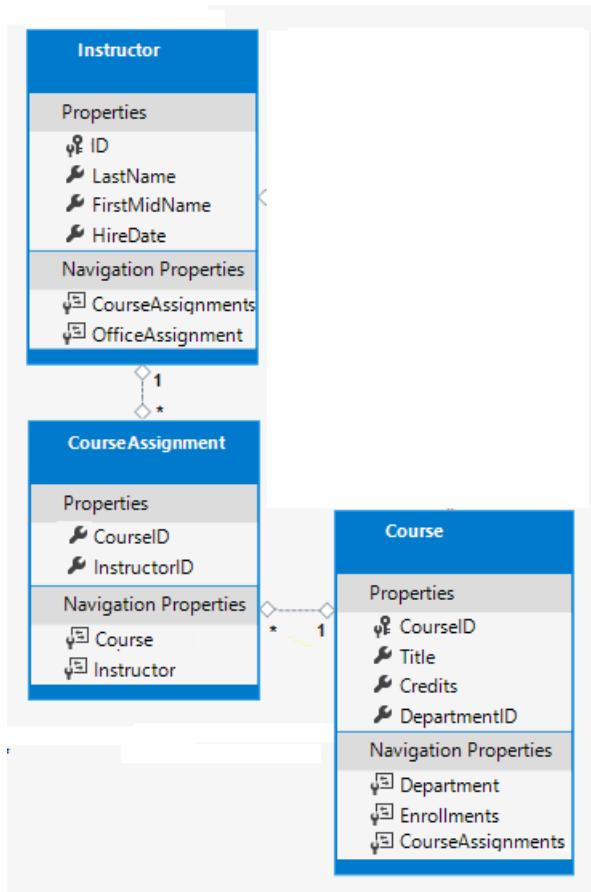
选择讲师时 (`id != null`)，将执行以下代码。从视图模型中的讲师列表检索所选讲师。向视图模型的 `Courses` 属性加载来自讲师 `CourseAssignments` 导航属性的 `Course` 实体。

```

if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Where(
        i => i.ID == id.Value).Single();
    Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
}

```

`Where` 方法返回一个集合。在前面的 `Where` 方法中，仅返回单个 `Instructor` 实体。`Single` 方法将集合转换为单个 `Instructor` 实体。`Instructor` 实体提供对 `CourseAssignments` 属性的访问。`CourseAssignments` 提供对相关 `Course` 实体的访问。



当集合仅包含一个项时，集合使用 `Single` 方法。如果集合为空或包含多个项，`Single` 方法会引发异常。还可使用 `SingleOrDefault`，该方式在集合为空时返回默认值(本例中为 `null`)。在空集合上使用 `SingleOrDefault`：

- 引发异常(因为尝试在空引用上找到 `Courses` 属性)。
- 异常信息不太能清楚指出问题原因。

选中课程时，视图模型的 `Enrollments` 属性将填充以下代码：

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

在 Pages/Courses/Index.cshtml Razor 页面末尾添加以下标记：

```
<a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
}
</tbody>
</table>

@if (Model.Instructor.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "OnGetAsync",
                        new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td> <td>
                    @item.Department.Name
                </td>
            </tr>
        }
    </table>
}
```

上述标记显示选中某讲师时与该讲师相关的课程列表。

测试应用。单击讲师页面上的“选择”链接。

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

显示学生数据

在本部分中，更新应用以显示所选课程的学生数据。

使用以下代码在 Pages/Instructors/Index.cshtml.cs 中更新 `OnGetAsync` 方法中的查询：

```
Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

更新 Pages/Instructors/Index.cshtml。在文件末尾添加以下标记：

```
@if (Model.Instructor.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

上述标记显示已注册所选课程的学生列表。

刷新页面并选择讲师。选择一门课程，查看已注册的学生及其成绩列表。

Instructors - Contoso University

localhost:1234/Instructors/3?courseID=1050&action=OnGetAsync

Instructors

Create New

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete
Kapoor	Candace	2001-01-15	Thompson 304	1050 Chemistry	Select Edit Details Delete
Zheng	Roger	2004-02-12		4022 Microeconomics 4041 Macroeconomics	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	1050	Chemistry	Engineering
Select	3141	Trigonometry	Mathematics

Students Enrolled in Selected Course

Name	Grade
Alexander, Carson	A
Anand, Arturo	No grade
Barzdukas, Gytis	B

© 2017 - Contoso University

使用 Single 方法

Single 方法可在 Where 条件中进行传递，无需分别调用 Where 方法：

```

public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();
}

if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Single(
        i => i.ID == id.Value);
    Instructor.Courses = instructor.CourseAssignments.Select(
        s => s.Course);
}

if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Single(
        x => x.CourseID == courseID).Enrollments;
}
}

```

使用 `Where` 时，前面的 `Single` 方法不适用。一些开发人员更喜欢 `Single` 方法样式。

显式加载

当前代码为 `Enrollments` 和 `Students` 指定预先加载：

```

Instructor.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

假设用户几乎不希望课程中显示注册情况。在此情况下，可仅在请求时加载注册数据进行优化。在本部分中，会更新 `OnGetAsync` 以使用 `Enrollments` 和 `Students` 的显式加载。

使用以下代码更新 `OnGetAsync`：

```

public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        // .Include(i => i.CourseAssignments)
        //     .ThenInclude(i => i.Course)
        //         .ThenInclude(i => i.Enrollments)
        //             .ThenInclude(i => i.Student)
        // .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}

```

上述代码取消针对注册和学生数据的 ThenInclude 方法调用。如果已选中课程，则突出显示的代码会检索：

- 所选课程的 Enrollment 实体。
- 每个 Enrollment 的 Student 实体。

请注意，上述代码为 `.AsNoTracking()` 加上注释。对于跟踪的实体，仅可显式加载导航属性。

测试应用。对用户而言，该应用的行为与上一版本相同。

下一个教程将介绍如何更新相关数据。

ASP.NET Core 中的 Razor 页面和 EF Core - 更新相关数据 - 第 7 个教程(共 8 个)

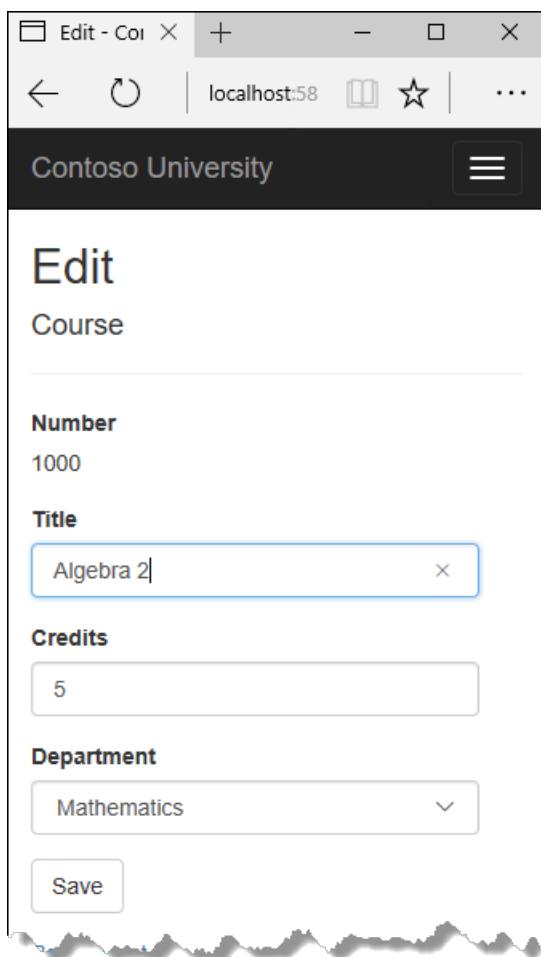
2018/5/14 • 17 min to read • [Edit Online](#)

作者:Tom Dykstra 和 Rick Anderson

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

本教程演示如何更新相关数据。如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

下图显示了部分已完成页面。



The screenshot shows a web browser window titled "Edit - Contoso Universit" with the URL "localhost:5813/Instruct". The page has a dark header with the text "Contoso University" and a three-line menu icon. The main content area is titled "Edit" and has a sub-section "Instructor". It contains several input fields: "Last Name" with the value "Abercrombie", "First Name" with the value "Kim", "Hire Date" with the value "3/11/1995", and "Office Location" with the value "44/3P". Below these are several checkboxes grouped by course ID: 1000 Algebra 2, 1045 Calculus, 1050 Chemistry, 2021 Composition (checked), 2042 Literature (checked), 3141 Trigonometry, 4022 Microeconomics, and 4041 Macroeconomics. At the bottom left is a "Save" button.

查看并测试“创建”和“编辑”课程页。创建新课程。系按其主键(一个整数)进行选择，而不是按名称。编辑新课程。
完成测试后，请删除新课程。

创建基类以共享通用代码

“课程/创建”和“课程/编辑”页分别需要一个系名称列表。针对“创建”和“编辑”页创建
Pages/Courses/DepartmentNamePageModel.cshtml.cs 基类：

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                orderby d.Name // Sort by name.
                select d;

            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

上面的代码创建 `SelectList` 以包含系名称列表。如果指定了 `selectedDepartment`，可在 `SelectList` 中选择该系。

“创建”和“编辑”页模型类将派生自 `DepartmentNamePageModel`。

自定义课程页

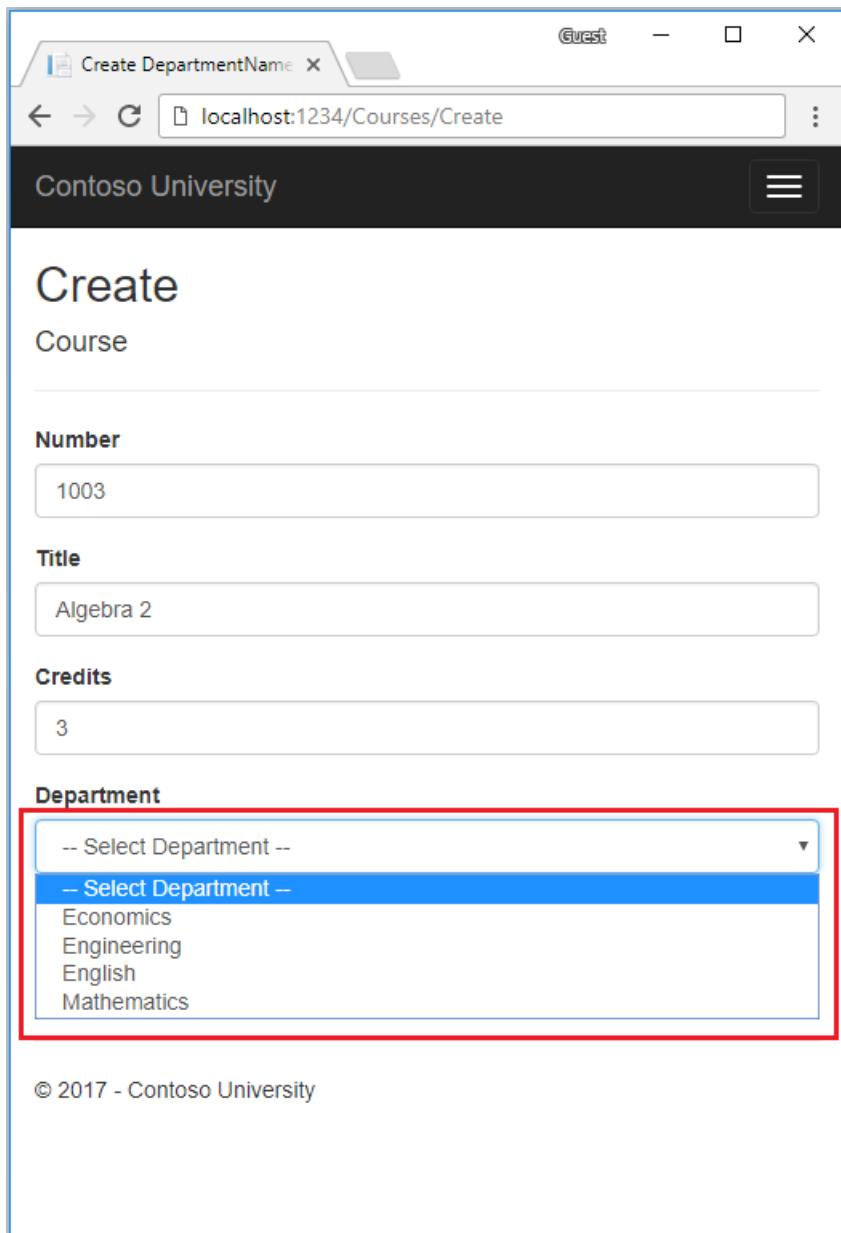
创建新的课程实体时，新实体必须与现有院系有关系。若要在创建课程的同时添加系，则“创建”和“编辑”的基类必须包含用于选择系的下拉列表。该下拉列表设置 `Course.DepartmentID` 外键 (FK) 属性。EF Core 使用 `Course.DepartmentID` FK 加载 `Department` 导航属性。

Screenshot of a web browser showing the "Create" page for a "Course". The URL in the address bar is "localhost:1234/Courses/Create". The page title is "Create DepartmentName".

The form fields are:

- Number**: Input field containing "1003".
- Title**: Input field containing "Algebra 2".
- Credits**: Input field containing "3".
- Department**: A dropdown menu with the following options:
 - Select Department -- (selected)
 - Economics
 - Engineering
 - English
 - Mathematics

At the bottom left, there is a copyright notice: "© 2017 - Contoso University".



使用以下代码更新“创建”页模型：

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                emptyCourse,
                "course", // Prefix for form value.
                s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
    }
}

```

前面的代码：

- 派生自 `DepartmentNamePageModel`。
- 使用 `TryUpdateModelAsync` 防止过多发布。
- 将 `ViewData["DepartmentID"]` 替换为 `DepartmentNameSL` (来自基类)。

`ViewData["DepartmentID"]` 将替换为强类型 `DepartmentNameSL`。建议使用强类型而非弱类型。有关详细信息，请参阅[弱类型数据 \(ViewData 和 ViewBag\)](#)。

更新“课程创建”页

使用以下标记更新 `Pages/Courses/Create.cshtml`:

```

@page
@model ContosoUniversity.Pages.Courses.CreateModel
 @{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control" asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

上述标记进行以下更改：

- 将标题从“DepartmentID”更改为“Department”。
- 将 `"ViewBag.DepartmentID"` 替换为 `DepartmentNameSL`（来自基类）。
- 添加“选择系”选项。此更改将导致呈现“选择系”而不是第一个系。
- 在未选择系时添加验证消息。

Razor 页面使用[选择标记帮助器](#)：

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
        asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

测试“创建”页。“创建”页显示系名称，而不是系 ID。

更新“课程编辑”页。

使用以下代码更新“编辑”页模型：

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

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

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                courseToUpdate,
                "course", // Prefix for form value.
                c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}

```

这些更改与在“创建”页模型中所做的更改相似。在上面的代码中，`PopulateDepartmentsDropDownList` 在系 ID 中传递并将选择下拉列表中指定的系。

使用以下标记更新 Pages/Courses/Edit.cshtml：

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}



## Edit



#### Course



---



<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <input type="hidden" asp-for="Course.CourseID" />
    <div class="form-group">
        <label asp-for="Course.CourseID" class="control-label"></label>
        <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
    </div>
    <div class="form-group">
        <label asp-for="Course.Title" class="control-label"></label>
        <input asp-for="Course.Title" class="form-control" />
        <span asp-validation-for="Course.Title" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Course.Credits" class="control-label"></label>
        <input asp-for="Course.Credits" class="form-control" />
        <span asp-validation-for="Course.Credits" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Course.Department" class="control-label"></label>
        <select asp-for="Course.DepartmentID" class="form-control"
                asp-items="@Model.DepartmentNameSL"></select>
        <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
    </div>
    <div class="form-group">
        <input type="submit" value="Save" class="btn btn-default" />
    </div>
</form>



上述标记进行以下更改：



- 显示课程 ID。通常不显示实体的主键 (PK)。PK 对用户不具有任何意义。在这种情况下，PK 就是课程编号。
- 将标题从“DepartmentID”更改为“Department”。
- 将 "ViewBag.DepartmentID" 替换为 DepartmentNameSL (来自基类)。
- 添加“选择系”选项。此更改将导致呈现“选择系”而不是第一个系。
- 在未选择系时添加验证消息。



该页面包含课程编号的隐藏域 (<input type="hidden">)。添加具有 asp-for="Course.CourseID" 的 <label> 标记帮助器也同样需要隐藏域。用户单击“保存”时，需要 <input type="hidden">，以便在已发布的数据中包括课程编号。


```

测试更新的代码。创建、编辑和删除课程。

将 AsNoTracking 添加到“详细信息”和“删除”页模型

`AsNoTracking` 可以在不需要跟踪时提高性能。将 `AsNoTracking` 添加到“删除”和“详细信息”页模型。下面的代码显示更新的“删除”页模型：

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

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

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

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

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("./Index");
    }
}
```

在 Pages/Courses/Details.cshtml.cs 文件中更新 `OnGetAsync` 方法：

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

    Course = await _context.Courses
        .AsNoTracking()
        .Include(c => c.Department)
        .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

修改“删除”和“详细信息”页

使用以下标记更新“删除”Razor 页面：

```

@page
@model ContosoUniversity.Pages.Courses.DeleteModel

 @{
     ViewData["Title"] = "Delete";
 }

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>

```

对“详细信息”页执行相同更改。

测试“课程”页

测试“创建”、“编辑”、“详细信息”和“删除”。

更新“讲师”页

以下各部分介绍更新“讲师”页。

添加办公室位置

编辑讲师记录时，可能希望更新讲师的办公室分配。`Instructor` 实体与 `OfficeAssignment` 实体是一对零或一关系。讲师代码必须处理：

- 如果用户清除办公室分配，则需删除 `OfficeAssignment` 实体。
- 如果用户输入办公室分配，且办公室分配原本为空，则需创建一个新 `OfficeAssignment` 实体。
- 如果用户更改办公室分配，则需更新 `OfficeAssignment` 实体。

使用以下代码更新讲师“编辑”页模型：

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

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

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");
    }
}
```

前面的代码：

- 使用 `OfficeAssignment` 导航属性的预先加载从数据库获取当前的 `Instructor` 实体。
- 用模型绑定器中的值更新检索到的 `Instructor` 实体。`TryUpdateModel` 可防止过多发布。

- 如果办公室位置为空，则将 `Instructor.OfficeAssignment` 设置为 null。当 `Instructor.OfficeAssignment` 为 null 时，`OfficeAssignment` 表中的相关行将会删除。

更新讲师“编辑”页

使用办公室位置更新 Pages/Instructors/Edit.cshtml：

```

@page
@model ContosoUniversity.Pages.Instructors.EditModel
 @{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page=".~/Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

验证是否可以更改讲师办公室位置。

向“讲师编辑”页添加课程分配

讲师可能教授任意数量的课程。本部分将添加更改课程分配的功能。下图显示更新的讲师“编辑”页：

Last Name
Abercrombie

First Name
Kim

Hire Date
3/11/1995

Office Location
44/3P

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Save

Course 和 Instructor 具有多对多关系。若要添加和删除关系，可以从 CourseAssignments 联接实体集中添加和删除实体。

通过复选框可对分配给讲师的课程进行更改。数据库中的每一门课程均有对应显示的复选框。已分配给讲师的课程将会被勾选。用户可以通过选择或清除复选框来更改课程分配。如果课程数量过多：

- 可以使用其他用户界面显示课程。
- 使用联接实体创建或删除关系的方法不会发生更改。

添加类以支持“创建”和“编辑”讲师页

使用以下代码创建 SchoolViewModels/AssignedCourseData.cs：

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

AssignedCourseData 类包含的数据可用于为讲师已分配的课程创建复选框。

创建 Pages/Instructors/InstructorCoursesPageModel.cshtml.cs 基类：

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                              Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
                                            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>(
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID)));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
                        context.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

```
        }  
    }  
}
```

`InstructorCoursesPageModel` 是将用于“编辑”和“创建”页模型的基类。`PopulateAssignedCourseData` 读取所有 `Course` 实体以填充 `AssignedCourseDataList`。该代码将设置每门课程的 `CourseID` 和标题，并决定是否为讲师分配该课程。`HashSet` 用于创建高效查找。

讲师“编辑”页模型

使用以下代码更新讲师“编辑”页模型：

```

public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

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

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}

```

上面的代码处理办公室分配更改。

更新“讲师”Razor 视图：

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
 @{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />


<div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;
                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                                name="selectedCourses"
                                                value="@course.CourseID"
                                                @(Html.Raw(course.Assigned ? "checked=\"checked\" : "")) />
                                            @course.CourseID @: @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                        </table>
                </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>


```

```
</div>

<div>
    <a asp-page=".~/Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

注意

将代码粘贴到 Visual Studio 中时，换行符会发生更改并导致代码中断。按 Ctrl+Z 一次可撤消自动格式设置。按 Ctrl+Z 可以修复换行符，使其看起来如此处所示。缩进不一定要完美，但 `@</tr><tr>`、`@:<td>`、`@:</td>` 和 `@:</tr>` 行必须各成一行（如下所示）。选中新的代码块后，按 Tab 三次，使新代码与现有代码对齐。[通过此链接投票或查看此 bug 的状态。](#)

上面的代码将创建一个具有三列的 HTML 表。每列均具有一个复选框和包含课程编号及标题的标题。所有复选框均具有相同的名称（“SelectedCourses”）。使用相同名称可指示模型绑定器将它们视为一个组。每个复选框的值特性都设置为 `CourseID`。发布页面时，模型绑定器会传递一个数组，该数组只包括所选复选框的 `CourseID` 值。

初次呈现复选框时，分配给讲师的课程具有已勾选的特性。

运行应用并测试更新的讲师“编辑”页。更改某些课程分配。这些更改将反映在“索引”页上。

注意：此处所使用的编辑讲师课程数据的方法适用于数量有限的课程。对于规模远大于此的集合，则使用不同的 UI 和不同的更新方法会更实用和更高效。

更新讲师“创建”页

使用以下代码更新讲师“创建”页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task OnPostAsync(string[] selectedCourses)
```

```

    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var newInstructor = new Instructor();
        if (selectedCourses != null)
        {
            newInstructor.CourseAssignments = new List<CourseAssignment>();
            foreach (var course in selectedCourses)
            {
                var courseToAdd = new CourseAssignment
                {
                    CourseID = int.Parse(course)
                };
                newInstructor.CourseAssignments.Add(courseToAdd);
            }
        }

        if (await TryUpdateModelAsync<Instructor>(
            newInstructor,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            _context.Instructors.Add(newInstructor);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        PopulateAssignedCourseData(_context, newInstructor);
        return Page();
    }
}
}

```

前面的代码与 Pages/Instructors/Edit.cshtml.cs 代码类似。

使用以下标记更新讲师“创建”Razor 页面：

```

@page
@model ContosoUniversity.Pages.Instructors.CreateModel

 @{
     ViewData["Title"] = "Create";
 }

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
    </div>
</div>

```

```

<input asp-for="Instructor.HireDate" class="form-control" />
<span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
</div>

<div class="form-group">
    <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
</div>
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

                    foreach (var course in Model.AssignedCourseDataList)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                name="selectedCourses"
                                value="@course.CourseID"
                                @(Html.Raw(course.Assigned ? "checked=\"checked\"" : ""))
                                @course.CourseID @: @course.Title
                            @:</td>
                        }
                        @:</tr>
                    }
                </table>
            </div>
        </div>
        <div class="form-group">
            <input type="submit" value="Create" class="btn btn-default" />
        </div>
    </form>
</div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

测试讲师“创建”页。

更新“删除”页

使用以下代码更新“删除”页模型：

```

using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

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

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}

```

上面的代码执行以下更改：

- 对 `CourseAssignments` 导航属性使用预先加载。必须包含 `CourseAssignments`，否则删除讲师时将不会删除课程。为避免阅读它们，可以在数据库中配置级联删除。
- 如果要删除的讲师被指派为任何系的管理员，则需从这些系中删除该讲师分配。

[上一页](#)

[下一页](#)

zh-cn/

ASP.NET Core 中的 Razor 页面和 EF Core - 并发 - 第 8 个教程(共 8 个)

作者: [Rick Anderson](#)、[Tom Dykstra](#) 和 [Jon P Smith](#)

Contoso University Web 应用演示了如何使用 EF Core 和 Visual Studio 创建 Razor 页面 Web 应用。若要了解系列教程, 请参阅[第一个教程](#)。

本教程介绍如何处理多个用户并发更新同一实体(同时)时出现的冲突。如果遇到无法解决的问题, 请下载[本阶段的已完成应用](#)。

并发冲突

在以下情况下, 会发生并发冲突:

- 用户导航到实体的编辑页面。
- 第一个用户的更改还未写入数据库之前, 另一用户更新同一实体。

如果未启用并发检测, 当发生并发更新时:

- 最后一个更新优先。也就是最后一个更新的值保存至数据库。
- 第一个并发更新将会丢失。

开放式并发

乐观并发允许发生并发冲突, 并在并发冲突发生时作出正确反应。例如, Jane 访问院系编辑页面, 将英语系的预算从 350,000.00 美元更改为 0.00 美元。

Edit

Department

Budget

0

Administrator

Abercrombie, Kim

Name

English

Start Date

9/1/2007

Save

在 Jane 单击“保存”之前, John 访问了相同页面, 并将开始日期字段从 2007/1/9 更改为 2013/1/9。

Edit

Department

Budget

350000.00

Administrator

Abercrombie, Kim

Name

English

Start Date

9/1/2013

Save

Jane 先单击“保存”，并在浏览器显示索引页时看到她的更改。

Name	Budget	Administrator	Start Date
English	\$0.00	Abercrombie, Kim	2007-09-01

John 单击“编辑”页面上的“保存”，但页面的预算仍显示为 350,000.00 美元。接下来的情况取决于并发冲突的处理方式。

乐观并发包括以下选项：

- 可以跟踪用户已修改的属性，并仅更新数据库中相应的列。

在这种情况下，数据不会丢失。两个用户更新了不同的属性。下次有人浏览英语系时，将看到 Jane 和 John 两个人的更改。这种更新方法可以减少导致数据丢失的冲突数。本方法：*如果对同一属性进行竞争性更改，则无法避免数据丢失。^{*}通常不适用于 Web 应用。它需要维持重要状态，以便跟踪所有提取值和新值。维持大量状态可能影响应用性能。^{*}相对于实体上的并发检测，可能增加应用复杂性。

- 可让 John 的更改覆盖 Jane 的更改。

下次有人浏览英语系时，将看到 2013/9/1 和提取的值 350,000.00 美元。这种方法称为“客户端优先”或“最后一个优先”方案。（客户端的所有值优先于数据存储的值。）如果不对并发处理进行任何编码，则自动执行“客户端优先”。

- 可以阻止在数据库中更新 John 的更改。通常，应用将：^{*}显示错误消息。^{*}显示数据的当前状态。^{*}允许用户重新应用更改。

这称为“存储优先”方案。（数据存储值优先于客户端提交的值。）本教程实施“存储优先”方案。此方法可确保用户在未收到警报时不会覆盖任何更改。

处理并发

当属性配置为并发令牌时：

- EF Core 验证提取属性后是否未更改属性。调用 `SaveChanges` 或 `SaveChangesAsync` 时会执行此检查。
- 如果提取属性后更改了属性，将引发 `DbUpdateConcurrencyException`。

数据库和数据模型必须配置为支持引发 `DbUpdateConcurrencyException`。

检测属性的并发冲突

可使用 `ConcurrencyCheck` 特性在属性级别检测并发冲突。该特性可应用于模型上的多个属性。有关详细信息，请参阅 [数据注释 - ConcurrencyCheck](#)。

本教程中不使用 `[ConcurrencyCheck]` 特性。

检测行的并发冲突

要检测并发冲突，请将 `rowversion` 跟踪列添加到模型。`rowversion`：

- 是 SQL Server 特定的。其他数据库可能无法提供类似功能。
- 用于确定从数据库提取实体后未更改实体。

数据库生成 `rowversion` 序号，该数字随着每次行的更新递增。在 `Update` 或 `Delete` 命令中，`Where` 子句包括 `rowversion` 的提取值。如果要更新的行已更改：

- `rowversion` 不匹配提取值。
- `Update` 或 `Delete` 命令不能找到行，因为 `Where` 子句包含提取的 `rowversion`。
- 引发一个 `DbUpdateConcurrencyException`。

在 EF Core 中，如果未通过 `Update` 或 `Delete` 命令更新行，则引发并发异常。

向 Department 实体添加跟踪属性

在 `Models/Department.cs` 中，添加名为 `RowVersion` 的跟踪属性：

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

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

`Timestamp` 特性指定此列包含在 `Update` 和 `Delete` 命令的 `Where` 子句中。该特性称为 `Timestamp`，因为之前版本的 SQL Server 在 SQL `rowversion` 类型将其替换之前使用 SQL `timestamp` 数据类型。

Fluent API 还可指定跟踪属性：

```
modelBuilder.Entity<Department>()
    .Property<byte[]>("RowVersion")
    .IsRowVersion();
```

以下代码显示更新 `Department` 名称时由 EF Core 生成的部分 T-SQL：

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

前面突出显示的代码显示包含 `RowVersion` 的 `WHERE` 子句。如果数据库 `RowVersion` 不等于 `RowVersion` 参数 (`@p2`)，则不更新行。

以下突出显示的代码显示验证更新哪一行的 T-SQL：

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

`@@ROWCOUNT` 返回受上一语句影响的行数。在没有行更新的情况下，EF Core 引发 `DbUpdateConcurrencyException`。

在 Visual Studio 的输出窗口中可看见 EF Core 生成的 T-SQL。

更新数据库

添加 `RowVersion` 属性可更改数据库模型，这需要迁移。

生成项目。在命令窗口中输入以下命令：

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

前面的命令：

- 添加 `Migrations/{time stamp}_RowVersion.cs` 迁移文件。
- 更新 `Migrations/SchoolContextModelSnapshot.cs` 文件。此次更新将以下突出显示的代码添加到 `BuildModel` 方法：

```

modelBuilder.Entity("ContosoUniversity.Models.Department", b =>
{
    b.Property<int>("DepartmentID")
        .ValueGeneratedOnAdd();

    b.Property<decimal>("Budget")
        .HasColumnType("money");

    b.Property<int?>("InstructorID");

    b.Property<string>("Name")
        .HasMaxLength(50);

    b.Property<byte[]>("RowVersion")
        .IsConcurrencyToken()
        .ValueGeneratedOnAddOrUpdate();

    b.Property<DateTime>("StartDate");

    b.HasKey("DepartmentID");

    b.HasIndex("InstructorID");

    b.ToTable("Department");
});

```

- 运行迁移以更新数据库。

构架院系模型

- 退出 Visual Studio。
- 打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。
- 运行下面的命令：

```
dotnet aspnet-codegenerator razorpage -m Department -dc SchoolContext -udl -outDir Pages\Departments -referenceScriptLibraries
```

上述命令为 `Department` 模型创建基架。在 Visual Studio 中打开项目。

生成项目。此版本生成如下错误：

```
1>Pages/Departments/Index.cshtml.cs(26,37,26,43): error CS1061: 'SchoolContext' does not contain a definition for 'Department' and no extension method 'Department' accepting a first argument of type 'SchoolContext' could be found (are you missing a using directive or an assembly reference?)
```

将 `_context.Department` 全局更改为 `_context.Departments` (即向 `Department` 添加一个“s”)。找到并更新 7 个匹配项。

更新院系索引页

基架引擎为索引页创建 `RowVersion` 列，但不应显示该字段。本教程中显示 `RowVersion` 的最后一个字节，以帮助理解并发。不能保证最后一个字节是唯一的。实际应用不会显示 `RowVersion` 或 `RowVersion` 的最后一个字节。

更新索引页：

- 用院系替换索引。
- 将包含 `RowVersion` 的标记替换为 `RowVersion` 的最后一个字节。
- 将 `FirstMidName` 替换为 `FullName`。

以下标记显示更新后的页面：

```

@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}



## Departments



Create New



| @Html.DisplayNameFor(model => model.Department[0].Name) | @Html.DisplayNameFor(model => model.Department[0].Budget) | @Html.DisplayNameFor(model => model.Department[0].StartDate) | @Html.DisplayNameFor(model => model.Department[0].Administrator) | RowVersion          |
|---------------------------------------------------------|-----------------------------------------------------------|--------------------------------------------------------------|------------------------------------------------------------------|---------------------|
| @Html.DisplayFor(modelItem => item.Name)                | @Html.DisplayFor(modelItem => item.Budget)                | @Html.DisplayFor(modelItem => item.StartDate)                | @Html.DisplayFor(modelItem => item.Administrator.FullName)       | @item.RowVersion[7] |


```

更新编辑页模型

使用以下代码更新 pages\departments\edit.cshtml.cs：

```
using ContosoUniversity.Data;
```

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator) // eager loading
                .AsNoTracking() // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            // null means Department was deleted by another user.
            if (departmentToUpdate == null)
            {
                return await HandleDeletedDepartment();
            }

            // Update the RowVersion to the value when this entity was
            // fetched. If the entity has been updated after it was
            // fetched, RowVersion won't match the DB RowVersion and
            // a DbUpdateConcurrencyException is thrown.
            // A second postback will make them match, unless a new
            // concurrency issue happens.
            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;
        }
    }
}
```

```

        if (await TryUpdateModelAsync<Department>(
            departmentToUpdate,
            "Department",
            s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty, "Unable to save. " +
                    "The department was deleted by another user.");
                return Page();
            }

            var dbValues = (Department)databaseEntry.ToObject();
            await setDbErrorMessage(dbValues, clientValues, _context);

            // Save the current RowVersion so next postback
            // matches unless an new concurrency issue happens.
            Department.RowVersion = (byte[])dbValues.RowVersion;
            // Must clear the model error for the next postback.
            ModelState.Remove("Department.RowVersion");
        }
    }

    InstructorNameSL = new SelectList(_context.Instructors,
        "ID", "FullName", departmentToUpdate.InstructorID);

    return Page();
}

private async Task<IActionResult> HandleDeletedDepartment()
{
    Department deletedDepartment = new Department();
    // ModelState contains the posted data because of the deletion error and will override the
    Department instance values when displaying Page().
    ModelState.AddModelError(string.Empty,
        "Unable to save. The department was deleted by another user.");
    InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName",
    Department.InstructorID);
    return Page();
}

private async Task setDbErrorMessage(Department dbValues,
    Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
}

```

```

        }

        if (dbValues.InstructorID != clientValues.InstructorID)
        {
            Instructor dbInstructor = await _context.Instructors
                .FindAsync(dbValues.InstructorID);
            ModelState.AddModelError("Department.InstructorID",
                $"Current value: {dbInstructor?.FullName}");
        }

        ModelState.AddModelError(string.Empty,
            "The record you attempted to edit "
            + "was modified by another user after you. The "
            + "edit operation was canceled and the current values in the database "
            + "have been displayed. If you still want to edit this record, click "
            + "the Save button again.");
    }
}
}
}

```

要检测并发问题，请使用来自所提取实体的 `rowVersion` 值更新 `OriginalValue`。EF Core 使用包含原始 `RowVersion` 值的 WHERE 子句生成 SQL UPDATE 命令。如果没有行受到 UPDATE 命令影响（没有行具有原始 `RowVersion` 值），将引发 `DbUpdateConcurrencyException` 异常。

```

public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    // null means Department was deleted by another user.
    if (departmentToUpdate == null)
    {
        return await HandleDeletedDepartment();
    }

    // Update the RowVersion to the value when this entity was
    // fetched. If the entity has been updated after it was
    // fetched, RowVersion won't match the DB RowVersion and
    // a DbUpdateConcurrencyException is thrown.
    // A second postback will make them match, unless a new
    // concurrency issue happens.
    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
}

```

在前面的代码中，`Department.RowVersion` 为实体提取后的值。使用此方法调用 `FirstOrDefaultAsync` 时，`OriginalValue` 为数据库中的值。

以下代码获取客户端值（向此方法发布的值）和数据库值：

```

try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless a new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}

```

以下代码为每列添加自定义错误消息，这些列中的数据库值与发布到 `OnPostAsync` 的值不同：

```

private async Task setDbErrorMessage(Department dbValues,
                                    Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
            .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit " +
        "was modified by another user after you. The " +
        "edit operation was canceled and the current values in the database " +
        "have been displayed. If you still want to edit this record, click " +
        "the Save button again.");
}

```

以下突出显示的代码将 `RowVersion` 值设置为从数据库检索的新值。用户下次单击“保存”时，将仅捕获最后一次显示编辑页后发生的并发错误。

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless a new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

`ModelState` 具有旧的 `RowVersion` 值，因此需使用 `ModelState.Remove` 语句。在 Razor 页面中，当两者都存在时，字段的 `ModelState` 值优于模型属性值。

更新“编辑”页

使用以下标记更新 Pages/Departments/Edit.cshtml：

```

@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
 @{
     ViewData["Title"] = "Edit";
 }
 <h2>Edit</h2>
 <h4>Department</h4>
 <hr />
 <div class="row">
     <div class="col-md-4">
         <form method="post">
             <div asp-validation-summary="ModelOnly" class="text-danger"></div>
             <input type="hidden" asp-for="Department.DepartmentID" />
             <input type="hidden" asp-for="Department.RowVersion" />
             <div class="form-group">
                 <label>RowVersion</label>
                 @Model.Department.RowVersion[7]
             </div>
             <div class="form-group">
                 <label asp-for="Department.Name" class="control-label"></label>
                 <input asp-for="Department.Name" class="form-control" />
                 <span asp-validation-for="Department.Name" class="text-danger"></span>
             </div>
             <div class="form-group">
                 <label asp-for="Department.Budget" class="control-label"></label>
                 <input asp-for="Department.Budget" class="form-control" />
                 <span asp-validation-for="Department.Budget" class="text-danger"></span>
             </div>
             <div class="form-group">
                 <label asp-for="Department.StartDate" class="control-label"></label>
                 <input asp-for="Department.StartDate" class="form-control" />
                 <span asp-validation-for="Department.StartDate" class="text-danger">
                     </span>
             </div>
             <div class="form-group">
                 <label class="control-label">Instructor</label>
                 <select asp-for="Department.InstructorID" class="form-control" asp-items="@Model.InstructorNameSL"></select>
                 <span asp-validation-for="Department.InstructorID" class="text-danger">
                     </span>
             </div>
             <div class="form-group">
                 <input type="submit" value="Save" class="btn btn-default" />
             </div>
         </form>
     </div>
 </div>
 <div>
     <a asp-page=".~/Index">Back to List</a>
 </div>
 @section Scripts {
     @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
 }

```

前面的标记：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int}"`。
- 添加隐藏的行版本。必须添加 `RowVersion`，以便回发绑定值。
- 显示 `RowVersion` 的最后一个字节以进行调试。
- 将 `ViewData` 替换为强类型 `InstructorNameSL`。

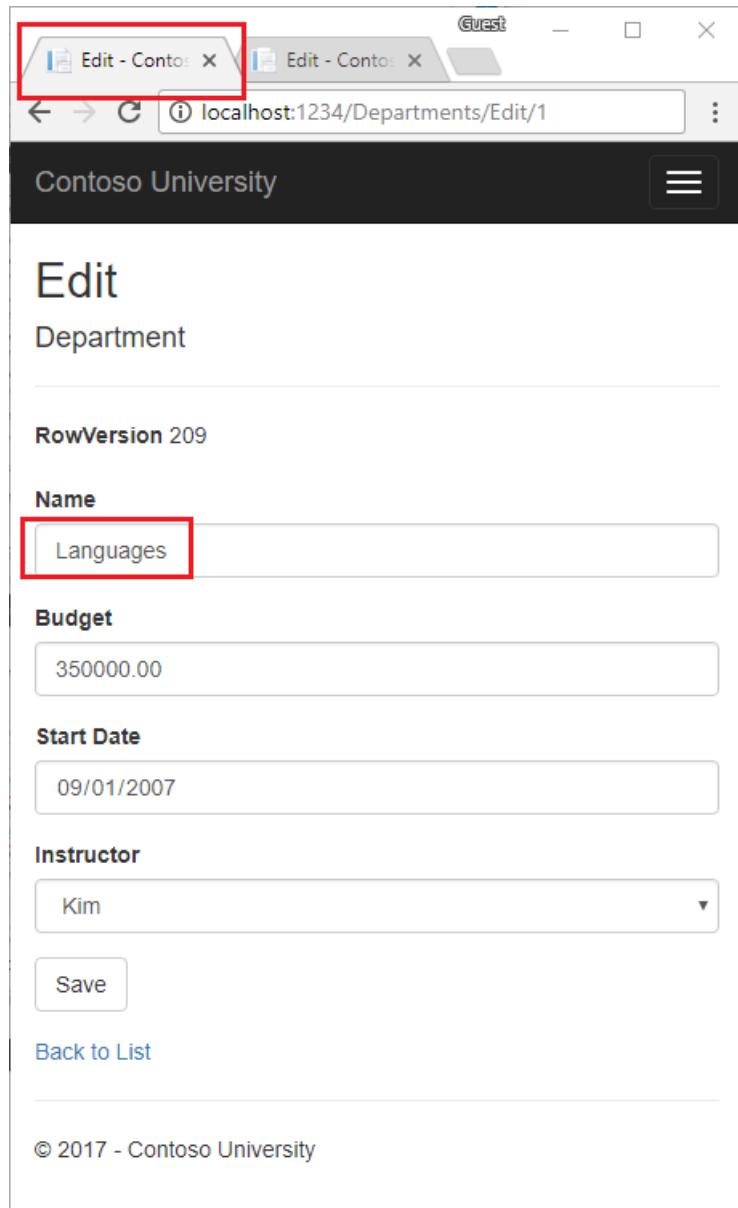
使用编辑页测试并发冲突

在英语系打开编辑的两个浏览器实例：

- 运行应用，然后选择“院系”。
- 右键单击英语系的“编辑”超链接，然后选择“在新选项卡中打开”。
- 在第一个选项卡中，单击英语系的“编辑”超链接。

两个浏览器选项卡显示相同信息。

在第一个浏览器选项卡中更改名称，然后单击“保存”。



The screenshot shows a web browser window with two tabs. Both tabs display the same 'Edit' form for a department. The active tab (foreground) has its title bar highlighted with a red box. The title bar says 'Edit - Contoso' and the address bar shows 'localhost:1234/Departments/Edit/1'. The form itself has a header 'Edit Department'. It contains fields for 'RowVersion' (set to 209), 'Name' (set to 'Languages'), 'Budget' (set to 350000.00), 'Start Date' (set to 09/01/2007), and 'Instructor' (set to 'Kim'). At the bottom are 'Save' and 'Back to List' buttons. The second tab (background) is also titled 'Edit - Contoso' and shows the same form with the same values.

浏览器显示更改值并更新 rowVersion 标记后的索引页。请注意更新后的 rowVersion 标记，它在其他选项卡的第二回发中显示。

在第二个浏览器选项卡中更改不同字段。

Screenshot of a web browser showing the 'Edit - Contoso University' page for a department. The browser title bar shows 'Edit - Contoso University'. The address bar shows 'localhost:1234/Departments/Edit/1'. The page header says 'Contoso University'. The main content area has a heading 'Edit' and 'Department'. It includes fields for 'Name' (value: English), 'Budget' (value: 5000000, highlighted with a red box), 'Start Date' (value: 09/01/2007), and 'Instructor' (value: Kim). A 'Save' button is present at the bottom left. A 'Back to List' link is at the bottom center. The footer says '© 2017 - Contoso University'.

RowVersion 209

Name

English

Budget

5000000

Start Date

09/01/2007

Instructor

Kim

Save

[Back to List](#)

© 2017 - Contoso University

单击“保存”。可看见所有不匹配数据库值的字段的错误消息：

Guest

Departments Edit - Contoso University

localhost:1234/Departments/Edit/1

Edit Department

The record you attempted to edit was modified by another user after you. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again.

RowVersion 21

Name
English
Current value: Languages

Budget
5000000
Current value: \$350,000.00

Start Date
09/01/2007

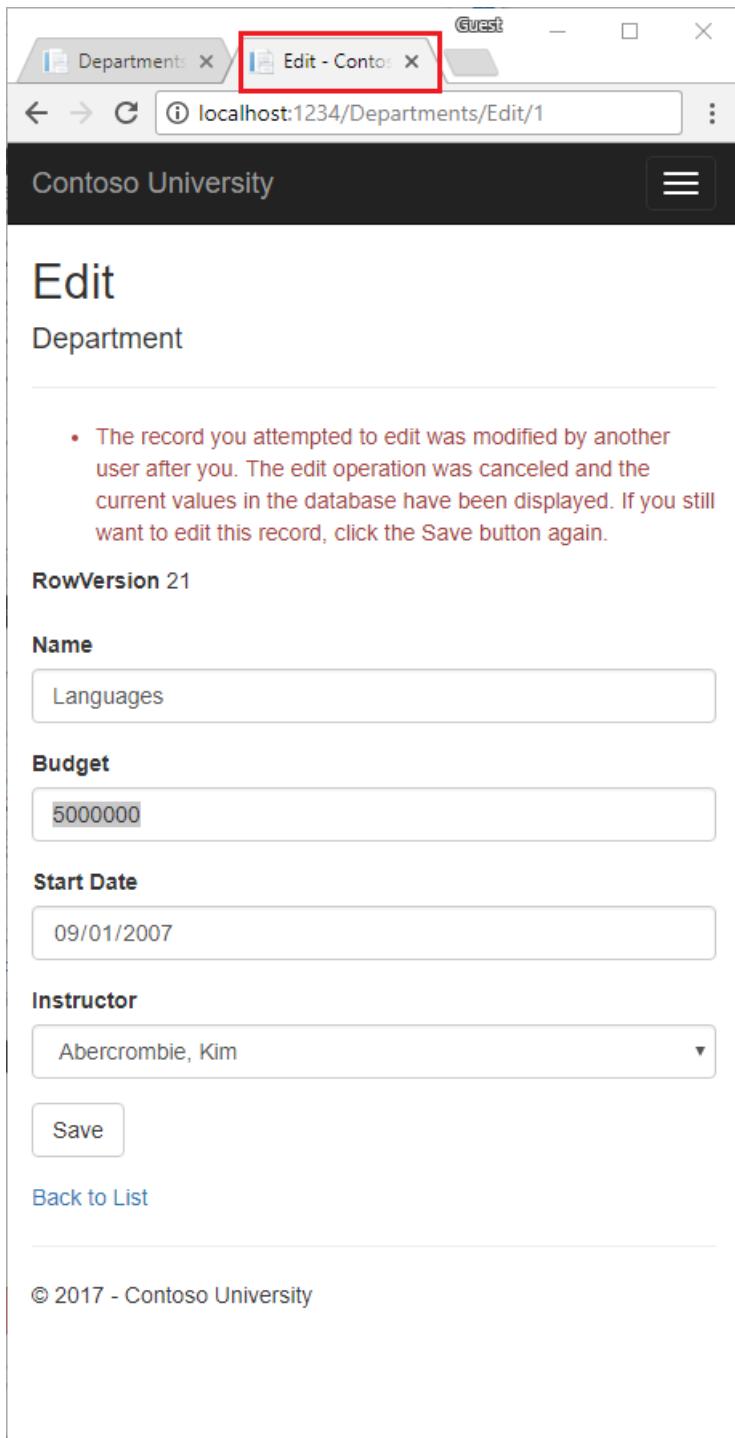
Instructor
Abercrombie, Kim

Save

[Back to List](#)

© 2017 - Contoso University

此浏览器窗口将不会更改名称字段。将当前值(语言)复制并粘贴到名称字段。退出选项卡。客户端验证将删除错误消息。



再次单击“保存”。保存在第二个浏览器选项卡中输入的值。在索引页中可以看到保存的值。

更新“删除”页

使用以下代码更新“删除”页模型：

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
```

```

public DeleteModel(ContosoUniversity.Data.SchoolContext context)
{
    _context = context;
}

[BindProperty]
public Department Department { get; set; }
public string ConcurrencyErrorMessage { get; set; }

public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
{
    Department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (Department == null)
    {
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ConcurrencyErrorMessage = "The record you attempted to delete "
            + "was modified by another user after you selected delete. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again.";
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    try
    {
        if (await _context.Departments.AnyAsync(
            m => m.DepartmentID == id))
        {
            // Department.rowVersion value is from when the entity
            // was fetched. If it doesn't match the DB, a
            // DbUpdateConcurrencyException exception is thrown.
            _context.Departments.Remove(Department);
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException)
    {
        return RedirectToPage("./Delete",
            new { concurrencyError = true, id = id });
    }
}
}
}

```

删除页检测提取实体并更改时的并发冲突。提取实体后，`Department.RowVersion` 为行版本。EF Core 创建 SQL DELETE 命令时，它包括具有 `RowVersion` 的 WHERE 子句。如果 SQL DELETE 命令导致零行受影响：

- SQL DELETE 命令中的 `RowVersion` 与数据库中的 `RowVersion` 不匹配。
- 引发 `DbUpdateConcurrencyException` 异常。
- 使用 `ConcurrencyErrorMessage` 调用 `OnGetAsync`。

更新“删除”页

使用以下代码更新 Pages/Departments/Delete.cshtml：

```

@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
    </form>
</div>

```

上述标记进行以下更改：

- 将 `page` 指令从 `@page` 更新为 `@page "{id:int}"`。
- 添加错误消息。
- 将“管理员”字段中的 `FirstMidName` 替换为 `FullName`。
- 更改 `RowVersion` 以显示最后一个字节。

- 添加隐藏的行版本。必须添加 `RowVersion`，以便回发绑定值。

使用删除页测试并发冲突

创建测试系。

在测试系打开删除的两个浏览器实例：

- 运行应用，然后选择“院系”。
- 右键单击测试系的“删除”超链接，然后选择“在新选项卡中打开”。
- 单击测试系的“编辑”超链接。

两个浏览器选项卡显示相同信息。

在第一个浏览器选项卡中更改预算，然后单击“保存”。

浏览器显示更改值并更新 `rowVersion` 标记后的索引页。请注意更新后的 `rowVersion` 标记，它在其他选项卡的第二回发中显示。

从第二个选项卡中删除测试部门。并发错误显示来自数据库的当前值。单击“删除”将删除实体，除非 `RowVersion` 已更新，院系已删除。

请参阅[继承](#)了解如何继承数据模型。

其他资源

- [EF Core 中的并发令牌](#)
- [EF Core 中的并发处理](#)

[上一篇](#)

ASP.NET Core MVC 和 EF Core - 教程系列

2018/5/17 • 1 min to read • [Edit Online](#)

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 [Razor 页面教程](#)。Razor 页面教程：

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

1. [入门](#)
2. [创建、读取、更新和删除操作](#)
3. [排序、筛选、分页和分组](#)
4. [迁移](#)
5. [创建复杂数据模型](#)
6. [读取相关数据](#)
7. [更新相关数据](#)
8. [处理并发冲突](#)
9. [继承](#)
10. [高级主题](#)

ASP.NET Core MVC 和 Entity Framework Core - 第 1 个教程, 共 10 个教程

2018/5/17 • 26 min to read • [Edit Online](#)

作者: [Tom Dykstra](#) 和 [Rick Anderson](#)

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择, 它是基于页面的编程模型, 可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 [Razor 页面](#) 教程。Razor 页面教程:

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework (EF) Core 2.0 和 Visual Studio 2017 创建 ASP.NET Core 2.0 MVC web 应用程序。

示例应用程序供一个虚构的 Contoso 大学网站使用。它包括诸如学生入学、课程创建和导师分配等功能。这是一系列教程中的第一个, 这一系列教程主要展示了如何从零开始构建 Contoso 大学示例应用程序。

[下载或查看已完成的应用程序。](#)

EF Core 2.0 是 EF 的最新版本, 但还没有包括 EF 6.x 的所有功能。有关如何在 EF 6.x 和 EF Core 之间选择, 请参阅 [EF Core vs. EF 6.x](#)。如果你选择使用 EF 6.x, 请参阅 [本系列教程的上一个版本](#)。

注意

- 本教程的 ASP.NET Core 1.1 版本, 请参阅 [本教程中 VS 2017 Update 2 版本的 PDF 文档](#)。
- 有关本教程的 Visual Studio 2015 版本, 请参阅 [ASP.NET Core 文档 VS 2015 版本的 PDF 文档](#)。

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

疑难解答

如果遇到无法解决的问题, 可以通过与 [已完成的项目](#) 对比代码来查找解决方案。常见错误以及对应的解决方案, 请参阅 [最新教程中的故障排除](#)。如果没有找到遇到的问题的解决方案, 可以将问题发布到

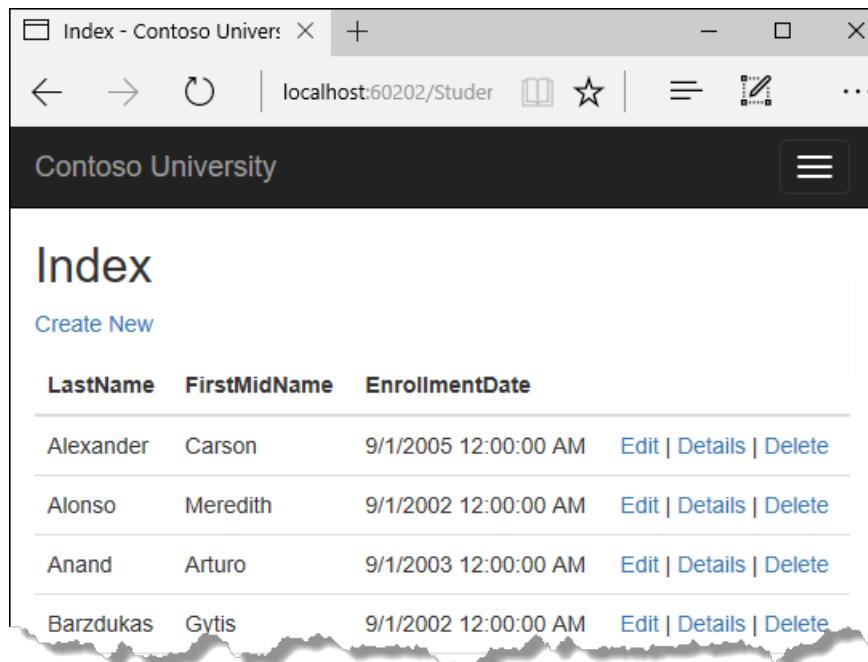
提示

这是一系列一共有十个教程，其中每个都是在前面教程已完成的基础上继续。请考虑在完成每一个教程后保存项目的副本。之后如果遇到问题，你可以从保存的副本中开始寻找问题，而不是从头开始。

Contoso University Web 应用程序

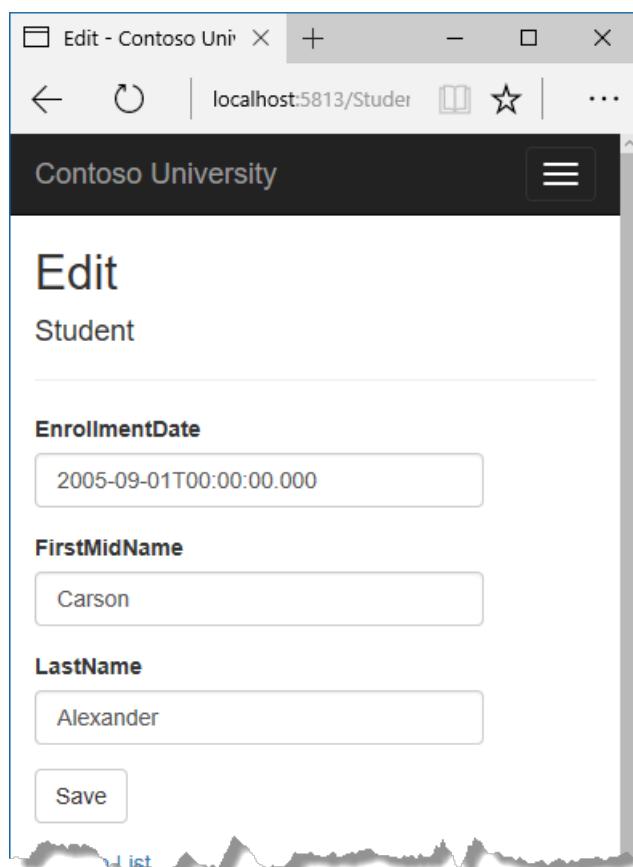
你将在这些教程中学习构建一个简单的大学网站的应用程序。

用户可以查看和更新学生、课程和教师信息。以下是一些你即将创建的页面。



The screenshot shows a web browser window titled "Index - Contoso University". The address bar shows "localhost:60202/Studen". The main content area displays a table with student data:

Last Name	First/Middle Name	Enrollment Date	Action
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete



The screenshot shows a web browser window titled "Edit - Contoso University". The address bar shows "localhost:5813/Studen". The main content area displays an "Edit" form for a student:

Student

Enrollment Date: 2005-09-01T00:00:00.000

First/Middle Name: Carson

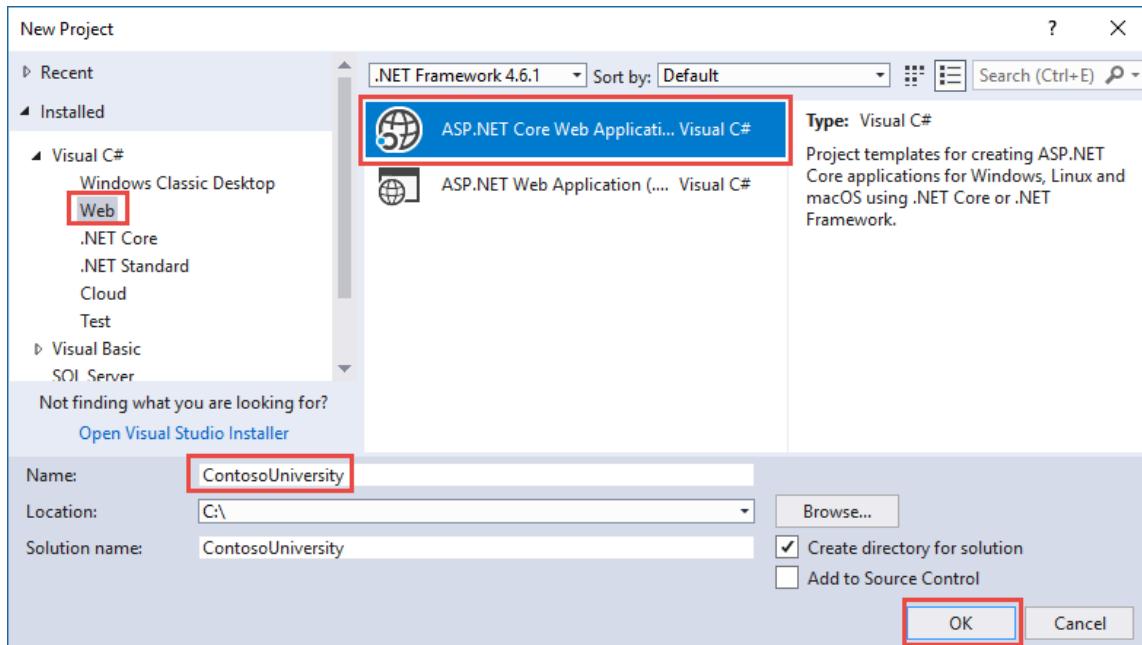
Last Name: Alexander

本教程主要关注于如何使用 Entity Framework，所以此站点的UI样式都是直接套用内置的模板。

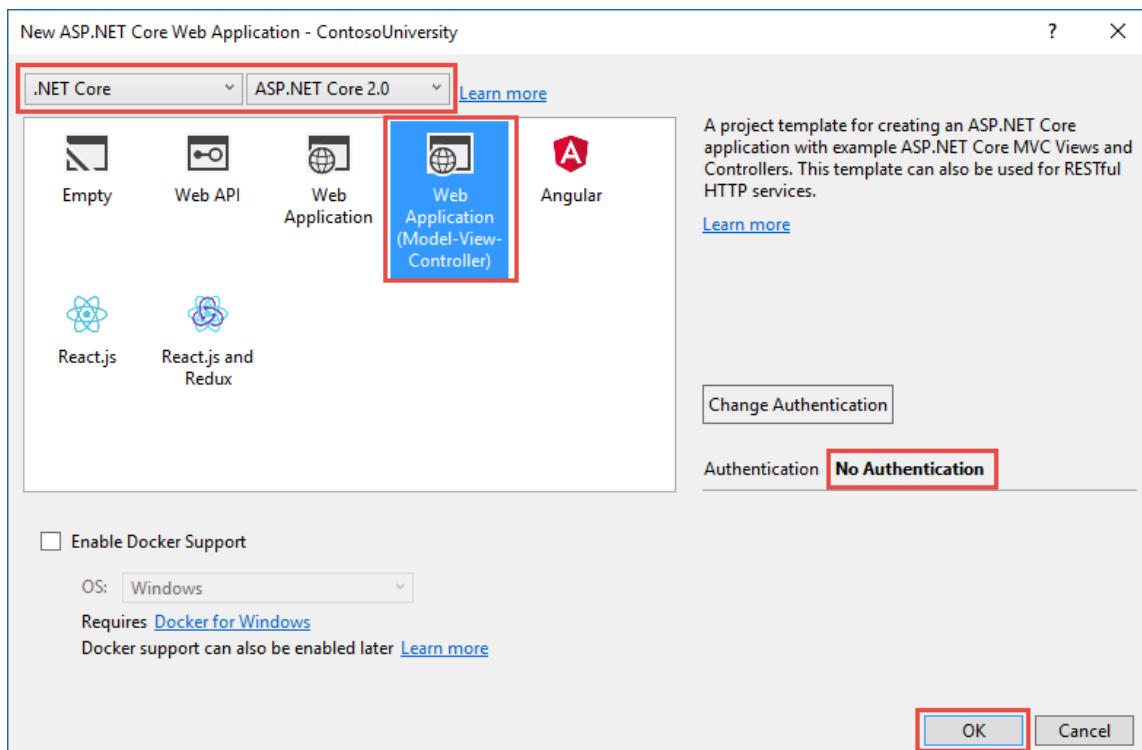
创建 ASP.NET Core MVC web 应用程序

打开 Visual Studio 并创建一个新 ASP.NET Core C# web 项目名为"ContosoUniversity"。

- 从“文件”菜单中选择“新建”>“项目”。
- 从左窗格中依次选择“已安装”>“Visual C#”>“Web”。
- 选择“ASP.NET Core Web 应用程序”项目模板。
- 输入“ContosoUniversity”作为名称，然后单击“确定”。



- 等待 新 ASP.NET Core Web 应用程序 (.NET Core) 显示对话框
 - 选择 **ASP.NET Core 2.0 和 Web 应用程序（模型-视图-控制器）** 模板。
- 注意：本教程需要安装 ASP.NET Core 2.0 和 EF Core 2.0 或更高版本-请确保 **ASP.NET Core 1.1** 未选中。
- 请确保 **身份验证** 设置为 **不进行身份验证**。
 - 单击“确定”



设置网站样式

通过几个简单的更改设置站点菜单、布局和主页。

打开 Views/Shared/_Layout.cshtml 并进行以下更改：

- 将文件中的"ContosoUniversity"更改为"Contoso University"。需要更改三个地方。
- 添加菜单项 **Students, Courses, Instructors, 和 Department**, 并删除 **Contact**菜单项。

突出显示所作更改。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Contoso University</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet"
            href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
            asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
            asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>

</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                </button>
            </div>
            <div class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li>Home</li>
                    <li>About</li>
                    <li>Contact</li>
                    <li>Students</li>
                    <li>Courses</li>
                    <li>Instructors</li>
                    <li>Department</li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container">
        <div class="row">
            <div class="col-md-3">
                <h2>Contoso University</h2>
                <p>Learn how to build web apps with ASP.NET Core</p>
                <ul>
                    <li>Home</li>
                    <li>About</li>
                    <li>Contact</li>
                    <li>Students</li>
                    <li>Courses</li>
                    <li>Instructors</li>
                    <li>Department</li>
                </ul>
            </div>
            <div class="col-md-9">
                <h1>Contoso University</h1>
                <h2>Contoso University</h2>
                <h3>Contoso University</h3>
                <h4>Contoso University</h4>
                <h5>Contoso University</h5>
                <h6>Contoso University</h6>
            </div>
        </div>
    </div>
</body>

```

```

        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">Contoso
University</a>
    </div>
    <div class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
            <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
            <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
            <li><a asp-area="" asp-controller="Students" asp-action="Index">Students</a></li>
            <li><a asp-area="" asp-controller="Courses" asp-action="Index">Courses</a></li>
            <li><a asp-area="" asp-controller="Instructors" asp-action="Index">Instructors</a>
        </li>
        <li><a asp-area="" asp-controller="Departments" asp-action="Index">Departments</a>
        </li>
    </ul>
    </div>
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2017 - Contoso University</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfhIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-Tc5IQib027qvyjSMFHjOMaLkfuWVxZxUPnCJA712mCWNIPG9mGCD8wGNICPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

    @RenderSection("Scripts", required: false)
</body>
</html>

```

在 *Views/Home/Index.cshtml*, 将文件的内容替换为以下代码以将有关 ASP.NET 和 MVC 的内容替换为有关此应用程序的内容:

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default" href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

按 CTRL + F5 来运行该项目或从菜单选择 调试 > **开始执行(不调试)**。你会看到首页，以及通过这个教程创建的页对应的选项卡。

The screenshot shows a web browser window with the title bar "Home Page - Contoso University". The address bar displays "localhost:5813". The page content is the "Contoso University" home page, featuring a large header with the university's name, a "Welcome to Contoso University" section, a "Build it from scratch" section with a tutorial link, and a "Download it" section with a GitHub link. At the bottom, there is a copyright notice: "© 2016 - Contoso University".

Contoso University

Welcome to Contoso University

Contoso University is a sample application that demonstrates how to use Entity Framework Core 1.0 in an ASP.NET Core MVC 1.0 web application.

Build it from scratch

You can build the application by following the steps in a series of tutorials.

[See the tutorial »](#)

Download it

You can download the completed project from GitHub.

[See project source code »](#)

© 2016 - Contoso University

Entity Framework Core NuGet 包

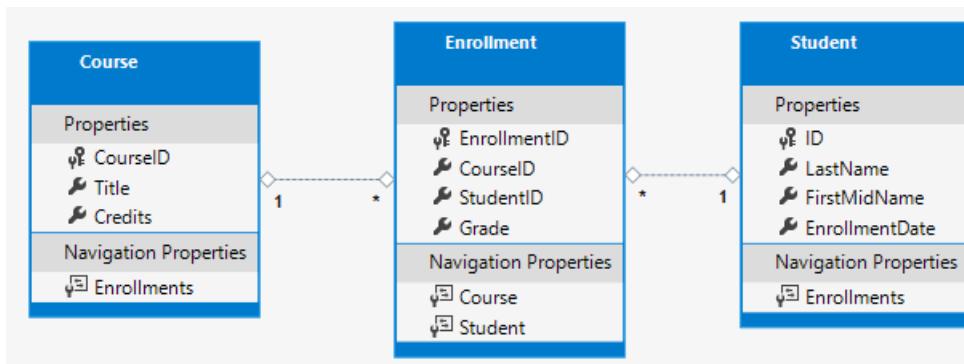
若要为项目添加 EF Core 支持，需要安装相应的数据库驱动包。本教程使用 SQL Server，相关驱动包 [Microsoft.EntityFrameworkCore.SqlServer](#)。该包包含在 [Microsoft.AspNetCore.All](#) 包中，因此不需要手动安装。

此包和其依赖项 (`Microsoft.EntityFrameworkCore` 和 `Microsoft.EntityFrameworkCore.Relational`) 一起提供 EF 的运行时支持。你将在之后的 [迁移](#) 教程中学习添加工具包。

有关其他可用于 EF Core 的数据库驱动的信息，请参阅 [数据库驱动](#)。

创建数据模型

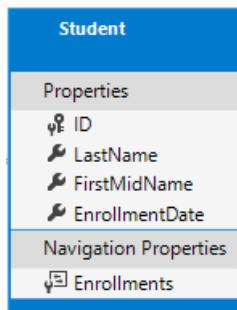
接下来你将创建 Contoso 大学应用程序的实体类。你将从以下三个实体类开始。



`Student` 和 `Enrollment` 实体之间是一对多的关系, `Course` 和 `Enrollment` 实体之间也是一个对多的关系。换而言之, 一名学生可以修读任意数量的课程, 并且某一课程可以被任意数量的学生修读。

接下来, 你将创建与这些实体对应的类。

Student 实体



在 `Models` 文件夹中, 创建一个名为 `Student.cs` 的类文件并且将模板代码替换为以下代码。

```

using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

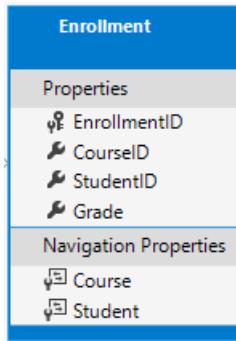
```

`ID` 属性将成为对应于此类的数据表中的主键。默认情况下, EF 将会将名为 `ID` 或 `classnameID` 的属性解析为主键。

`Enrollments` 属性是导航属性。导航属性中包含与此实体相关的其他实体。在这个案例下, `Student entity` 中的 `Enrollments` 属性会保留所有与 `Student` 实体相关的 `Enrollment`。换而言之, 如果在数据库中有两行描述同一个学生的修读情况 (两行的 `StudentID` 值相同, 而且 `StudentID` 作为外键和某位学生的主键值相同), `Student` 实体的 `Enrollments` 导航属性将包含那两个 `Enrollment` 实体。

如果导航属性可以具有多个实体 (如多对多或一对多关系), 那么导航属性的类型必须是可以添加、删除和更新条目的容器, 如 `ICollection<T>`。你可以指定 `ICollection<T>` 或实现该接口类型, 如 `List<T>` 或 `HashSet<T>`。如果指定 `ICollection<T>`, EF 在默认情况下创建 `HashSet<T>` 集合。

Enrollment 实体



在 *Models* 文件夹中，创建 *Enrollment.cs* 并且用以下代码替换现有代码：

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

`EnrollmentID` 属性将被设为主键；此实体使用 `classnameID` 模式而不是如 `Student` 实体那样直接使用 `ID`。通常情况下，你选择一个主键模式，并在你的数据模型自始至终使用这种模式。在这里，使用了两种不同的模式只是为了说明你可以使用任一模式来指定主键。在 [后面的教程](#)，你将了解到使用 `ID` 这种模式可以更轻松地在数据模型之间实现继承。

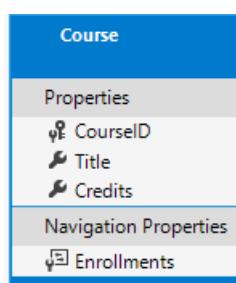
`Grade` 属性是 `enum`。`Grade` 声明类型后的 `?` 表示 `Grade` 属性可以为 `null`。评级为 `null` 和评级为零是有区别的 --`null` 意味着评级未知或者尚未分配。

`StudentID` 属性是一个外键，`Student` 是与其且对应的导航属性。`Enrollment` 实体与一个 `student` 实体相关联，因此该属性只包含单个 `Student` 实体（与前面所看到的 `Student.Enrollments` 导航属性不同后，`Student` 中可以容纳多个 `Enrollment` 实体）。

`CourseID` 属性是一个外键，`Course` 是与其对应的导航属性。`Enrollment` 实体与一个 `Course` 实体相关联。

如果一个属性名为 `<navigation property name><primary key property name>`，Entity Framework 就会将这个属性解析为外键属性（例如，`Student` 实体的主键是 `ID`，`Student` 是 `Enrollment` 的导航属性所以 `Enrollment` 实体中 `StudentID` 会被解析为外键）。此外还可以将需要解析为外键的属性命名为 `<primary key property name>`（例如，`CourseID` 由于 `Course` 实体的主键所以 `CourseID` 也被解析为外键）。

Course 实体



在 *Models* 文件夹中，创建 *Course.cs* 并且用以下代码替换现有代码：

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

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`Enrollments` 属性是导航属性。一个 `Course` 体可以与任意数量的 `Enrollment` 实体相关。

我们在本系列 [后面的教程](#) 中会有更多有关 `DatabaseGenerated` 特性的例子。简单来说，此特性让你能自行指定主键，而不是让数据库自动指定主键。

创建数据库上下文

使得给定的数据模型与 Entity Framework 功能相协调的主类是数据库上下文类。可以通过继承 `Microsoft.EntityFrameworkCore.DbContext` 类的方式创建此类。在该类中你可以指定数据模型中包含哪些实体。你还可以定义某些 Entity Framework 行为。在此项目中将数据库上下文类命名为 `SchoolContext`。

在项目文件夹中，创建名为的文件夹 *Data*。

在 *Data* 文件夹创建名为 *SchoolContext.cs* 的类文件，并将模板代码替换为以下代码：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {

        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

此代码将为每个实体集创建 `DbSet` 属性。在 Entity Framework 中，实体集通常与数据表相对应，具体实体与表中的行相对应。

在这里可以省略 `DbSet<Enrollment>` 和 `DbSet<Course>` 语句，实现的功能没有任何改变。Entity Framework 会隐式包含这两个实体因为 `Student` 实体引用了 `Enrollment` 实体、`Enrollment` 实体引用了 `Course` 实体。

当数据库创建完成后，EF 创建一系列数据表，表名默认和 `DbSet` 属性名相同。集合属性的名称一般使用复数形式，但不同的开发人员的命名习惯可能不一样，开发人员根据自己的情况确定是否使用复数形式。在定义 `DbSet` 属性的代码之后，添加下面高亮代码，对 `DbContext` 指定单数的表名来覆盖默认的表名。此教程在最后一个 `DbSet` 属性之后，添加以下高亮显示的代码。

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {

        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

使用依赖注入注册上下文

ASP.NET Core 默认实现 [依赖注入](#)。在应用程序启动过程通过依赖注入注册相关服务（例如 EF 数据库上下文）。需要这些服务的组件（如 MVC 控制器）可以通过向构造函数添加相关参数来获得对应服务。在本教程后面你将看到控制器构造函数的代码，就是通过上述方式获得上下文实例。

若要将 `SchoolContext` 注册为一种服务，打开 `Startup.cs`，并将高亮代码添加到 `ConfigureServices` 方法中。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

通过调用 `DbContextOptionsBuilder` 中的一个方法将数据库连接字符串在配置文件中的名称传递给上下文对象。进行本地开发时，[ASP.NET Core 配置系统](#) 在 `appsettings.json` 文件中读取数据库连接字符串。

添加 `using` 语句引用 `ContosoUniversity.Data` 和 `Microsoft.EntityFrameworkCore` 命名空间，然后生成项目。

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

打开 `appsettings.json` 文件，并如以下示例所示添加连接字符串。

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"  
    },  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    }  
}
```

SQL Server Express LocalDB

数据库连接字符串指定使用 SQL Server LocalDB 数据库。LocalDB 是 SQL Server Express 数据库引擎的轻量级版本，用于应用程序开发，不在生产环境中使用。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 在 `C:/Users/<user>` 目录下创建 `.mdf` 数据库文件。

添加代码以使用测试数据初始化数据库

Entity Framework 已经为你创建了一个空数据库。在本部分中，你将编写一个方法用于向数据库填充测试数据，该方法会在数据库创建完成之后执行。

此处将使用 `EnsureCreated` 方法来自动创建数据库。在 [后面的教程](#) 你将了解如何通过使用 Code First Migration 来更改而不是删除并重新创建数据库来处理模型更改。

在 `Data` 文件夹中，创建名为的新类文件 `DbInitializer.cs` 并且将模板代码替换为以下代码，使得在需要时能创建数据库并向其填充测试数据。

```
using ContosoUniversity.Models;  
using System;  
using System.Linq;  
  
namespace ContosoUniversity.Data  
{  
    public static class DbInitializer  
    {  
        public static void Initialize(SchoolContext context)  
        {  
            context.Database.EnsureCreated();  
  
            // Look for any students.  
            if (context.Students.Any())  
            {  
                return; // DB has been seeded  
            }  
  
            var students = new Student[]  
            {  
                new Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Parse("2005-09-01")},  
                new Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Parse("2003-09-01")},  
                new Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Yan", LastName="Li", EnrollmentDate=DateTime.Parse("2002-09-01")},  
                new Student{FirstMidName="Peggy", LastName="Justice", EnrollmentDate=DateTime.Parse("2001-09-01")},  
                new Student{FirstMidName="Laura", LastName="Norman", EnrollmentDate=DateTime.Parse("2003-09-01")},  
                new Student{FirstMidName="Nina", LastName="Olivetto", EnrollmentDate=DateTime.Parse("2005-09-01")},  
            };  
        }  
    }  
}
```

```

        new Student{First Name="John", Last Name="Doe", Enrollment Date=DateTime.Parse("2005-09-01")};

    };

    foreach (Student s in students)
    {
        context.Students.Add(s);
    }

    context.SaveChanges();

    var courses = new Course[]
    {
        new Course{CourseID=1050,Title="Chemistry",Credits=3},
        new Course{CourseID=4022,Title="Microeconomics",Credits=3},
        new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
        new Course{CourseID=1045,Title="Calculus",Credits=4},
        new Course{CourseID=3141,Title="Trigonometry",Credits=4},
        new Course{CourseID=2021,Title="Composition",Credits=3},
        new Course{CourseID=2042,Title="Literature",Credits=4}
    };
    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }

    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
        new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
        new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
        new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
        new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
        new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
        new Enrollment{StudentID=3,CourseID=1050},
        new Enrollment{StudentID=4,CourseID=1050},
        new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
        new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
        new Enrollment{StudentID=6,CourseID=1045},
        new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
    };
    foreach (Enrollment e in enrollments)
    {
        context.Enrollments.Add(e);
    }

    context.SaveChanges();
}

}

```

这段代码首先检查是否有学生数据在数据库中，如果没有的话，就可以假定数据库是新建的，然后使用测试数据进行填充。代码中使用数组存放测试数据而不是使用 `List<T>` 集合是为了优化性能。

在 `Program.cs`, 修改 `Main` 方法, 使得在应用程序启动时能执行以下操作:

- 从依赖注入容器中获取数据库上下文实例。
 - 调用 seed 方法，将上下文传递给它。
 - Seed 方法完成此操作时释放上下文。

```

public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}

```

添加 `using` 语句：

```

using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

```

在旧版教程中，你可能会在 `Startup.cs` 中的 `Configure` 方法看到类似的代码。我们建议你只在为了设置请求管道时使用 `Configure` 方法。将应用程序启动代码放入 `Main` 方法。

现在首次运行该应用程序，创建数据库并使用测试数据作为种子数据。每当你更改数据模型时，可以删除数据库、更新你的 `Initialize` 方法，然后使用上述方式更新新数据库。在之后的教程中，你将了解如何在数据模型更改时，只需修改数据库而无需删除重建数据库。

创建控制器和视图

接下来，将使用 Visual Studio 中的基架引擎添加一个 MVC 控制器，以及使用 EF 来查询和保存数据的视图。

CRUD 操作方法和视图的自动创建被称为基架。基架与代码生成不同，基架的代码是一个起点，可以修改基架以满足自己需求，而你通常无需修改生成的代码。当你需要自定义生成代码时，可使用一部分分类或需求发生变化时重新生成代码。

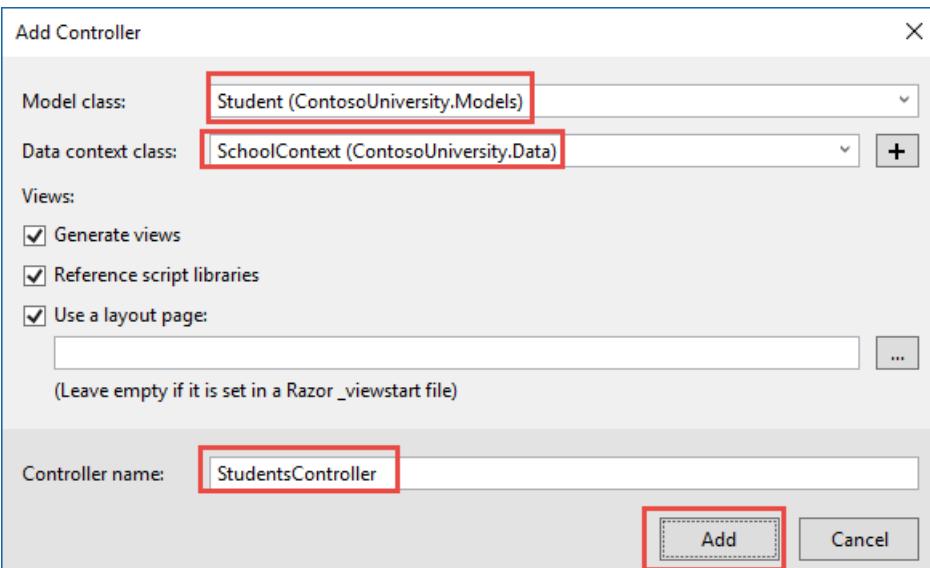
- 右键单击 解决方案资源管理器 中的 **Controllers** 文件夹选择 **添加 > 新搭建基架的项目**。

如果出现“添加 MVC 依赖项”对话框：

- 将 Visual Studio 更新到最新版本**。15.5 之前的 Visual Studio 版本显示此对话框。
- 如果无法更新，请选择“添加”，然后再次按照添加控制器步骤操作。
- 在“添加基架”对话框中：
 - 选择 视图使用 Entity Framework 的 MVC 控制器。
 - 单击 **添加**。
- 在“添加控制器”对话框中：
 - 在 **模型类** 选择 **Student**。
 - 在“数据上下文类”中选择 **SchoolContext**。

- 使用 **StudentsController** 作为默认名称。

- 单击 **添加**。



当你单击 **添加** 后, Visual Studio 基架引擎创建 *StudentsController.cs* 文件和一组对应于控制器的视图 (.cshtml 文件)。

(如果你之前手动创建数据库上下文, 基架引擎还可以自动创建。你可以在 **添加控制器** 对话框中单击右侧的加号框 **数据上下文类** 来指定在一个新上下文类。然后, Visual Studio 将创建你的 **DbContext**, 控制器和视图类。)

注意控制器将 **SchoolContext** 作为构造函数参数。

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
    }
}
```

ASP.NET 依赖注入机制会传递一个 **SchoolContext** 实例到控制器。在前面的教程中已经通过修改 *Startup.cs* 文件来配置注入规则。

控制器包含 **Index** 操作方法, 用于显示数据库中的所有学生。该方法从学生实体集中获取学生列表, 学生实体集则是通过读取数据库上下文实例中的 **Students** 属性获得:

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

本教程后面部分将介绍此代码中的异步编程元素。

Views/Students/Index.cshtml 视图使用table标签显示此列表:

```

@model IEnumerable<ContosoUniversity.Models.Student>

 @{
     ViewData["Title"] = "Index";
 }

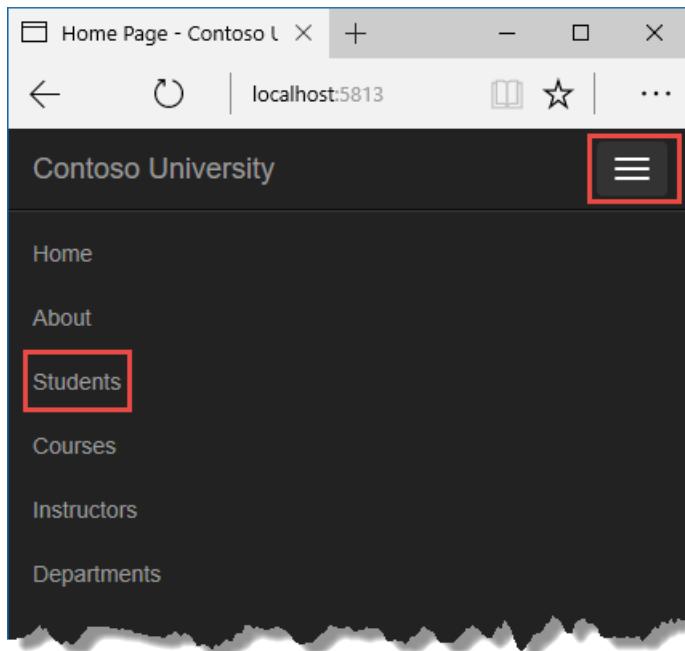
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.EnrollmentDate)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>

```

按 CTRL + F5 来运行该项目，或从菜单选择 调试 > 开始执行(不调试)。

单击学生选项卡以查看 `DbInitializer.Initialize` 插入的测试的数据。你将看到 `Student` 选项卡链接在页的顶部或在单击右上角后的导航图标中，具体显示在哪里取决于浏览器窗口宽度。



Last Name	First Name	Middle Name	Enrollment Date	Action
Alexander	Carson		9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith		9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo		9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis		9/1/2002 12:00:00 AM	Edit Details Delete

查看数据库

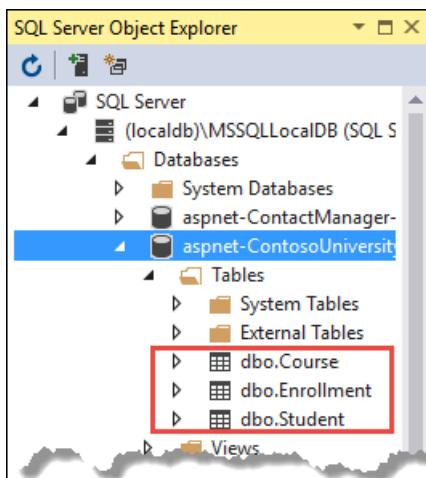
当你启动了应用程序, `DbInitializer.Initialize` 方法调用 `EnsureCreated`。EF 没有检测到相关数据库, 因此自己创建了一个, 接着 `Initialize` 方法的其余代码向数据库中填充数据。你可以使用 Visual Studio 中的 **SQL Server 对象资源管理器 (SSOX)** 查看数据库。

关闭浏览器。

如果 SSOX 窗口尚未打开, 请从Visual Studio 中的视图 菜单中选择。

在 SSOX 中, 单击 **(localdb) \MSSQLLocalDB > 数据库**, 然后单击和 `appsettings.json` 文件中的连接字符串对应的数据库。

展开“表”节点, 查看数据库中的表。



右键单击 **Student** 表，然后单击 **查看数据**，即可查看已创建的列和已插入到表的行。

	ID	EnrollmentDate	FirstMidName	LastName
▶	1	9/1/2005 12:00:...	Carson	Alexander
	2	9/1/2002 12:00:...	Meredith	Alonso
	3	9/1/2003 12:00:...	Arturo	Anand
	4	9/1/2002 12:00:...	Gytis	Barzdukas
	5	9/1/2002 12:00:...	Yan	Li

.mdf 和 .ldf 数据库文件位于 C:\Users\ 文件夹中。

因为调用 `EnsureCreated` 的初始化方法在启动应用程序时才运行，所以在这之前你可以更改 `Student` 类、删除数据库、再运行一次应用程序，这时候数据库将自动重新创建，以匹配所做的更改。例如，如果向 `Student` 类添加 `EmailAddress` 属性，重新的创建表中会有 `EmailAddress` 列。

约定

由于 Entity Framework 有一定的约束条件，你只需要按规则编写很少的代码就能够创建一个完整的数据库。

- `DbSet` 类型的属性用作表名。如果实体未被 `DbSet` 属性引用，实体类名称用作表名称。
- 使用实体属性名作为列名。
- 以 `ID` 或 `classnameID` 命名的实体属性被视为主键属性。
- 如果属性名为 将被解释为外键属性（例如，`StudentID` 对应 `Student` 导航属性，`Student` 实体的主键是 `ID`，所以 `StudentID` 被解释为外键属性）。此外也可以将外键属性命名为（例如，`EnrollmentID`，由于 `Enrollment` 实体的主键是 `EnrollmentID`，因此被解释为外键）。

约定行为可以重写。例如，本教程前半部分显式指定表名称。本系列 [后面教程](#) 则设置列名称并将任何属性设置为主键或外键。

异步代码

异步编程是 ASP.NET Core 和 EF Core 的默认模式。

Web 服务器的可用线程是有限的，而在高负载情况下的可能所有线程都被占用。当发生这种情况的时候，服务器就无法处理新请求，直到线程被释放。使用同步代码时，可能会出现多个线程被占用但不能执行任何操作的情况，因为它们正在等待 I/O 完成。使用异步代码时，当进程正在等待 I/O 完成，服务器可以将其线程释放用于处理其他请求。因此，异步代码使得服务器更有效地使用资源，并且该服务器可以无延迟地处理更多流量。

异步代码在运行时，会引入的少量开销，在低流量时对性能的影响可以忽略不计，但在针对高流量情况下潜在的性能提升是可观的。

在以下代码中，`async` 关键字、`Task<T>` 返回值、`await` 关键字和 `ToListAsync` 方法让代码异步执行。

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- `async` 关键字用于告知编译器该方法主体将生成回调并自动创建 `Task<IActionResult>` 返回对象。
- 返回类型 `Task<IActionResult>` 表示正在进行的工作返回的结果为 `IActionResult` 类型。
- `await` 关键字会使得编译器将方法拆分为两个部分。第一部分是以异步方式结束已启动的操作。第二部分是当操作完成时注入调用回调方法的地方。
- `ToListAsync` 是 `ToList` 方法的异步扩展版本。

使用 Entity Framework 编写异步代码时的一些注意事项：

- 只有导致查询或发送数据库命令的语句才能以异步方式执行。包括 `ToListAsync`，`SingleOrDefaultAsync`，和 `SaveChangesAsync`。不包括只需更改 `IQueryable` 的语句，如
`var students = context.Students.Where(s => s.LastName == "Davolio")`。
- EF 上下文是线程不安全的：请勿尝试并行执行多个操作。当调用异步 EF 方法时，始终使用 `await` 关键字。
- 如果你想要利用异步代码的性能优势，请确保你所使用的任何库和包在它们调用导致 Entity Framework 数据库查询方法时也使用异步。

有关在 .NET 异步编程的详细信息，请参阅 [异步概述](#)。

总结

你现在已创建了一个使用 Entity Framework Core 和 SQL Server Express LocalDB 来存储和显示数据的简单应用程序。在下一个教程中，你将学习如何执行基本的 CRUD（创建、读取、更新、删除）操作。

[下一篇](#)

ASP.NET Core MVC 和 EF Core 教程 - 创建、读取、更新和删除 (2/10)

2018/5/17 • 22 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Rick Anderson

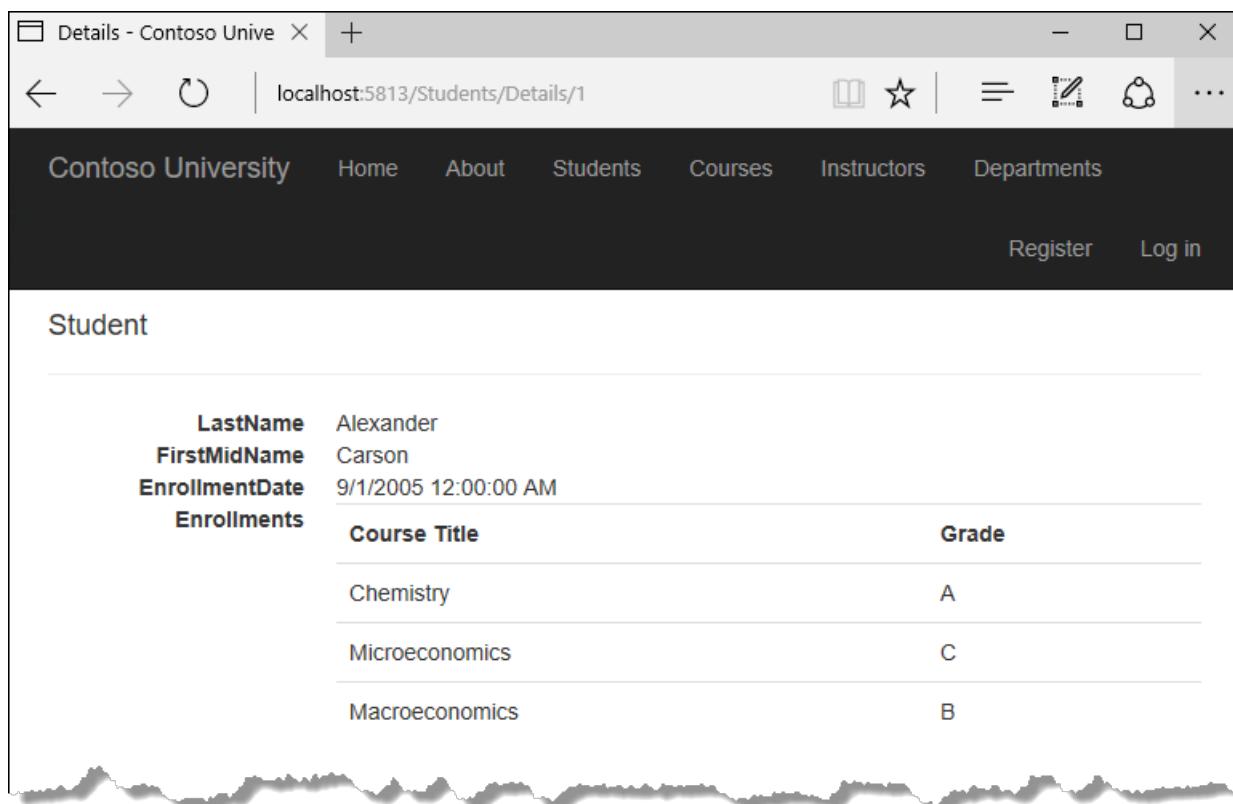
Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

在上一个教程中, 创建了一个使用 Entity Framework 和 SQL Server LocalDB 来存储和显示数据的 MVC 应用程序。在本教程中, 将评审和自定义 MVC 基架在控制器和视图中自动创建的 CRUD (创建、读取、更新、删除) 代码。

注意

为了在控制器和数据访问层之间创建一个抽象层, 常见的做法是实现存储库模式。为了保持这些教程内容简单并重点介绍如何使用 Entity Framework 本身, 它们不使用存储库。有关存储库和 EF 的信息, 请参阅[本系列中的最后一个教程](#)。

在本教程中, 将使用以下网页:



The screenshot shows a Microsoft Edge browser window with the title "Details - Contoso Univne". The address bar displays "localhost:5813/Students/Details/1". The page content is titled "Student". It shows a table with student information and enrollment details.

Last Name	Alexander	
First/Middle Name	Carson	
Enrollment Date	9/1/2005 12:00:00 AM	
Enrollments	Course Title	Grade
	Chemistry	A
	Microeconomics	C
	Macroeconomics	B

Create X + - □ ×

← ⌂ s/Create ⚒ ☆ ...

Contoso University Ⓛ

Create

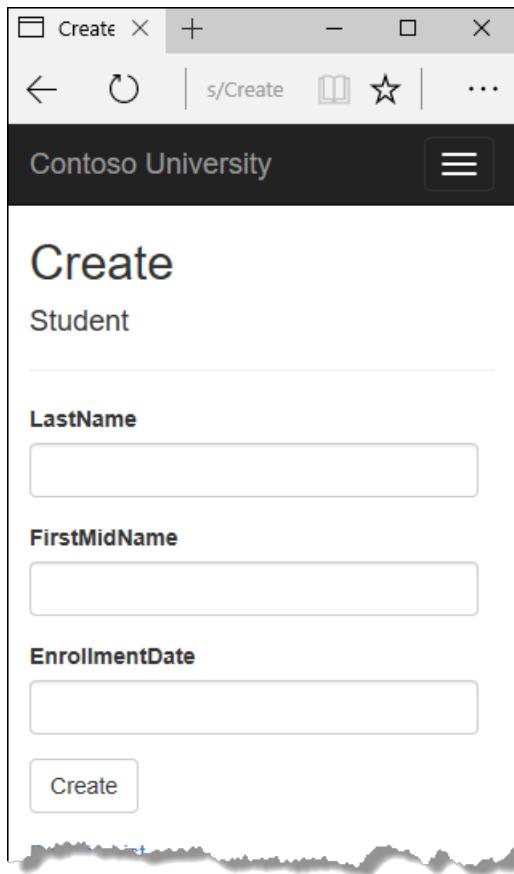
Student

Last Name

First Mid Name

Enrollment Date

Create



Edit - c X + - □ ×

localhost: Ⓛ ☆ ...

Contoso University Ⓛ

Edit

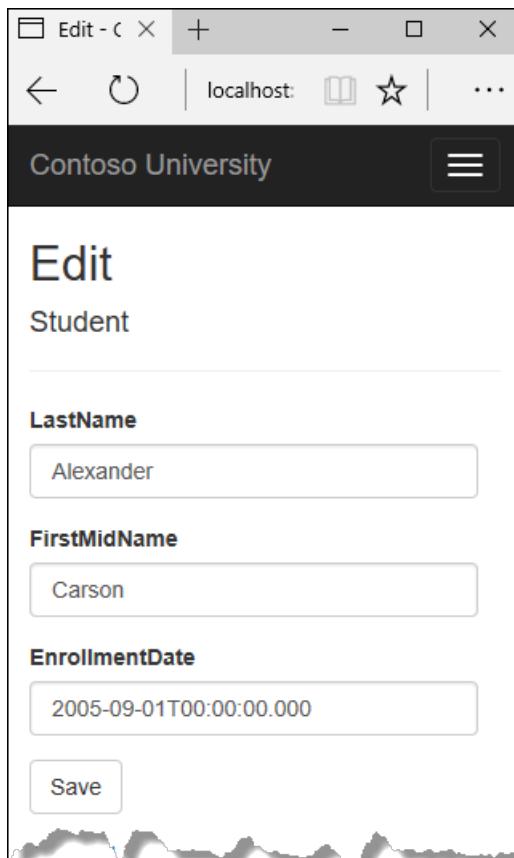
Student

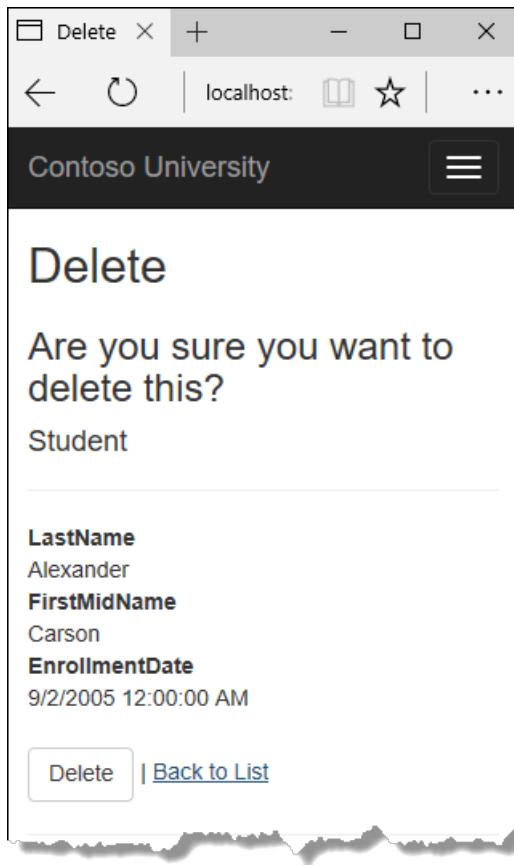
Last Name

First Mid Name

Enrollment Date

Save





自定义“详细信息”页

学生索引页的基架代码省略了 `Enrollments` 属性，因为该属性包含一个集合。在“详细信息”页上，将以 HTML 表形式显示集合的内容。

在 `Controllers/StudentsController.cs` 中，“详细信息”视图的操作方法使用 `SingleOrDefaultAsync` 方法检索单个 `Student` 实体。添加调用 `Include` 的代码。`ThenInclude` 和 `AsNoTracking` 方法，如以下突出显示的代码所示。

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

    var student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

`Include` 和 `ThenInclude` 方法使上下文加载 `Student.Enrollments` 导航属性，并在每个注册中加载 `Enrollment.Course` 导航属性。有关这些方法的详细信息，请参阅[读取相关数据](#)教程。

对于返回的实体未在当前上下文生存期中更新的情况，`AsNoTracking` 方法将会提升性能。本教程末尾将介绍有关 `AsNoTracking` 的详细信息。

路由数据

传递到 `Details` 方法的键值来自路由数据。路由数据是模型绑定器在 URL 的段中找到的数据。例如，默认路由指定控制器、操作和 ID 段：

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

在下面的 URL 中，默认路由将 `Instructor` 映射作为控制器、`Index` 作为操作，`1` 作为 ID；这些都是路由数据值。

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

URL 的最后部分 ("?courseID=2021") 是一个查询字符串。如果将 `id` 作为查询字符串值传递，模型绑定器也会将 ID 值作为参数传递给 `Details` 方法：

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

在索引页中，超链接 URL 由 Razor 视图中的标记帮助器语句创建。在以下 Razor 代码中，`id` 参数与默认路由相匹配，因此 `id` 将添加到路由数据。

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

`item.ID` 为 6 时，会生成以下 HTML：

```
<a href="/Students/Edit/6">Edit</a>
```

在以下 Razor 代码中，`studentID` 与默认路由中的参数不匹配，所以将它作为查询字符串添加。

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

`item.ID` 为 6 时，会生成以下 HTML：

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

有关标记帮助器的详细信息，请参阅 [ASP.NET Core 中的标记帮助器](#)。

将注册添加到“详细信息”视图

打开 `Views/Students/Details.cshtml`。每个字段都使用 `DisplayNameFor` 和 `DisplayFor` 帮助器来显示，如下面的示例中所示：

```
<dt>
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd>
    @Html.DisplayFor(model => model.LastName)
</dd>
```

在最后一个字段之后和 `</dl>` 闭合标记之前，添加以下代码以显示注册列表：

```

<dt>
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd>
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>

```

如果代码缩进在粘贴代码后出现错误, 请按 CTRL+K+D 进行更正。

此代码循环通过 `Enrollments` 导航属性中的实体。它将针对每个注册显示课程标题和成绩。课程标题从 `Course` 实体中检索, 该实体存储在 `Enrollments` 实体的 `Course` 导航属性中。

运行应用, 选择“学生”选项卡, 然后单击学生的“详细信息”链接。将看到所选学生的课程和年级列表:

	Course Title	Grade
Chemistry	A	
Microeconomics	C	
Macroeconomics	B	

更新“创建”页

在 `StudentsController.cs` 中修改 `HttpPost [Create]` 方法, 在 `Bind` 特性中添加 `try catch` 块并删除 `ID` 值。

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}

```

此代码将 ASP.NET MVC 模型绑定器创建的 Student 实体添加到 Students 实体集，然后将更改保存到数据库。（模型绑定器指的是 ASP.NET MVC 功能，用户可利用它来轻松处理使用表单提交的数据；模型绑定器将已发布的表单值转换为 CLR 类型，并将其传递给操作方法的参数。在本例中，模型绑定器将使用 Form 集合的属性值实例化 Student 实体。）

已从 Bind 特性删除 ID，因为 ID 是插入行时 SQL Server 将自动设置的主键值。来自用户的输入不会设置 ID 值。

除了 Bind 特性，try-catch 块是对基架代码所做的唯一更改。如果保存更改时捕获到来自 DbUpdateException 的异常，则会显示一般错误消息。有时 DbUpdateException 异常是由应用程序外部的某些内容而非编程错误引起的，因此建议用户再次尝试。尽管在本示例中未实现，但生产质量应用程序会记录异常。有关详细信息，请参阅[监视和遥测\(使用 Azure 构建真实世界云应用\)](#)中的“见解记录”部分。

ValidateAntiForgeryToken 特性帮助抵御跨网站请求伪造 (CSRF) 攻击。令牌通过 FormTagHelper 自动注入到视图中，并在用户提交表单时包含该令牌。令牌由 ValidateAntiForgeryToken 特性验证。有关 CSRF 的详细信息，请参阅[反请求伪造](#)。

有关过多发布的安全说明

基架代码包含在 Create 方法中的 Bind 特性是防止在创建方案中过多发布的一种方法。例如，假设 Student 实体包含不希望此网页设置的 Secret 属性。

```

public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}

```

即使网页上没有 Secret 字段，黑客也可以使用 Fiddler 之类的工具，或者编写一些 JavaScript 来发布 Secret 表单值。创建 Student 实例时，如果不利用 Bind 特性来限制模型绑定器使用的字段，模型绑定器会选取该 Secret 表单值并使用它来创建 Student 实体实例。然后将在数据库中更新黑客为 Secret 表单字段指定的任意值。下图显示 Fiddler 工具正在将 Secret 字段(值为“OverPost”)添加到已发布的表单值。

The screenshot shows the Fiddler interface with a red box highlighting the 'Execute' button. The 'Request Headers' section contains standard HTTP headers like Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Referer, Cookie, Connection, Content-Type, and Content-Length. The 'Request Body' section shows a URL-encoded form with parameters: s1&LastName=Smith&FirstName=Joe&EnrollmentDate=7%2F31%2F2013+4%3A58%3A23+PM&Secret=OverPost. The 'Secret=OverPost' part is specifically highlighted with a red box.

然后值“OverPost”将成功添加到插入行的 `Secret` 属性，尽管你从未打算网页可设置该属性。

可以防止在编辑方案中过多发布，方法是首先从数据库读取实体，然后调用 `TryUpdateModel` 并在显式允许的属性列表中传递。这些教程中使用的也是这种方法。

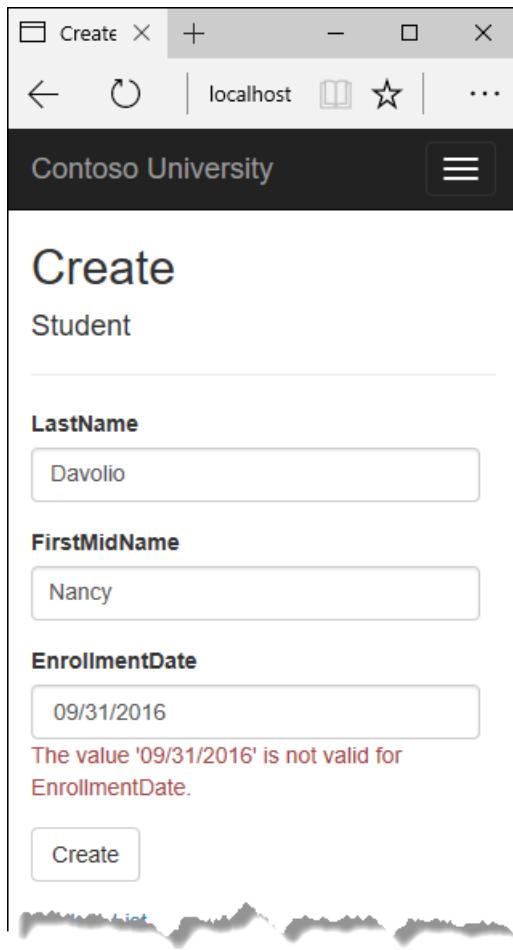
许多开发者首选的防止过多发布的另一种方法是使用视图模型，而不是包含模型绑定的实体类。仅包含想要在视图模型中更新的属性。完成 MVC 模型绑定器后，根据需要使用 AutoMapper 之类的工具将视图模型属性复制到实体实例。使用实体实例上的 `_context.Entry` 将其状态设置为 `Unchanged`，然后在视图模型中包含的每个实体属性上将 `Property("PropertyName").IsModified` 设置为 `true`。此方法同时适用于编辑和创建方案。

测试创建页

Views/Students/Create.cshtml 中的代码对每个字段使用 `label`、`input` 和 `span`（适用于验证消息）标记帮助器。

运行应用，选择“学生”选项卡，并单击“新建”。

输入姓名和日期。如果浏览器允许输入无效日期，请尝试输入。（某些浏览器强制要求使用日期选取器。）然后单击“创建”，查看错误消息。



这是默认获取的服务器端验证;在下一个教程中,还将介绍如何添加生成客户端验证代码的特性。以下突出显示的代码显示 `Create` 方法中的模型验证检查。

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

将日期更改为有效值,并单击“创建”,查看“索引”页中显示的新学生。

更新“编辑”页

在 `StudentController.cs` 中, `HttpGet Edit` 方法(不具有 `HttpPost` 特性)使用 `SingleOrDefaultAsync` 方法检索所

选的 Student 实体, 如 Details 方法中所示。不需要更改此方法。

建议的 **HttpPost** 编辑代码: 读取和更新

使用以下代码替换 **HttpPost Edit** 操作方法。

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.SingleOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

这些更改实现安全最佳做法, 防止过多发布。基架生成了 **Bind** 特性, 并将模型绑定器创建的实体添加到具有 **Modified** 标记的实体集。不建议将该代码用于多个方案, 因为 **Bind** 特性将清除未在 **Include** 参数中列出的字段中的任何以前存在的数据。

新代码读取现有实体并调用 **TryUpdateModel**, 以基于已发布表单数据中的用户输入更新已检索实体中的字段。Entity Framework 的自动更改跟踪在由表单输入更改的字段上设置 **Modified** 标记。调用 **SaveChanges** 方法时, Entity Framework 会创建 SQL 语句, 以更新数据库行。忽略并发冲突, 并且仅在数据库中更新由用户更新的表列。(下一个教程将介绍如何处理并发冲突。)

作为防止过多发布的最佳做法, 请将希望通过“编辑”页更新的字段列入 **TryUpdateModel** 参数。(参数列表中字段列表之前的空字符串用于与表单字段名称一起使用的前缀。)目前没有要保护的额外字段, 但是列出希望模型绑定器绑定的字段可确保以后将字段添加到数据模型时, 它们将自动受到保护, 直到明确将其添加到此处为止。

这些更改会导致 **HttpPost Edit** 方法与 **HttpGet Edit** 方法的方法签名相同, 因此已重命名 **EditPost** 方法。

可选 **HttpPost** 编辑代码: 创建和附加

建议的 **HttpPost** 编辑代码确保只更新已更改的列, 并保留不希望包含在模型绑定内的属性中的数据。但是, 读取优先的方法需要额外的数据库读取, 并可能产生处理并发冲突的更复杂代码。另一种方法是将模型绑定器创建的实体附加到 EF 上下文, 并将其标记为已修改。(请勿使用此代码更新项目, 它只是显示一种可选的方法。)

```

public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student
student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}

```

网页 UI 包含实体中的所有字段并能更新其中任意字段时，可以使用此方法。

基架代码使用创建和附加方法，但仅捕获 `DbUpdateConcurrencyException` 异常并返回 404 错误代码。显示的示例捕获任意数据库更新异常并显示错误消息。

实体状态

数据库上下文跟踪内存中的实体是否与数据库中相应的行同步，并且此信息确定调用 `SaveChanges` 方法时会发生的情况。例如，将新实体传递到 `Add` 方法时，该实体的状态会设置为 `Added`。然后调用 `SaveChanges` 方法时，数据库上下文发出 SQL INSERT 命令。

实体可能处于以下状态之一：

- `Added`。数据库中尚不存在实体。`SaveChanges` 方法发出 INSERT 语句。
- `Unchanged`。不需要通过 `SaveChanges` 方法对此实体执行操作。从数据库读取实体时，实体将从此状态开始。
- `Modified`。已修改实体的部分或全部属性值。`SaveChanges` 方法发出 UPDATE 语句。
- `Deleted`。已标记该实体进行删除。`SaveChanges` 方法发出 DELETE 语句。
- `Detached`。数据库上下文未跟踪该实体。

在桌面应用程序中，通常会自动设置状态更改。读取一个实体并对其某些属性值做出更改。这将使其实体状态自动更改为 `Modified`。然后调用 `SaveChanges` 时，Entity Framework 生成 SQL UPDATE 语句，该语句仅更新已更改的实际属性。

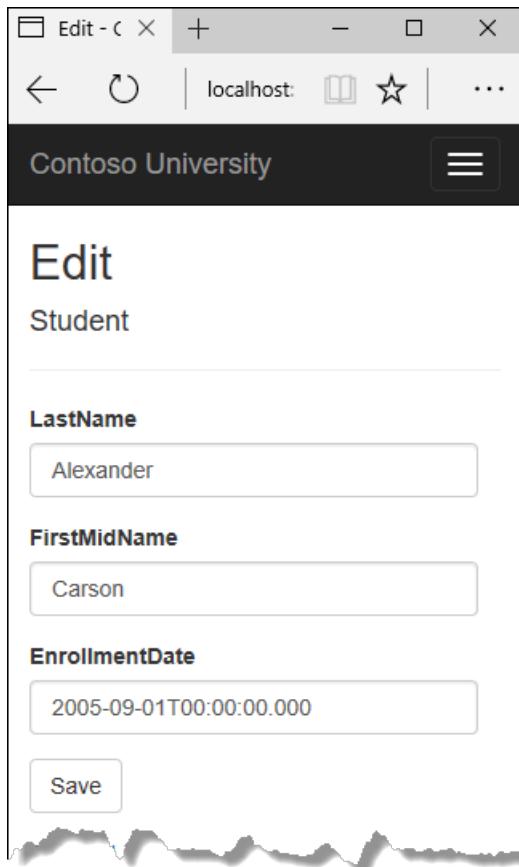
在 Web 应用中，最初读取实体并显示其要编辑的数据的 `DbContext` 将在页面呈现后进行处理。调用 `HttpPost Edit` 操作方法时，将发出新的 Web 请求并且具有 `DbContext` 的新实例。如果在新的上下文中重新读取实体，则将模拟桌面处理。

但如果希望进行额外的读取操作，则必须使用模型绑定器创建的实体对象。执行此操作最简单的方法是将实体状态设置为“已修改”，就像在之前所示的替代 `HttpPost` 编辑代码中完成的一样。然后调用 `SaveChanges` 时，Entity Framework 会更新数据库行的所有列，因为上下文无法知道已更改的属性。

如果想避免读取优先的方法，但还希望 SQL UPDATE 语句只更新用户实际更改的字段，则代码将更复杂。调用 `HttpPost` `Edit` 方法时，必须以某种方式保存初始值（如使用隐藏字段），以便初始值可用。然后可以使用初始值创建 `Student` 实体、调用具有初始实体版本的 `Attach` 方法、将实体的值更新为新值，再调用 `SaveChanges`。

测试编辑页

运行应用，选择“学生”选项卡，然后单击“编辑”超链接。



更改某些数据并单击“保存”。将打开“索引”页，将看到已更改的数据。

更新“删除”页

在 `StudentController.cs` 中，`HttpGet` `Delete` 方法的模板代码使用 `SingleOrDefaultAsync` 方法来检索所选的 `Student` 实体，如 `Details` 和 `Edit` 方法中所示。但是，若要在调用 `SaveChanges` 失败时实现自定义错误消息，请将部分功能添加到此方法及其相应的视图中。

正如所看到的更新和创建操作，删除操作需要两个操作方法。为响应 GET 请求而调用的方法将显示一个视图，使用户有机会批准或取消操作。如果用户批准，则创建 POST 请求。发生此情况时，将调用 `HttpPost` `Delete` 方法，然后该方法实际执行删除操作。

将 try-catch 块添加到 `HttpPost` `Delete` 方法，以处理更新数据库时可能出现的任何错误。如果发生错误，`HttpPost Delete` 方法会调用 `HttpGet Delete` 方法，并向其传递一个指示发生错误的参数。然后 `HttpGet Delete` 方法重新显示确认页以及错误消息，向用户提供取消或重试的机会。

使用以下管理错误报告的代码替换 `HttpGet` `Delete` 操作方法。

```

public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}

```

此代码接受可选参数，指示保存更改失败后是否调用此方法。没有失败的情况下调用 `HttpGet` `Delete` 方法时，此参数为 `false`。由 `HttpPost` `Delete` 方法调用以响应数据库更新错误时，此参数为 `true`，并且将错误消息传递到视图。

HttpPost Delete 的读取优先方法

使用以下执行实际删除操作并捕获任何数据库更新错误的代码替换 `HttpPost` `Delete` 操作方法（名为 `DeleteConfirmed`）。

```

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}

```

此代码检索所选的实体，然后调用 `Remove` 方法以将实体的状态设置为 `Deleted`。调用 `SaveChanges` 时生成 SQL `DELETE` 命令。

HttpPost Delete 的创建和附加方法

如果在大容量应用程序中提高性能是优先事项，则可以通过只使用主键值实例化 Student 实体，然后将实体状态设置为 Deleted 来避免不必要的 SQL 查询。这是 Entity Framework 删除实体需要执行的所有操作。(请勿将此代码放在项目中;这里只是为了说明替代方法。)

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

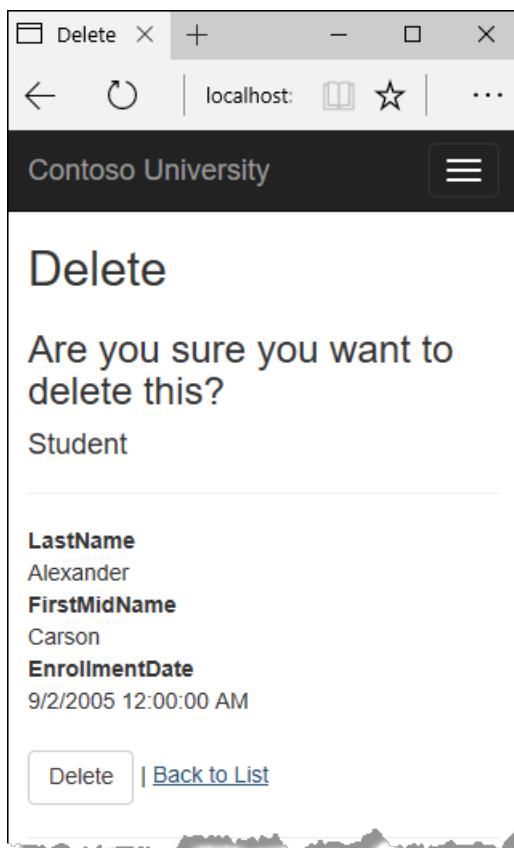
如果实体包含还应删除的相关数据，请确保在数据库中配置了级联删除。若使用此方法删除实体，EF 可能不知道有需要删除的相关实体。

更新“删除”视图

在 Views/Student/Delete.cshtml 中，在 H2 标题和 H3 标题之间添加错误消息，如以下示例所示：

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

运行应用，选择“学生”选项卡，并单击“删除”超链接：



单击“删除”。将显示不含已删除学生的索引页。(将看到并发教程中错误处理代码的效果示例。)

关闭数据库连接

若要释放数据库连接包含的资源，完成此操作时必须尽快处理上下文实例。ASP.NET Core 内置[依赖关系注入](#)会完成此任务。

在 Startup.cs 中，调用 [AddDbContext 扩展方法](#) 来预配 ASP.NET DI 容器的 `DbContext` 类。默认情况下，该方法将服务生存期设置为 `Scoped`。`Scoped` 表示上下文对象生存期与 Web 请求生存期一致，并在 Web 请求结束时将自动调用 `Dispose` 方法。

处理事务

默认情况下，Entity Framework 隐式实现事务。在对多个行或表进行更改并调用 `SaveChanges` 的情况下，Entity Framework 自动确保所有更改都成功或全部失败。如果完成某些更改后发生错误，这些更改会自动回退。如果需要更多控制操作（例如，如果想要在事务中包含在 Entity Framework 外部完成的操作），请参阅[事务](#)。

非跟踪查询

当数据库上下文检索表行并创建表示它们的实体对象时，默认情况下，它会跟踪内存中的实体是否与数据库中的内容同步。更新实体时，内存中的数据充当缓存并使用该数据。在 Web 应用程序中，此缓存通常是不必要的，因为上下文实例通常生存期较短（创建新的实例并用于处理每个请求），并且通常在再次使用该实体之前处理读取实体的上下文。

可以通过调用 `AsNoTracking` 方法禁用对内存中的实体对象的跟踪。可能想要执行的典型方案包括以下操作：

- 在上下文生存期内，不需要更新任何实体，并且不需要 EF [自动加载具有由单独的查询检索的实体的导航属性](#)。在控制器的 `HttpGet` 操作方法中经常遇到这些情况。
- 正在运行检索大量数据的查询，将只更新一小部分返回的数据。关闭对大型查询的跟踪可能更有效，稍后为少数需要更新的实体运行查询。
- 想要附加一个实体来更新它，但之前为了其他目的，已检索了相同的实体。由于数据库上下文已跟踪了该实体，因此无法附加要更改的实体。处理这种情况的一种方法是在早前的查询上调用 `AsNoTracking`。

有关详细信息，请参阅[跟踪与非跟踪](#)。

总结

现在拥有一组完整的页面，可对 `Student` 实体执行简单的 CRUD 操作。在下一个教程中，将通过添加排序、筛选和分页来扩展“索引”页。

[上一页](#)

[下一页](#)

ASP.NET Core MVC 和 EF Core - 排序、筛选、分页 - 第 3 个教程, 共 10 个教程

2018/5/17 • 16 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

在上一个教程中, 已为 Student 实体实现了一组网页用于执行基本的 CRUD 操作。在本教程中, 将向学生索引页添加排序、筛选和分页功能。同时, 还将创建一个执行简单分组的页面。

下图展示了完成操作后的页面外观。列标题是用户可以单击以按该列排序的链接。重复单击列标题可在升降排序顺序之间切换。

The screenshot shows a browser window titled "Index - Contoso Univers" with the URL "localhost:5813/Student". The page header includes a back arrow, forward arrow, refresh icon, a star icon, and a menu icon. The main content area has a dark header with the text "Contoso University". Below it, the word "Index" is displayed in a large, bold, dark font. Underneath, there is a "Create New" link and a search bar with the placeholder "Find by name:" and a "Search" button. A link "Back to Full List" is also present. The main table has three columns: "Last Name", "First Name", and "Enrollment Date". Each row contains three entries: Alexander Carson (9/1/2005), Alonso Meredith (9/1/2002), and Anand Arturo (9/1/2003). To the right of each entry are three links: "Edit", "Details", and "Delete". At the bottom of the table, there are "Previous" and "Next" navigation buttons.

Last Name	First Name	Enrollment Date
Alexander	Carson	9/1/2005 12:00:00 AM
Alonso	Meredith	9/1/2002 12:00:00 AM
Anand	Arturo	9/1/2003 12:00:00 AM

向学生索引页添加列排序链接

要向学生索引页添加排序功能, 需更改学生控制器的 `Index` 方法并将代码添加到学生索引视图。

向 `Index` 方法添加排序功能

在 `StudentsController.cs` 中, 将 `Index` 方法替换为以下代码:

```

public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                  select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

此代码接收来自 URL 中的查询字符串的 `sortOrder` 参数。查询字符串值由 ASP.NET Core MVC 提供，作为操作方法的参数。该参数将是一个字符串，可为“Name”或“Date”，可选择后跟下划线和字符串“desc”来指定降序。默认排序顺序为升序。

首次请求索引页时，没有任何查询字符串。学生按照姓氏升序显示，这是 `switch` 语句中的 fall-through 事例所建立的默认值。当用户单击列标题超链接时，查询字符串中会提供相应的 `sortOrder` 值。

视图使用两个 `ViewData` 元素（`NameSortParm` 和 `DateSortParm`）来为列标题超链接配置相应的查询字符串值。

```

public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                  select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

这些是三元语句。第一个指定如果 `sortOrder` 参数为 NULL 或空，则应将 `NameSortParm` 设置为“`name_desc`”；否则，应将其设置为空字符串。通过这两个语句，视图可如下设置列标题超链接：

当前排序顺序	姓氏超链接	日期超链接
姓氏升序	descending	ascending
姓氏降序	ascending	ascending
日期升序	ascending	descending
日期降序	ascending	ascending

该方法使用 LINQ to Entities 指定要作为排序依据的列。该代码在 `switch` 语句之前创建一个 `IQueryable` 变量，在 `switch` 语句中对其进行修改，并在 `switch` 语句后调用 `ToListAsync` 方法。当创建和修改 `IQueryable` 变量时，不会向数据库发送任何查询。只有通过调用 `ToListAsync` 之类的方法将 `IQueryable` 对象转换为集合，查询才会执行。因此，此代码导致直到 `return View` 语句才会执行单个查询。

此代码可能获得包含大量列的详细信息。[本系列的最后一个教程](#)演示了如何编写在字符串变量中传递 `OrderBy` 列名的代码。

向“学生索引”视图添加列标题超链接

将 Views / Students / Index.cshtml 中的代码替换为以下代码，以添加列标题超链接。已更改的行为突出显示状态。

```

@model IEnumerable<ContosoUniversity.Models.Student>

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
    </tbody>
</table>

```

此代码使用 `ViewData` 属性中的信息来设置具有相应查询字符串值的超链接。

运行应用，选择“学生”选项卡，然后单击“姓氏”和“注册日期”列标题以验证排序是否正常工作。

Last Name	First/Middle Name	Enrollment Date	
Alexander	Carson	9/2/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete

向“学生索引”页添加搜索框

要向学生索引页添加筛选功能，需将文本框和提交按钮添加到视图，并在 `Index` 方法中做出相应的更改。在文本框中输入一个字符串以在名字和姓氏字段中进行搜索。

向 `Index` 方法添加筛选功能

在 `StudentsController.cs` 中，将 `Index` 方法替换为以下代码（所做的更改为突出显示状态）。

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                  select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                           || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

已向 `Index` 方法添加 `searchString` 参数。从要添加到索引视图的文本框中接收搜索字符串值。并且，还向

LINQ 语句添加了 where 子句，该子句仅选择名字或姓氏中包含搜索字符串的学生。只有在有搜索值的情况下，才会执行添加了 where 子句的语句。

注意

此处正在调用 `IQueryable` 对象上的 `Where` 方法，并且将在服务器上处理该筛选器。在某些情况下，可能会将 `Where` 方法作为内存中集合上的扩展方法进行调用。(例如，假设将引用更改为 `_context.Students`，这样它引用的不再是 EF `DbSet`，而是一个返回 `IEnumerable` 集合的存储库方法。)结果通常是相同的，但在某些情况下可能不同。

例如，`Contains` 方法的 .NET Framework 实现在默认情况下执行区分大小写比较，但在 SQL Server 中，这由 SQL Server 实例的排序规则设置决定。该设置默认为不区分大小写。可调用 `ToUpper` 方法使测试显式不区分大小写：`Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))`。如果稍后更改代码以使用返回 `IEnumerable` 集合而不是 `IQueryable` 对象的存储库，则可确保结果保持不变。(在 `IEnumerable` 集合上调用 `Contains` 方法时，将获得 .NET Framework 实现；当在 `IQueryable` 对象上调用它时，将获得数据库提供程序实现。)但是，此解决方案会降低性能。`ToUpper` 代码将在 TSQL SELECT 语句的 WHERE 子句中放置一个函数。这将阻止优化器使用索引。鉴于 SQL 主要安装为不区分大小写，在迁移到区分大小写的数据存储之前，最好避免使用 `ToUpper` 代码。

向“学生索引”视图添加搜索框

在 Views/Student/Index.cshtml 中，请在打开表格标签之前立即添加突出显示的代码，以创建标题栏、文本框和搜索按钮。

```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

此代码使用 `<form>` 标记帮助器添加搜索文本框和按钮。默认情况下，`<form>` 标记帮助器使用 POST 提交表单数据，这意味着参数在 HTTP 消息正文传递，而不是作为查询字符串在 URL 中传递。当指定 HTTP GET 时，表单数据作为查询字符串在 URL 中传递，从而使用户能够将 URL 加入书签。W3C 指南建议在操作不会导致更新时使用 GET。

运行应用，选择“学生”选项卡，输入搜索字符串，然后单击“搜索”以验证筛选是否正常工作。

Last Name	First Mid Name	Enrollment Date	
Alexander	Carson	9/2/2005 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Li	Yan	9/1/2002 12:00:00 AM	Edit Details Delete
Norman	Laura	9/1/2003 12:00:00 AM	Edit Details Delete

请注意，该 URL 包含搜索字符串。

```
http://localhost:5813/Students?SearchString=an
```

如果将此页加入书签，当使用书签时，将获得已筛选的列表。向 `form` 标记添加 `method="get"` 是导致生成查询字符串的原因。

在此阶段，如果单击列标题排序链接，则会丢失已在“搜索”框中输入的筛选器值。此问题将在下一部分得以解决。

向“学生索引”页添加分页功能

要向学生索引页添加分页功能，需创建一个使用 `Skip` 和 `Take` 语句的 `PaginatedList` 类来筛选服务器上的数据，而不总是对表中的所有行进行检索。然后在 `Index` 方法中进行其他更改，并将分页按钮添加到 `Index` 视图。下图显示了分页按钮。

The screenshot shows a web browser window with the title "Index - Contoso Univers" and the URL "localhost:5813/Student". The page header includes a back arrow, forward arrow, refresh icon, and a search bar with placeholder text "Find by name: []" and a "Search" button. Below the header is a navigation menu with three horizontal bars. The main content area is titled "Index" and contains a "Create New" link. A search bar with placeholder text "Find by name: []" and a "Search" button is followed by a link "Back to Full List". A table lists student records:

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

At the bottom of the page are "Previous" and "Next" navigation buttons.

在项目文件夹中，创建 `PaginatedList.cs`，然后用以下代码替换模板代码。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}

```

此代码中的 `CreateAsync` 方法将提取页面大小和页码，并将相应的 `Skip` 和 `Take` 语句应用于 `IQueryable`。当在 `IQueryable` 上调用 `ToListAsync` 时，它将返回仅包含请求页的列表。属性 `HasPreviousPage` 和 `HasNextPage` 可用于启用或禁用“上一页”和“下一页”的分页按钮。

由于构造函数不能运行异步代码，因此使用 `CreateAsync` 方法来创建 `PaginatedList<T>` 对象，而非构造函数。

向 Index 方法添加分页功能

在 `StudentsController.cs` 中，将 `Index` 方法替换为以下代码。

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                  select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                           || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
}

```

该代码向方法签名中添加一个页码参数、一个当前排序顺序参数和一个当前筛选器参数。

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)

```

第一次显示页面时，或者如果用户没有单击分页或排序链接，所有参数都将为 NULL。如果单击了分页链接，页面变量将包含要显示的页码。

名为 CurrentSort 的 `ViewData` 元素为视图提供当前排序顺序，因为此值必须包含在分页链接中，以便在分页时保持排序顺序相同。

名为 CurrentFilter 的 `ViewData` 元素为视图提供当前筛选器字符串。此值必须包含在分页链接中，以便在分页过

程中保持筛选器设置，并且在页面重新显示时必须将其还原到文本框中。

如果在分页过程中搜索字符串发生变化，则页面必须重置为 1，因为新的筛选器会导致显示不同的数据。在文本框中输入值并按下“提交”按钮时，搜索字符串将被更改。在这种情况下，`searchString` 参数不为 NULL。

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

在 `Index` 方法最后，`PaginatedList.CreateAsync` 方法会将学生查询转换为支持分页的集合类型中的学生的单个页面。然后将学生的单个页面传递给视图。

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
```

`PaginatedList.CreateAsync` 方法需要一个页码。两个问号表示 NULL 合并运算符。NULL 合并运算符为可为 NULL 的类型定义默认值;表达式 `(page ?? 1)` 表示如果 `page` 有值, 则返回该值, 如果 `page` 为 NULL, 则返回 1。

向学生索引视图添加分页链接

在 Views/Students/Index.cshtml 中，将现有代码替换为以下代码。突出显示所作更改。

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
        </tr>
    <tbody>
        <tr>
            <td>Dwight K.</td>
            <td>Dwight</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Michael S.</td>
            <td>Michael</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Phyllis L.</td>
            <td>Phyllis</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Angela M.</td>
            <td>Angela</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Toby F.</td>
            <td>Toby</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Kevin B.</td>
            <td>Kevin</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Meredith P.</td>
            <td>Meredith</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Dwight K.</td>
            <td>Dwight</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Michael S.</td>
            <td>Michael</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Phyllis L.</td>
            <td>Phyllis</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Angela M.</td>
            <td>Angela</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Toby F.</td>
            <td>Toby</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Kevin B.</td>
            <td>Kevin</td>
            <td>1996-09-16</td>
        </tr>
        <tr>
            <td>Meredith P.</td>
            <td>Meredith</td>
            <td>1996-09-16</td>
        </tr>
    </tbody>
</table>
```

```

        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> | 
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> | 
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@{ModelPageIndex - 1}"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@{ModelPageIndex + 1}"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @nextDisabled">
    Next
</a>

```

页面顶部的 `@model` 语句指定视图现在获取的是 `PaginatedList<T>` 对象，而不是 `List<T>` 对象。

列标题链接使用查询字符串向控制器传递当前搜索字符串，以便用户可以在筛选结果中进行排序：

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
    ="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

分页按钮由标记帮助器显示：

```
<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@{ModelPageIndex - 1}"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>
```

运行应用并转到“学生”页。

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

单击不同排序顺序的分页链接，以确保分页正常工作。然后输入一个搜索字符串并再次尝试分页，以验证分页也可以正确地进行排序和筛选。

创建显示学生统计信息的“关于”页

对于 Contoso 大学网站的“关于”页，将显示每个注册日期注册了多少名学生。这需要对组进行分组和简单计算。若要完成此操作，需要执行以下操作：

- 为需要传递给视图的数据创建一个视图模型类。
- 修改主控制器中的“关于”方法。
- 修改“关于”视图。

创建视图模型

在 Models 文件夹中创建一个 SchoolViewModels 文件夹。

在新文件夹中，添加一个类文件 EnrollmentDateGroup.cs，并用以下代码替换模板代码：

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

修改主控制器

在 HomeController.cs 中，使用文件顶部的语句添加以下内容：

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

在类的左大括号之后立即为数据库上下文添加一个类变量，并从 ASP.NET Core DI 获取上下文的实例：

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
}
```

将 About 方法替换为以下代码：

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
    {
        EnrollmentDate = dateGroup.Key,
        StudentCount = dateGroup.Count()
    };
    return View(await data.AsNoTracking().ToListAsync());
}
```

LINQ 语句按注册日期对学生实体进行分组，计算每组中实体的数量，并将结果存储在 EnrollmentDateGroup 视图模型对象的集合中。

注意

在 1.0 版 Entity Framework Core 中，整个结果集被返回给客户端，并在客户端上完成分组。在某些情况下，这可能会造成性能问题。请务必使用生产数据量测试性能，必要时请使用原始 SQL 在服务器上进行分组。有关如何使用原始 SQL 的信息，请参阅本系列中的最后一个教程。

修改“关于”视图

将 Views/Home/About.cshtml 文件中的代码替换为以下代码：

```

@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}



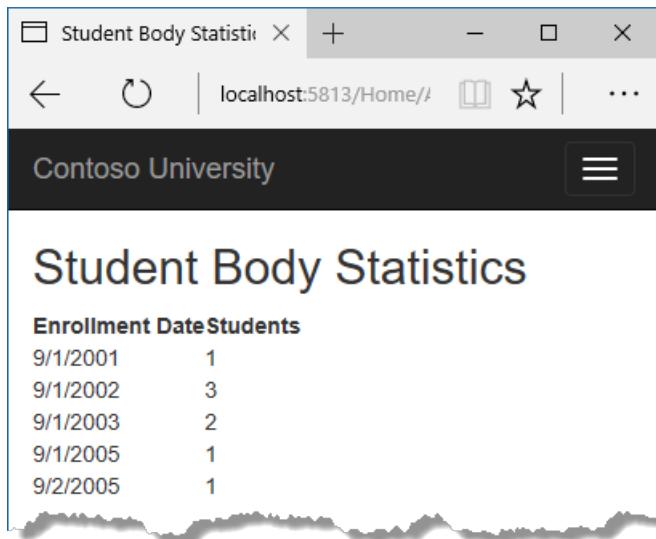
## Student Body Statistics



| Enrollment Date | Students |
|-----------------|----------|
|-----------------|----------|


```

运行应用并转到“关于”页。表格中会显示每个注册日期的学生计数。



总结

在本教程中，你已了解了如何执行排序、筛选、分页和分组。在下一个教程中，你将了解如何使用迁移来处理数据模型更改。

ASP.NET Core MVC 和 EF Core - 迁移 - 第 4 个教程(共 10 个)

2018/5/17 • 8 min to read • [Edit Online](#)

作者 : [Tom Dykstra](#) 和 [Rick Anderson](#)

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

本教程使用 EF Core 迁移功能管理数据模型更改。后续教程将在更改数据模型时添加更多迁移。

迁移简介

开发新应用程序时, 数据模型会频繁更改。每当模型更改时, 模型都无法与数据库保持同步。本系列教程首先配置 Entity Framework 以创建数据库(如果不存在)。之后, 每当更改数据模型(添加、删除或更改实体类或更改 DbContext 类)时, 你都可以删除数据库, EF 将创建匹配该模型的新数据库并用测试数据为其设定种子。

这种使数据库与数据模型保持同步的方法适用于多种情况, 但将应用程序部署到生产环境的情况除外。当应用程序在生产环境中运行时, 它通常会存储要保留的数据, 以便不会在每次更改(如添加新列)时丢失所有数据。EF Core 迁移功能可通过使 EF 更新数据库 架构而不是创建新数据库来解决此问题。

用于进行迁移的 Entity Framework Core NuGet 包

要使用迁移, 可使用“包管理器控制台”(PMC) 或命令行接口 (CLI)。以下教程演示如何使用 CLI 命令。有关 PMC 的信息, 请转到[本教程末尾](#)。

`Microsoft.EntityFrameworkCore.Tools.DotNet` 中提供了适用于命令行接口 (CLI) 的 EF 工具。若要安装此程序包, 请将它添加到 .csproj 文件中的 `DotNetCliToolReference` 集合, 如下所示。注意: 必须通过编辑 .csproj 文件来安装此包; 不能使用 `install-package` 命令或程序包管理器 GUI。若要编辑 .csproj 文件, 可右键单击解决方案资源管理器中的项目名称, 然后选择“编辑 ContosoUniversity.csproj”。

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

(编写本教程时, 本示例中的版本号是最新的。)

更改连接字符串

在 appsettings.json 文件中, 将连接字符串中的数据库的名称更改为 ContosoUniversity2 或正在使用的计算机上未使用过的其他名称。

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

此更改将设置项目, 以便初始迁移创建新的数据库。这并不是开始使用迁移的必要操作, 但稍后你便会了解这样做的好处。

注意

除更改数据库名称外，删除数据库同样可行。使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` CLI 命令：

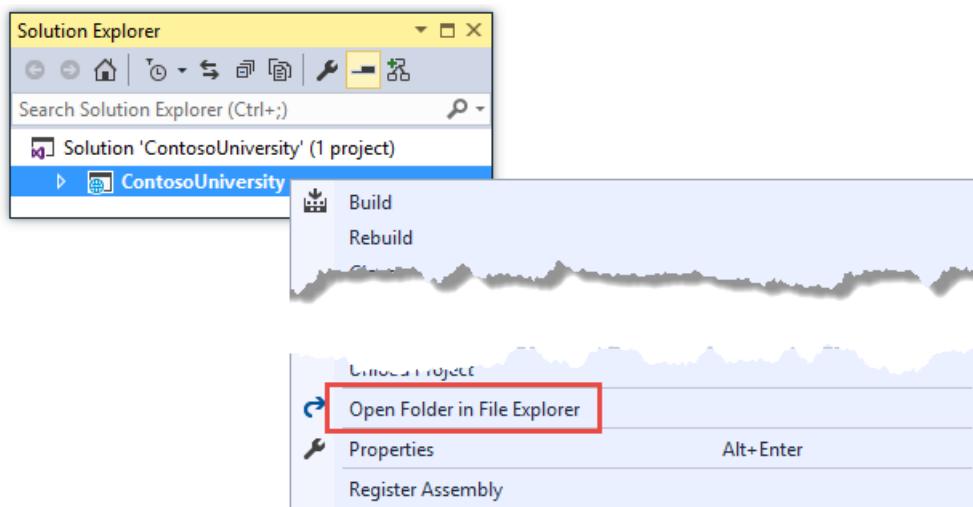
```
dotnet ef database drop
```

下面的部分说明如何运行 CLI 命令。

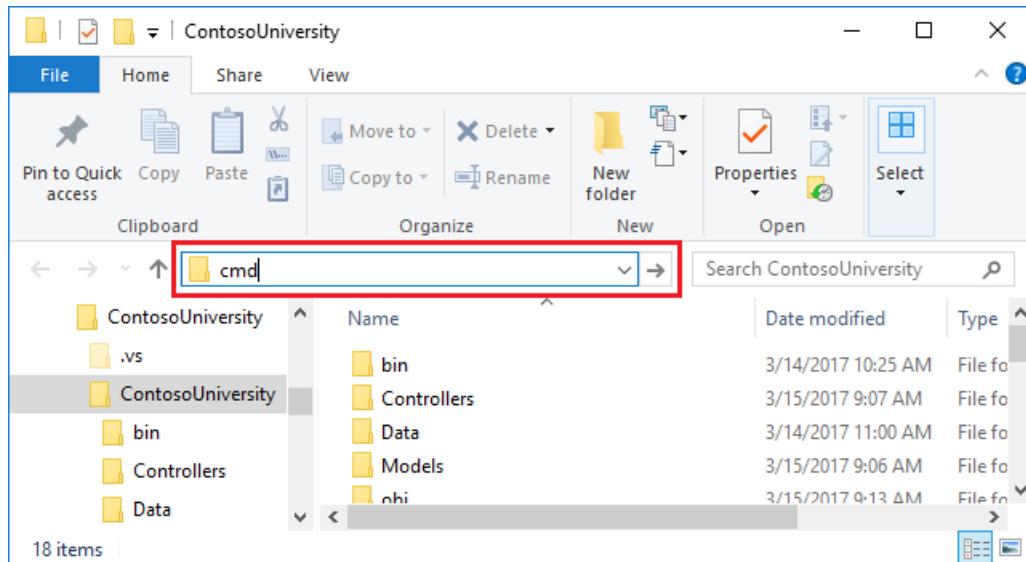
创建初始迁移

保存更改并生成项目。然后打开命令窗口并导航到项目文件夹。下面是执行此操作的快速方法：

- 在解决方案资源管理器中，右键单击项目，然后从上下文菜单中选择“在文件资源管理器中打开”。



- 在地址栏中输入“cmd”，然后按 Enter。



在命令窗口中输入以下命令：

```
dotnet ef migrations add InitialCreate
```

命令窗口中出现如下输出：

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
      repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

注意

如果出现错误消息“找不到任何匹配“dotnet ef”命令的可执行文件”，请参阅[此博客文章](#)获取故障排除帮助。

如果看到错误消息“无法访问文件...ContosoUniversity.dll，因为它正被另一个进程使用。”，请在Windows系统托盘中找到IIS Express图标并右键单击，然后单击“ContosoUniversity”>“停止站点”*。

了解 Up 和 Down 方法

执行 `migrations add` 命令时，EF 已生成将用于从头创建数据库的代码。此代码位于“Migrations”文件夹中名为 `<timestamp>_InitialCreate.cs` 的文件中。`InitialCreate` 类的 `Up` 的方法将创建与数据模型实体集相对应的数据库表，`Down` 方法将删除这些表，如下面的示例所示。

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });
    }

    // Additional code not shown
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Enrollment");
    // Additional code not shown
}
```

迁移调用 `Up` 方法为迁移实现数据模型更改。输入用于回退更新的命令时，迁移调用 `Down` 方法。

此代码适用于输入 `migrations add InitialCreate` 命令时所创建的初始迁移。迁移名称参数（本示例中为“InitialCreate”）用于指定文件名，并且你可以按需使用任何名称。最好选择能概括迁移中所执行操作的字词或短语。例如，可将后面的迁移命名为“AddDepartmentTable”。

如果创建初始迁移时已存在数据库，则会生成数据库创建代码，但此代码不必运行，因为数据库已与数据库模型相匹配。将应用部署到其中尚不存在数据库的其他环境时，此代码将运行以创建数据库，因此最好提前进行测试。这也是提前更改连接字符串中数据库的名称的原因，这样迁移才能从头创建新数据库。

数据模型快照

迁移在 Migrations/SchoolContextModelSnapshot.cs 中创建当前数据库架构的快照。添加迁移时，EF 会通过将数据模型与快照文件进行对比来确定已更改的内容。

删除迁移时，请使用 `dotnet ef migrations remove` 命令。`dotnet ef migrations remove` 删除迁移，并确保正确重置快照。

有关如何使用快照文件的详细信息，请参阅[团队环境中的 EF Core 迁移](#)。

将迁移应用到数据库

在命令窗口中，输入以下命令以创建数据库并在其中创建表。

```
dotnet ef database update
```

该命令的输出与 `migrations add` 命令的输出相似，但其中还包含设置该数据库的 SQL 命令的日志。下面的示例输出中省略了大部分日志。如果希望日志消息中不使用此详细级别，则可更改 appsettings.Development.json 文件中的日志级别。有关详细信息，请参阅[日志记录介绍](#)。

```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\username\AppData\Local\ASP.NET\DataProtection-Keys' as key
      repository and Windows DPAPI to encrypt keys at rest.
info: Microsoft.EntityFrameworkCore.Infrastructure[100403]
      Entity Framework Core 2.0.0-rtm-26452 initialized 'SchoolContext' using provider
      'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (467ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (20ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );
<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[200101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20170816151242_InitialCreate', N'2.0.0-rtm-26452');
Done.
```

使用 SQL Server 对象资源管理器检查数据库（与第一个教程中的做法相同）。你会发现添加了 `__EFMigrationsHistory` 表，该表可用于跟踪已应用到数据库的迁移。查看该表中的数据，其中显示对应初始迁移的一行数据。（上面的 CLI 输出示例中最后部分的日志显示了创建此行的 `INSERT` 语句。）

运行应用程序以验证所有内容照旧运行。

Last Name	First Name	Enrollment Date	
Alexander	Carson	9/1/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete

命令行接口 (CLI) 与包管理器控制台 (PMC)

可通过 .NET Core CLI 命令或 Visual Studio 包管理器控制台 (PMC) 窗口中的 PowerShell cmdlet 使用可管理迁移的 EF 工具。本教程演示如何使用 CLI，但也可以根据喜好使用 PMC。

适用于 PMC 命令的 EF 命令位于 [Microsoft.EntityFrameworkCore.Tools](#) 程序包中。此程序包已包含在 [Microsoft.AspNetCore.All](#) 元包中，因此无需另外安装。

重要说明：此程序包与通过编辑 .csproj 文件为 CLI 安装的程序包不同。此程序包的名称以 `Tools` 结尾，而 CLI 程序包的名称以 `Tools.DotNet` 结尾。

有关 CLI 命令的详细信息，请参阅 [.NET Core CLI](#)。

有关 PMC 命令的详细信息，请参阅[包管理器控制台 \(Visual Studio\)](#)。

总结

本教程已介绍如何创建并应用初始迁移。下一教程将介绍有关展开数据模型的更高级主题。同时还将介绍创建并应用其他迁移的方法。

ASP.NET Core MVC 和 EF Core - 数据模型 - 第 5 个教程(共 10 个)

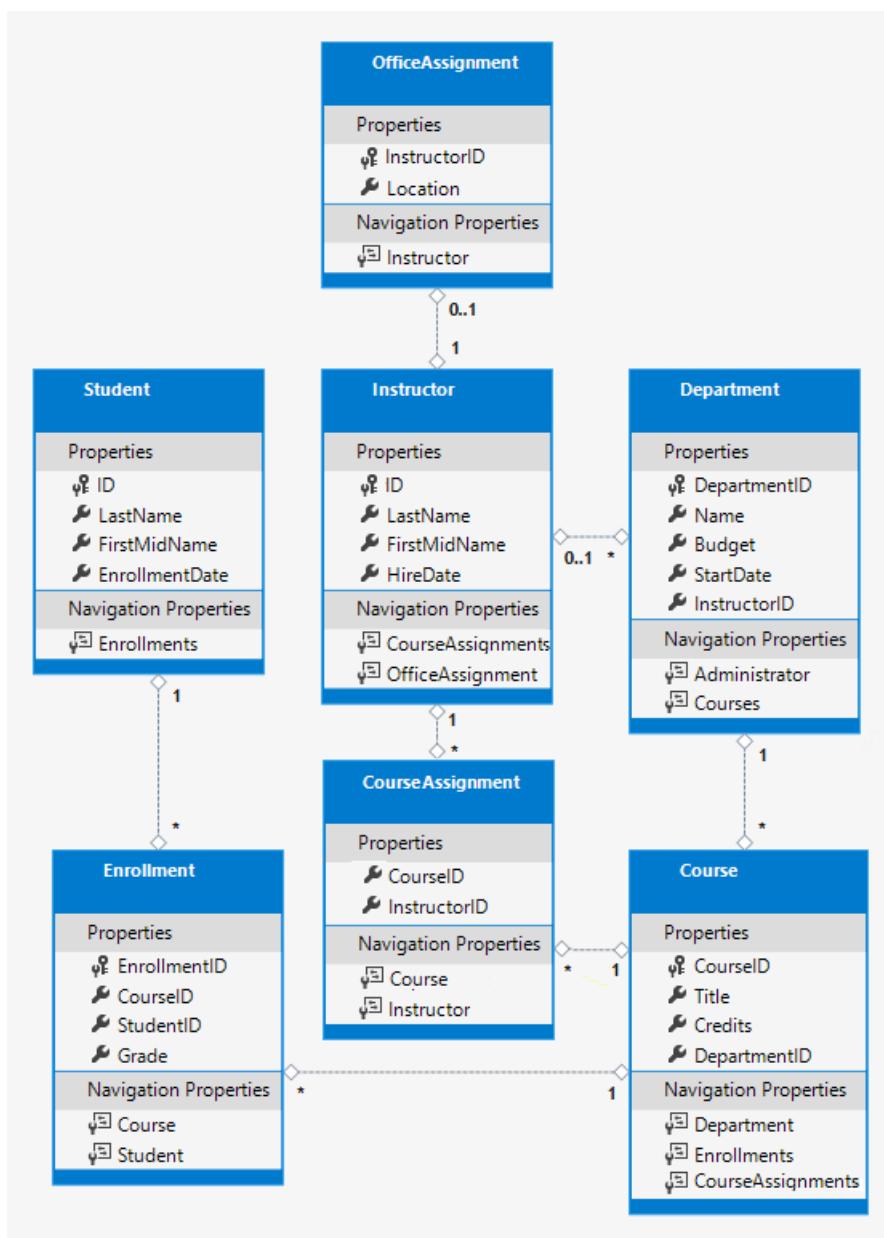
2018/5/17 • 34 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

之前的教程介绍了由三个实体组成的简单数据模型。本教程将添加更多实体和关系, 并通过指定格式化、验证和数据库映射规则来自定义数据模型。

完成本教程后, 实体类将构成下图所示的完整数据模型:



使用特性自定义数据模型

本节介绍如何使用指定格式化、验证和数据库映射规则的特性来自定义数据模型。随后在接下来的几节中创建完

整的学校数据模型，创建方法：向已创建的类添加特性，并为模型中剩余的实体类型创建新类。

DataType 特性

对于学生注册日期，目前所有网页都显示有时间和日期，尽管对此字段而言重要的只是日期。使用数据注释特性，可更改一次代码，修复每个视图中数据的显示格式。若要查看如何执行此操作，请向 `Student` 类的 `EnrollmentDate` 属性添加一个特性。

在 `Models/Student.cs` 中，为 `System.ComponentModel.DataAnnotations` 命名空间添加一个 `using` 语句，将 `DataType` 和 `DisplayFormat` 特性添加到 `EnrollmentDate` 属性，如以下示例所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`DataType` 属性用于指定比数据库内部类型更具体的数据类型。在此示例中，我们只想跟踪日期，而不是日期和时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-` (读作 data dash) 特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据默认格式显示，后者以服务器的 `CultureInfo` 为基础。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。(你可能不想为某些字段执行此操作—例如，对于货币值，你可能不希望文本框中的货币符号可编辑。)

可单独使用 `DisplayFormat` 特性，但通常建议同时使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达数据在屏幕上的呈现方式，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能(例如，显示日历控件、区域设置适用的货币符号、电子邮件链接、某种客户端输入验证等)。
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。

有关详细信息，请参阅 [<input> 标记帮助器文档](#)。

运行应用，进入“学生”索引页，注意注册日期中不再显示时间。任何使用“学生”模型的视图都是如此。

Last Name	First Name	Enrollment Date	
Alexander	Carson	2005-09-01	Edit Details Delete
Alonso	Meredith	2002-09-01	Edit Details Delete
Anand	Arturo	2003-09-01	Edit Details Delete

StringLength 特性

还可使用特性指定数据验证规则和验证错误消息。`StringLength` 特性设置数据库中的最大长度，并为 ASP.NET MVC 提供客户端和服务器端验证。还可在此属性中指定最小字符串长度，但最小值对数据库架构没有影响。

假设要确保用户输入的名称不超过 50 个字符。若要添加此限制，请将 `StringLength` 特性添加到 `LastName` 和 `FirstMidName` 属性，如下例所示：

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`StringLength` 特性不会阻止用户在名称中输入空格。可使用 `RegularExpression` 特性应用输入限制。例如，以下代码要求第一个字符为大写，其余字符按字母顺序排列：

```
[RegularExpression(@"^([A-Z]+[a-zA-Z]*\s-]*)$")]
```

`MaxLength` 特性的作用与 `StringLength` 特性类似，但不提供客户端验证。

现在数据库模型发生了变化，需要更改数据库架构。请使用迁移来更新架构，迁移时不会丢失通过应用程序 UI 添加到数据库的任何数据。

保存更改并生成项目。随后打开项目文件夹中的命令窗口并输入以下命令：

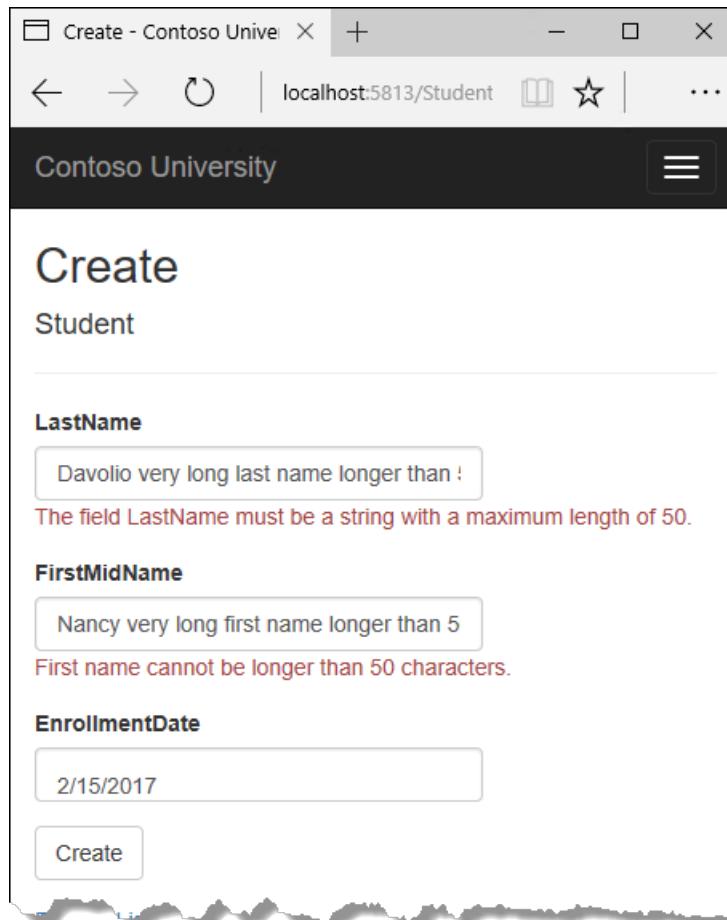
```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

`migrations add` 命令发出警告：可能出现数据丢失，因为更改后，有两列的最大长度缩短。迁移时会创建一个名为 `<timeStamp>_MaxLengthOnNames.cs` 的文件。此文件包含 `Up` 方法中的代码，该代码将更新数据库以匹配当前数据模型。`database update` 命令运行该代码。

Entity Framework 使用迁移文件名的前缀时间戳发出迁移命令。可在运行 `update-database` 命令前创建多个迁移，然后按照创建顺序应用所有迁移。

运行该应用，选择“学生”选项卡，单击“新建”，然后输入名称（超过 50 个字符）。单击“创建”时，客户端验证会显示一条错误消息。



Column 特性

还可使用特性来控制类和属性映射到数据库的方式。假设在名字字段使用了 `FirstMidName`，这是因为该字段也可能包含中间名。但却希望将数据库列命名为 `FirstName`，因为要针对数据库编写即席查询的用户习惯使用该姓名。若要进行此映射，可使用 `Column` 特性。

`Column` 特性指定，创建数据库时，映射到 `FirstMidName` 属性的 `Student` 表的列将被命名为 `FirstName`。换言之，在代码引用 `Student.FirstMidName` 时，数据来源将是 `Student` 表的 `FirstName` 列或在其中进行更新。如果不指定列名称，则其名称与属性名称相同。

在 `Student.cs` 文件中，为 `System.ComponentModel.DataAnnotations.Schema` 添加一个 `using` 语句，并将列名称特性

添加到 `FirstMidName` 属性，如以下突出显示的代码所示：

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]  
[Column("FirstName")]
        public string FirstName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

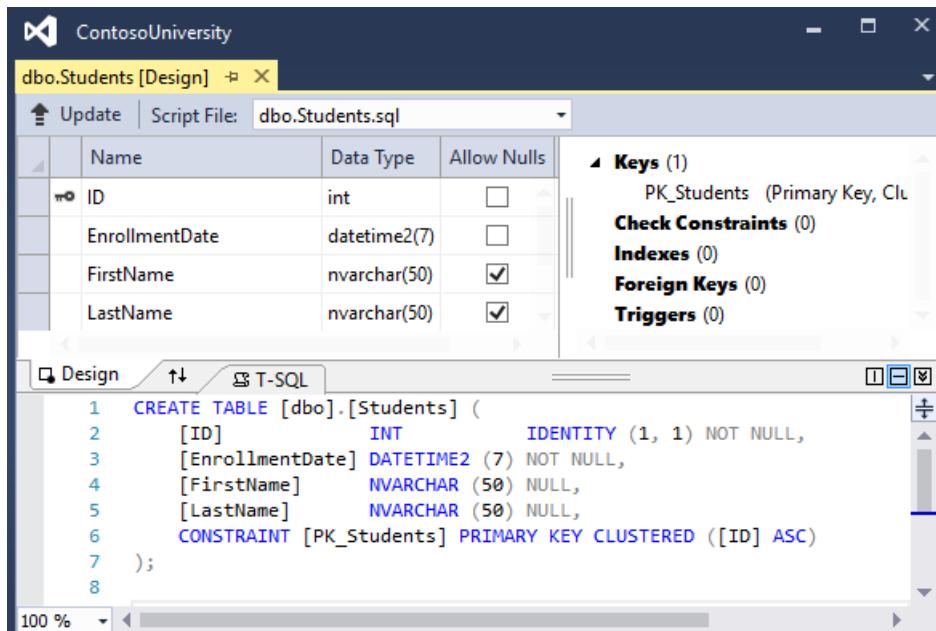
添加 `Column` 特性后，`SchoolContext` 的支持模型会发生改变，与数据库不再匹配。

保存更改并生成项目。随后打开项目文件夹中的命令窗口，输入以下命令，创建另一个迁移：

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

在 SQL Server 对象资源管理器中，双击 Student 表，打开 Student 表设计器。

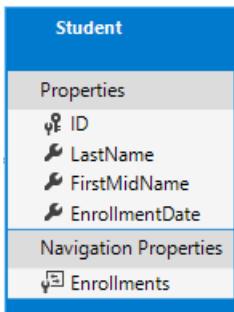


在进行前两次迁移前，名称列的类型为 `nvarchar (MAX)`。而现在则是 `nvarchar (50)`，列名从 `FirstMidName` 变为 `FirstName`。

注意

如果尚未按以下各节所述创建所有实体类就尝试进行编译，则可能会出现编译器错误。

Student 实体的最终更改



在 Models/Student.cs 中，将之前添加的代码替换为以下代码。突出显示所作更改。

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Required 特性

Required 特性使名称属性成为必填字段。值类型(DateTime、int、double、float 等)等不可为 null 的类型不需要

Required 特性。系统会将不可为 null 的类型自动视为必填字段。

可删除 **Required** 特性，并用 **StringLength** 特性的最小长度参数来替换：

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

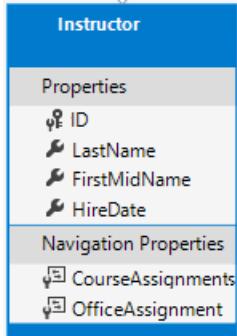
Display 特性

`Display` 特性指定文本框的标题应是“名”、“姓”、“全名”和“注册日期”，而不是每个实例中的属性名称（其中没有分隔单词的空格）。

FullName 计算属性

`FullName` 是计算属性，可返回通过串联两个其他属性创建的值。因此它只有一个 `get` 访问器，且数据库中不会生成 `FullName` 列。

创建 Instructor 实体



创建 Models/Instructor.cs，使用以下代码替换模板代码：

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

请注意，在 Student 和 Instructor 实体中有几个属性是相同的。本系列后面的[实现继承教程](#)将重构此代码以消除冗余。

可在一行中放置多个特性，因此也可以按如下所示编写 `HireDate` 特性：

```
[DataType(DataType.Date), Display(Name = "Hire Date"), DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

CourseAssignments 和 OfficeAssignment 导航属性

`CourseAssignments` 和 `OfficeAssignment` 是导航属性。

一名讲师可以教授任意数量的课程，因此 `CourseAssignments` 定义为集合。

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

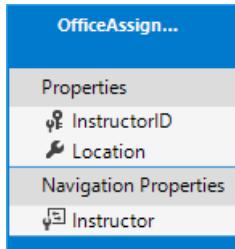
如果导航属性可包含多个实体，则其类型必须是可添加、可删除和可更新实体的列表。可指定 `ICollection<T>` 或诸如 `List<T>` 或 `HashSet<T>` 的类型。如果指定 `ICollection<T>`，EF 默认会创建一个 `HashSet<T>` 集合。

下面在关于多对多关系的部分中解释了这些作为 `CourseAssignment` 实体的原因。

Contoso University 业务规则规定，讲师最多只能有一个办公室，因此 `OfficeAssignment` 属性拥有一个 `OfficeAssignment` 实体（如果未分配办公室，则该实体可能为 null）。

```
public OfficeAssignment OfficeAssignment { get; set; }
```

创建 OfficeAssignment 实体



用以下代码创建 Models/OfficeAssignment.cs:

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

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

Key 特性

讲师与 OfficeAssignment 实体间存在一对零或一的关系。办公室分配仅与分配有办公室的讲师相关，因此其主键也是 Instructor 实体的外键。但 Entity Framework 无法自动识别 InstructorID 作为此实体的主键，因为其名称不符合 ID 或 classnameID 命名约定。因此，`Key` 特性用于将其识别为主键：

```
[Key]
public int InstructorID { get; set; }
```

如果实体具有自己的主键，但你希望使用 classnameID 或 ID 以外的其他属性名称，则也可使用 `Key` 特性。

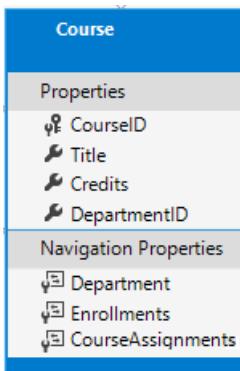
默认情况下，EF 将键视为非数据库生成，因为该列面向的是识别关系。

Instructor 导航属性

Instructor 实体具有可为 null `OfficeAssignment` 导航属性（因为可能未向讲师分配办公室），而 OfficeAssignment 实体具有不可为 null `Instructor` 导航属性（因为在没有讲师的情况下不会分配办公室 - `InstructorID` 不可为 null）。Instructor 实体具有相关 OfficeAssignment 实体时，每个实体都将在其导航属性中引用另一实体。

可向 Instructor 导航属性添加 `[Required]` 特性，指定必须有相关讲师，但这不是必须的，因为 `InstructorID` 外键（也是此表的键）不可为 null。

修改 Course 实体



在 Models/Course.cs 中，将之前添加的代码替换为以下代码。突出显示所作更改。

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

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

Course 实体具有外键属性 `DepartmentID`，该属性指向相关 `Department` 实体，同时它还具有 `Department` 导航属性。

如果拥有相关实体的导航属性，则 Entity Framework 不会要求为数据模型添加外键属性。只要有需要，EF 就会自动在数据库中创建外键，并为其创建 [阴影属性](#)。但如果数据模型包含外键，则更新会变得更简单、更高效。例如，提取 Course 实体进行编辑时，如果未加载该实体，那么 `Department` 实体为 null，因此，更新 Course 实体时，必须先提取 `Department` 实体。数据模型中包含外键属性 `DepartmentID` 时，更新前无需提取 `Department` 实体。

DatabaseGenerated 特性

`CourseID` 属性上具有 `None` 参数的 `DatabaseGenerated` 特性指定主键值由用户提供，而不是由数据库生成。

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

默认情况下，Entity Framework 假定主键值由数据库生成。大多数情况下，这是理想情况。但对 Course 实体而言，需使用用户指定的课程编号，例如一个系为 1000 系列，另一个系为 2000 系列等。

`DatabaseGenerated` 特性也可用于生成默认值，如在用于记录行创建或更新日期的数据库列中。有关详细信息，请参阅[生成的属性](#)。

外键和导航属性

Course 实体中的外键属性和导航属性可反映以下关系：

向一个系分配课程后，出于上述原因，会出现 `DepartmentID` 外键和 `Department` 导航属性。

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

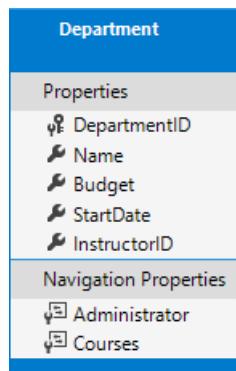
参与一门课程的学生数量不定，因此 `Enrollments` 导航属性是一个集合：

```
public ICollection<Enrollment> Enrollments { get; set; }
```

一门课程可能有多位授课讲师，因此 `CourseAssignments` 导航属性是一个集合（稍后会解释 `CourseAssignment` 类型）：

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

创建 Department 实体



用以下代码创建 Models/Department.cs：

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

Column 特性

`Column` 特性之前用于更改列名称映射。在 `Department` 实体的代码中, `Column` 特性用于更改 SQL 数据类型映射, 以便使用数据库中的 SQL Server 货币类型来定义该列:

```

[Column(TypeName="money")]
public decimal Budget { get; set; }

```

通常不需要列映射, 因为 Entity Framework 会根据你为属性定义的 CLR 类型选择适当的 SQL Server 数据类型。CLR `decimal` 类型会映射到 SQL Server `decimal` 类型。但假如你知道该列要表示货币金额, 那么货币数据类型会更加合适。

外键和导航属性

外键和导航属性可反映以下关系:

一个系可能有也可能没有管理员, 而管理员始终是讲师。因此 `InstructorID` 属性作为 `Instructor` 实体内外键, 且 `int` 类型指定后跟有一个问号, 将该属性标记为可为 null。导航属性名为 `Administrator`, 但其中包含 `Instructor` 实体:

```

public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }

```

一个系可以有多门课程, 因此存在 `Courses` 导航属性:

```

public ICollection<Course> Courses { get; set; }

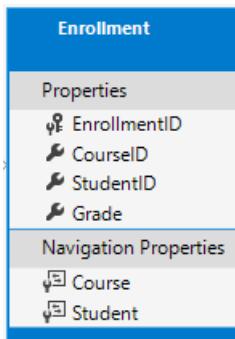
```

注意

按照约定, Entity Framework 能针对不可为 null 的外键和多对多关系启用级联删除。这可能导致循环级联删除规则, 尝试添加迁移时该规则会造成异常。例如, 如果未将 Department.InstructorID 属性定义为可为 null, 那么在删除系时, EF 会配置级联删除规则来删除讲师, 这是预期外的情况。如果业务规则要求 InstructorID 属性不可为 null, 则必须使用以下 Fluent API 语句禁用关系中的级联删除:

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

修改 Enrollment 实体



在 Models/Enrollment.cs 中, 将之前添加的代码替换为以下代码:

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

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

外键和导航属性

外键属性和导航属性可反映以下关系:

注册记录面向一门课程, 因此存在 CourseID 外键属性和 Course 导航属性:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

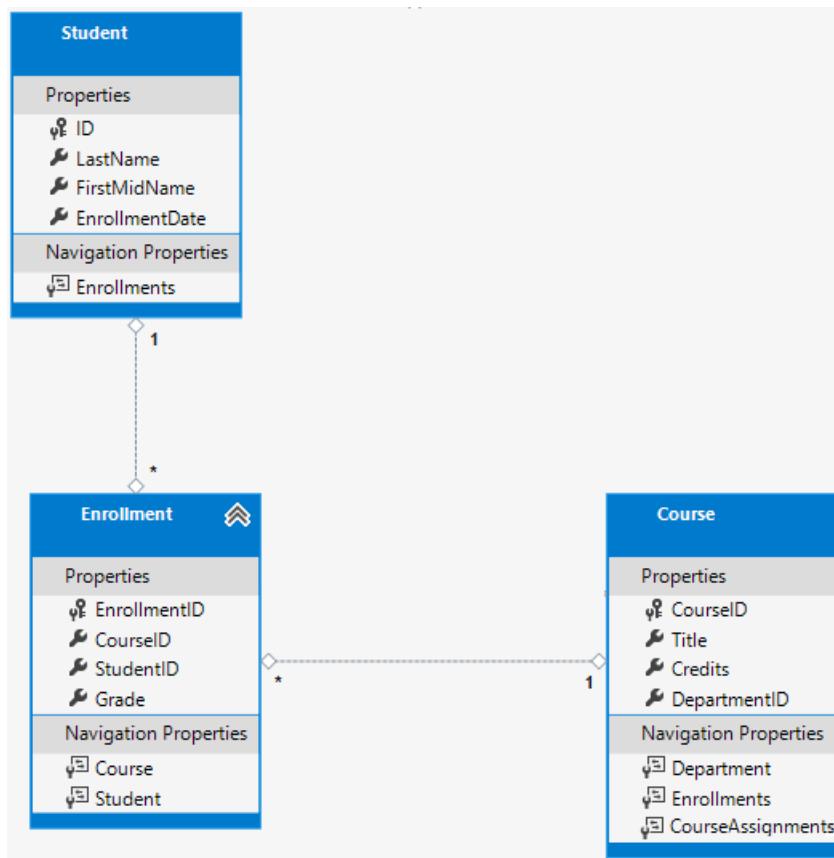
注册记录面向一名学生，因此存在 `StudentID` 外键属性和 `Student` 导航属性：

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

多对多关系

Student 和 Course 实体间存在多对多关系，Enrollment 实体在数据库中充当带有效负载的多对多联接表。“带有有效负载”是指 Enrollment 表包含除联接表外键之外的其他数据（在此示例中为主键和 Grade 属性）。

下图显示这些关系在实体关系图中的外观。（该图是使用 EF 6.x 的 Entity Framework Power Tools 生成的，教程未介绍如何创建该图，此处仅作为示例使用。）

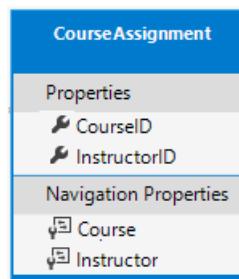


每条关系线的一端显示 1，另一端显示星号（*），这表示一对多关系。

如果 Enrollment 表不包含年级信息，则它只需包含两个外键（CourseID 和 StudentID）。在这种情况下，该表是数据库中不带有效负载的多对多联接表（或纯联接表）。Instructor 和 Course 实体具有这种多对多关系，下一步是创建实体类，将其作为不带有效负载的联接表。

（EF 6.x 支持多对多关系的隐式联接表，但 EF Core 不支持。有关详细信息，请参阅 [EF Core GitHub 存储库中的讨论](#)。）

CourseAssignment 实体



用以下代码创建 Models/CourseAssignment.cs：

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

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

联接实体名称

数据库中的 Instructor 与 Course 多对多关系需要联接表，并且必须由实体集表示。将联接实体命名为 EntityName1EntityName2 很常见，在此示例中实体名为 CourseInstructor。但是，建议选择一个可描述关系的名称。数据模型开始很简单，且会不断增长，随后无有效负载联接会频繁获取有效负载。如果一开始实体名称为描述性名称，那么之后就不必更改名称。理想情况下，联接实体在业务域中可能具有自己的自带名称（可能是单个字）。例如，可通过 Rating 链接 Book 和 Customer。对于此关系，相比 CourseInstructor，CourseAssignment 是更好的选择。

组合键

由于外键不可为 null，且它们共同唯一标识表的每一行，因此不需要单独的主键。InstructorID 和 CourseID 属性应充当组合主键。标识 EF 组合主键的唯一方法是使用 Fluent API（无法借助特性来完成）。下一节将介绍如何配置组合主键。

在一个课程可以有多个行，一个讲师可以有多个行的情况下，组合键可确保同一讲师和课程不会有多个行。

Enrollment 联接实体定义其自己的主键，因此可能会出现此类重复。若要防止出现此类重复，可在外键字段上添加唯一索引，或使用类似于 CourseAssignment 的主组合键配置 Enrollment。有关详细信息，请参阅[索引](#)。

更新数据库上下文

将以下突出显示的代码添加到 Data/SchoolContext.cs 文件：

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {}

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

此代码添加新实体并配置 CourseAssignment 实体的组合主键。

用 Fluent API 替代特性

`DbContext` 类的 `OnModelCreating` 方法中的代码使用 Fluent API 来配置 EF 行为。API 称为“Fluent”，因为它通常在将一系列方法调用连接成单个语句后才能使用，如 [EF Core 文档](#) 中的此示例所示：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}

```

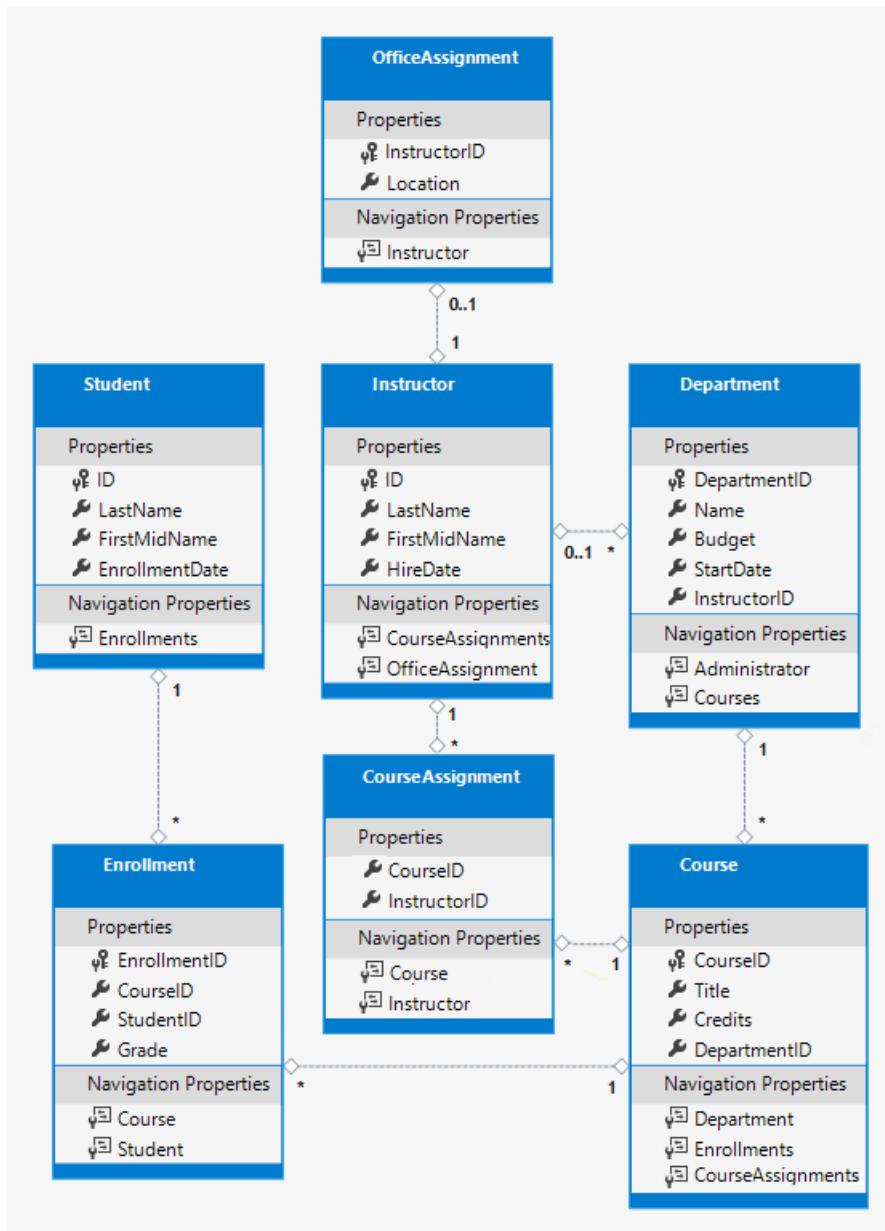
本教程仅将 Fluent API 用于数据库映射，这是无法使用特性实现的。但 Fluent API 还可用于指定大多数格式化、验证和映射规则，这可通过特性完成。`MinimumLength` 等特性不能通过 Fluent API 应用。如前所述，`MinimumLength` 不会更改架构，它只应用客户端和服务器端验证规则。

某些开发者倾向于仅使用 Fluent API 以保持实体类的“纯净”。如有需要，可混合使用特性和 Fluent API，且有些自定义只能通过 Fluent API 实现，但通常建议选择一种方法并尽可能坚持使用这一种。如果确实要使用两种，请注意，只要出现冲突，Fluent API 就会覆盖特性。

有关特性和 Fluent API 的详细信息，请参阅[配置方法](#)。

显示关系的实体关系图

下图显示 Entity Framework Power Tools 针对已完成的学校模型创建的关系图。



除一对多关系线(1到 *)外, 此处还显示了 Instructor 和 OfficeAssignment 实体间的一对零或一关系线(1 到 0..1), 以及 Instructor 和 Department 实体间的零对多或一对多关系线(0..1 到 *)。

使用测试数据设定数据库种子

使用以下代码替换 Data/DbInitializer.cs 文件中的代码, 从而为创建的新实体提供种子数据。

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())

```

```

    }

    return; // DB has been seeded
}

var students = new Student[]
{
    new Student { FirstMidName = "Carson", LastName = "Alexander",
        EnrollmentDate = DateTime.Parse("2010-09-01") },
    new Student { FirstMidName = "Meredith", LastName = "Alonso",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Arturo", LastName = "Anand",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Gytis", LastName = "Barzdukas",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Yan", LastName = "Li",
        EnrollmentDate = DateTime.Parse("2012-09-01") },
    new Student { FirstMidName = "Peggy", LastName = "Justice",
        EnrollmentDate = DateTime.Parse("2011-09-01") },
    new Student { FirstMidName = "Laura", LastName = "Norman",
        EnrollmentDate = DateTime.Parse("2013-09-01") },
    new Student { FirstMidName = "Nino", LastName = "Olivetto",
        EnrollmentDate = DateTime.Parse("2005-09-01") }
};

foreach (Student s in students)
{
    context.Students.Add(s);
}
context.SaveChanges();

var instructors = new Instructor[]
{
    new Instructor { FirstMidName = "Kim", LastName = "Abercrombie",
        HireDate = DateTime.Parse("1995-03-11") },
    new Instructor { FirstMidName = "Fadi", LastName = "Fakhouri",
        HireDate = DateTime.Parse("2002-07-06") },
    new Instructor { FirstMidName = "Roger", LastName = "Harui",
        HireDate = DateTime.Parse("1998-07-01") },
    new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
        HireDate = DateTime.Parse("2001-01-15") },
    new Instructor { FirstMidName = "Roger", LastName = "Zheng",
        HireDate = DateTime.Parse("2004-02-12") }
};

foreach (Instructor i in instructors)
{
    context.Instructors.Add(i);
}
context.SaveChanges();

var departments = new Department[]
{
    new Department { Name = "English", Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Abercrombie").ID },
    new Department { Name = "Mathematics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID },
    new Department { Name = "Engineering", Budget = 350000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID },
    new Department { Name = "Economics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)
{
    context.Departments.Add(d);
}

```

```

        context.Departments.Add(u),
    }
    context.SaveChanges();

    var courses = new Course[]
    {
        new Course {CourseID = 1050, Title = "Chemistry", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature", Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
    };

    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }
    context.SaveChanges();

    var officeAssignments = new OfficeAssignment[]
    {
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
            Location = "Smith 17" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
            Location = "Gowan 27" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
            Location = "Thompson 304" },
    };

    foreach (OfficeAssignment o in officeAssignments)
    {
        context.OfficeAssignments.Add(o);
    }
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
        new CourseAssignment {
    
```

```

        CourseID = courses.Single(c => c.Title == "Macroeconomics").CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus").CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry").CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition").CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature").CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
    },
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
        Grade = Grade.A
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
        Grade = Grade.C
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry").CourseID
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
        Grade = Grade.B
    },
    new Enrollment {

```

```

        StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Li").ID,
        CourseID = courses.Single(c => c.Title == "Composition").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Justice").ID,
        CourseID = courses.Single(c => c.Title == "Literature").CourseID,
        Grade = Grade.B
    }
}

foreach (Enrollment e in enrollments)
{
    var enrollmentInDataBase = context.Enrollments.Where(
        s =>
            s.Student.ID == e.StudentID &&
            s.Course.CourseID == e.CourseID).SingleOrDefault();
    if (enrollmentInDataBase == null)
    {
        context.Enrollments.Add(e);
    }
}
context.SaveChanges();
}
}

```

如第一个教程所述，大部分此类代码仅创建新实体对象，并按测试要求将示例数据加载到属性中。注意多对多关系的处理方法：代码在 `Enrollments` 和 `CourseAssignment` 联接实体集中创建实体，以此来创建关系。

添加迁移

保存更改并生成项目。然后打开项目文件夹中的命令窗口，输入 `migrations add` 命令（先不要执行 `updatedatabase` 命令）：

```
dotnet ef migrations add ComplexDataModel
```

会出现数据可能丢失的警告。

An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'

如果此时尝试运行 `database update` 命令(先不要执行此操作), 则会出现以下错误:

ALTER TABLE 语句与 FOREIGN KEY 约束“FK_dbo.Course_dbo.Department_DepartmentID”冲突。冲突发生位置：数据库“ContosoUniversity”、表“dbo.Department”和列“DepartmentID”。

有时使用现有数据执行迁移时，需将存根数据插入数据库，满足外键约束。`Up` 方法中生成的代码将不可为 null 的 `DepartmentID` 外键添加到 `Course` 表中。如果代码运行时，`Course` 表中已经有了行，则 `AddColumn` 操作失败，因为 SQL Server 不知道要向不可为 null 的列中放入什么值。本教程将在新数据库上运行迁移，但在生产应用程序中，必须使迁移处理现有数据，因此下方通过示例介绍如何执行此操作。

为使此迁移处理现有数据，必须更改代码，赋予新列默认值，并创建一个名为“Temp”的存根系，作为默认系。之后，Course 行将在 `Up` 方法运行后与“Temp”系建立联系。

- 打开 {timestamp}_ComplexDataModel.cs 文件。
- 对将 DepartmentID 列添加到 Course 表的代码行添加注释。

```

migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);

```

- 在创建 Department 表的代码后添加以下突出显示的代码：

```

migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
            SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });
}

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);

```

在生产应用程序中，可编写代码或脚本来添加 Department 行并将 Course 行与新 Department 行相关联。随后将不在需要“Temp”系或 Course.DepartmentID 列中的默认值。

保存更改并生成项目。

更改连接字符串并更新数据库

现在 `DbInitializer` 类中就有了新代码，可将新实体的种子数据添加到空数据库。若要让 EF 创建新的空数据库，请将 `appsettings.json` 中连接字符串内的数据库名称更改为 `ContosoUniversity3` 或正在使用的计算机上未使

用过的其他名称。

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"  
  },
```

将更改保存到 appsettings.json。

注意

除更改数据库名称外，删除数据库同样可行。使用 SQL Server 对象资源管理器 (SSOX) 或 `database drop` CLI 命令：

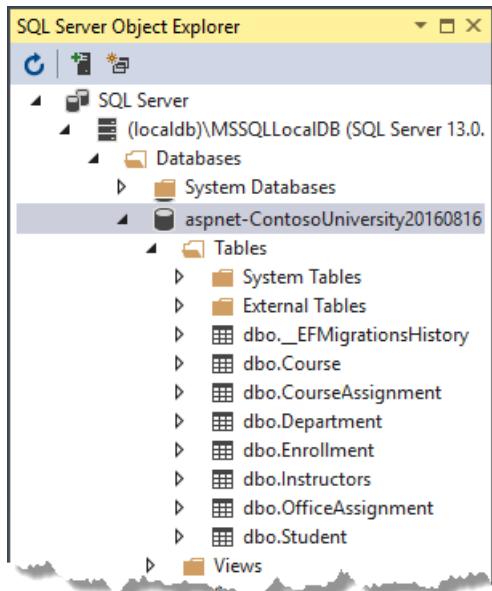
```
dotnet ef database drop
```

更改数据库名称或删除数据库后，在命令窗口运行 `database update` 命令，执行迁移。

```
dotnet ef database update
```

运行应用，使 `DbInitializer.Initialize` 方法运行并填充新数据库。

像之前一样在 SSOX 中打开数据库，然后展开 Tables 节点，查看是否已创建所有表。（如果之前打开的 SSOX 尚未关闭，请单击“刷新”按钮。）



运行应用，触发设定数据库种子的初始化代码。

右键单击“CourseAssignment”表，然后选择“查看数据”，验证其中是否存在数据。

The screenshot shows the SSMS interface with the title bar 'ContosoUniversity'. A window titled 'dbo.CourseAssignment [Data]' is open, displaying a table with two columns: 'CourseID' and 'InstructorID'. The data consists of nine rows:

	CourseID	InstructorID
▶	2021	1
	2042	1
	1045	2
	1050	3
	3141	3
	1050	4
	4022	5
*	4041	5
	NULL	NULL

总结

现在你就得到了更复杂的数据模型和相应的数据库。后面教程将更多详细的介绍如何访问相关数据。

[上一页](#) [下一页](#)

ASP.NET Core MVC 和 EF Core - 读取相关数据 - 第 6 个课程(共 10 个课程)

2018/5/17 • 16 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

前面教程创建了学校数据模型。本教程将读取并显示相关数据 - 即 Entity Framework 加载到导航属性中的数据。

下图是将会用到的页面。

The screenshot shows a web browser window titled "Courses - Contoso Univ". The address bar displays "localhost:5813/Courses". The main content area is titled "Courses" and contains a table with three rows of course data:

Number	Title	Credits	Department	Action
1045	Calculus	4	Mathematics	Edit Details Delete
1050	Chemistry	3	Engineering	Edit Details Delete
2021	Composition	3	English	Edit Details Delete

Last Name	First Name	Hire Date	Office	Courses
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry

Courses Taught by Selected Instructor

Number	Title	Department
Select	2021	Composition English
Select	2042	Literature English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

相关数据的预先加载、显式加载和延迟加载

对象关系映射 (ORM) 软件 (如 Entity Framework) 可通过多种方式将相关数据加载到实体的导航属性中：

- 预先加载。读取该实体时，会同时检索相关数据。此时通常会出现单一联接查询，检索所有必需数据。可使用 `Include` 和 `ThenInclude` 方法指定 Entity Framework Core 中的预先加载。

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

可在单独查询中检索一些数据，EF 会“修正”导航属性。也就是说，EF 会自动添加单独检索的实体，将其添加到之前检索的实体的导航属性中所属的位置。对于检索相关数据的查询，可使用 `Load` 方法，而不采用返回列表或对象的方法，如 `ToList` 或 `Single`。

```

var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

Query: all Department rows

Query: Course rows related to Department d

- 显式加载。首次读取实体时，不检索相关数据。如有需要，可编写检索相关数据的代码。就像使用单独查询进行预先加载一样，显式加载时会向数据库发送多个查询。二者的区别在于，代码通过显式加载指定要加载的导航属性。在 Entity Framework Core 1.1 中，可使用 `Load` 方法执行显式加载。例如：

```

var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}

```

Query: all Department rows

Query: Course rows related to Department d

- 延迟加载。首次读取实体时，不检索相关数据。然而，首次尝试访问导航属性时，会自动检索导航属性所需的数据。每次首次尝试从导航属性获取数据时，都向数据库发送查询。Entity Framework Core 1.0 不支持延迟加载。

性能注意事项

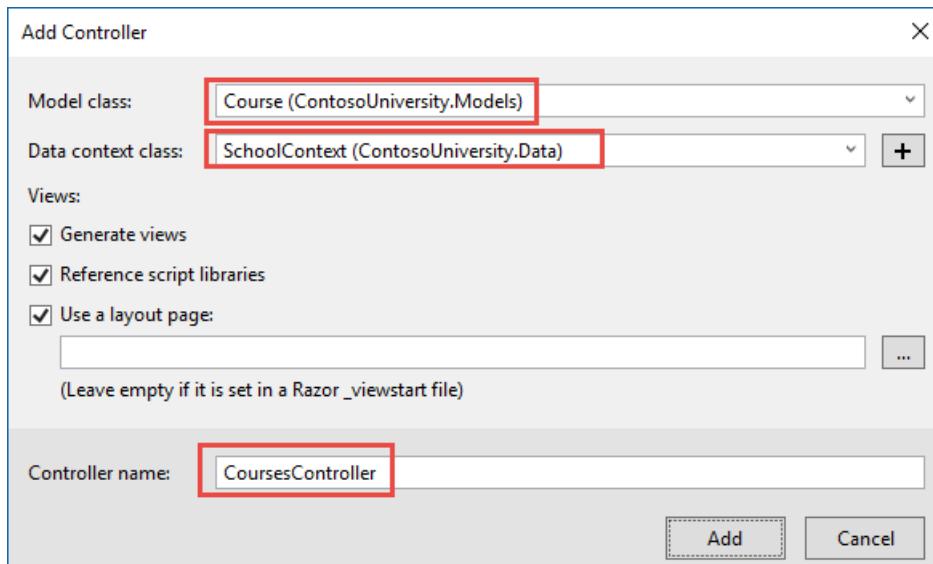
如果知道自己需要每个检索的实体的相关数据，选择预先加载可获得最佳性能，因为相比每个检索的实体的单独查询，发送到数据库的单个查询更加有效。例如，假设每个系有十个相关课程。预先加载所有相关数据时，只会进行单一（联接）查询，往返数据库一次。单独查询每个系的课程时，会往返数据库十一次。延迟较高时，额外往返数据库对性能尤为不利。

另一方面，在某些情况下，单独查询会更加高效。在一个查询中预先加载所有相关数据时，可能会生成一个非常复杂的联接，SQL Server 无法有效处理该联接。或者，如果你正在处理一组实体且只需访问其子集的导航属性，那么采用单独查询可获得更佳性能，因为预先加载所有数据后，会检索不需要的数据。如果看重性能，那么最好测试两种方式的性能，以便做出最佳选择。

创建显示院系名称的“课程”页

Course 实体包括导航属性，其中包含分配有课程的系的 Department 实体。若要在课程列表中显示接受分配的系的名称，需从位于 `Course.Department` 导航属性中的 Department 实体获取 Name 属性。

使用与带视图的 MVC 控制器相同的选项，及之前用于学生控制器的 Entity Framework 基架为 Course 实体类型创建名为 CoursesController 的控制器，如下图所示：



打开 CoursesController.cs 并检查 Index 方法。自动基架使用 Include 方法为 Department 导航属性指定了预先加载。

将 Index 方法替换为以下代码，该代码为返回 Course 实体（是 courses 而不是 schoolContext）的 IQueryable 赋予了更合适的名称：

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

在 Views/Courses/Index.cshtml 中，将模板代码替换为以下代码。突出显示所作更改：

```

@model IEnumerable<ContosoUniversity.Models.Course>

 @{
     ViewData["Title"] = "Courses";
 }

<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

已对基架代码进行了如下更改：

- 将标题从“索引”更改为“课程”。
- 添加了显示 `CourseID` 属性值的“数字”列。默认情况下，不针对主键进行架构，因为对最终用户而言，它们通常没有意义。但在这种情况下主键是有意义的，而你需要将其呈现出来。
- 更改“院系”列，显示院系名称。该代码显示已加载到 `Department` 导航属性中的 `Department` 实体的 `Name` 属性：

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

运行应用并选择“课程”选项卡，查看包含系名称的列表。

The screenshot shows a web browser window titled "Courses - Contoso Univ". The address bar indicates the URL is "localhost:5813/Courses". The main content area is titled "Contoso University" and displays a table of courses:

Number	Title	Credits	Department	
1045	Calculus	4	Mathematics	Edit Details Delete
1050	Chemistry	3	Engineering	Edit Details Delete
2021	Composition	3	English	Edit Details Delete

创建显示“课程”和“注册”的“讲师”页

本节将为 Instructor 实体创建一个控制器和视图，从而显示“讲师”页：

The screenshot shows a web application window titled "Instructors - Contoso UI". The address bar indicates the URL is "localhost:5813/Instructors/Index/1?". The main content area is titled "Instructors" and contains a table of instructor data. Below the table is a decorative wavy footer graphic.

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

Number	Title	Department
Select	2021	Composition English
Select	2042	Literature English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

该页面通过以下方式读取和显示相关数据：

- 讲师列表显示 OfficeAssignment 实体的相关数据。Instructor 与 OfficeAssignment 实体间存在一对零或一的关系。将预先加载 OfficeAssignment 实体。如前所述，需要主表所有检索行的相关数据时，预先加载通常更有效。在这种情况下，你希望显示所有显示的讲师的办公室分配情况。
- 用户选择一名讲师时，显示相关 Course 实体。Instructor 和 Course 实体是多对多关系。预先加载 Course 实体及其相关 Department 实体。在这种情况下，单独查询可能更有效，因为仅需显示所选讲师的课程。但此示例显示的是如何在本身就位于导航属性内的实体中预先加载导航属性。
- 用户选择一门课程时，会显示 Enrollment 实体集的相关数据。Course 和 Enrollment 实体是一对多关系。单独查询 Enrollment 实体及其相关 Student 实体。

创建“讲师索引”视图模型

“讲师”页显示来自三个不同表格的数据。因此将创建包含三个属性的视图模型，每个属性都包含一个表的数据。

在 SchoolViewModels 文件夹中，创建 InstructorIndexData.cs，并使用以下代码替换现有代码：

```

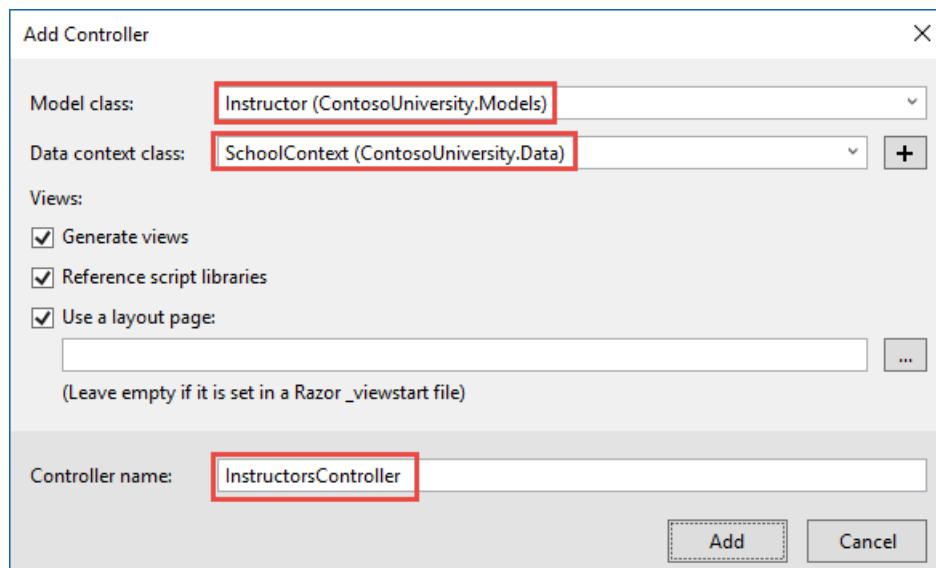
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}

```

创建讲师控制器和视图

使用 EF 读/写操作创建讲师控制器，如下图所示：



打开 InstructorsController.cs 并为 ViewModels 名称空间添加 using 语句：

```
using ContosoUniversity.Models.SchoolViewModels;
```

使用以下代码替换 Index 方法，预先加载相关数据并将其放入视图模型。

```

public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
        .ThenInclude(i => i.Student)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .AsNoTracking()
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}

```

该方法接受可选路由数据 (`id`) 和查询字符串参数 (`courseID`)，二者提供所选讲师和课程的 ID 值。参数由页面上的“选择”超链接提供。

代码先创建一个视图模型实例，并在其中放入讲师列表。代码指定预先加载 `Instructor.OfficeAssignment` 和 `Instructor.CourseAssignments` 导航属性。在 `CourseAssignments` 属性中，加载 `Course` 属性，在其中加载 `Enrollments` 和 `Department` 属性，同时在每个 `Enrollment` 实体中加载 `Student` 属性。

```

viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

```

由于视图始终需要 `OfficeAssignment` 实体，因此更有效的做法是在同一查询中获取。在网页中选择讲师后，需要 `Course` 实体，因此只有在页面频繁显示选中课程时，单个查询才比多个查询更有效。

代码重复 `CourseAssignments` 和 `Course`，因为你需要 `Course` 中的两个属性。`ThenInclude` 调用的第一个字符串获取 `CourseAssignment.Course`、`Course.Enrollments` 和 `Enrollment.Student`。

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
            .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

此时，代码中的另一个 `ThenInclude` 将成为 `Student` 的导航属性，你不需要该属性。但调用 `Include` 是从 `Instructor` 属性重新开始，因此必须再次遍历该链，这次指定 `Course.Department` 而不是 `Course.Enrollments`。

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
            .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

选择讲师时，将执行以下代码。从视图模型中的讲师列表检索所选讲师。然后，该视图模型的 `Courses` 属性加载讲师 `CourseAssignments` 导航属性中的 `Course` 实体。

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

`Where` 方法返回一个集合，但在这种情况下，向该方法传入条件后，只返回一个 `Instructor` 实体。`Single` 方法将集合转换为单个 `Instructor` 实体，让你可以访问该实体的 `CourseAssignments` 属性。`CourseAssignments` 属性包含多个 `CourseAssignment` 实体，而你只需要相关的 `Course` 实体。

如果知道集合只有一个项，则可在集合上使用 `Single` 方法。如果集合为空或包含多个项，`Single` 方法将引发异常。还可使用 `SingleOrDefault`，该方式在集合为空时返回默认值（本例中为 `null`）。但在这种情况下，仍会引发异常（尝试在 `null` 引用上查找 `Courses` 属性时），并且异常消息不会清楚指出异常原因。调用 `Single` 方法时，还可传入 `Where` 条件，而不是分别调用 `Where` 方法：

```
.Single(i => i.ID == id.Value)
```

而不是：

```
.Where(i => i.ID == id.Value).Single()
```

接着，如果选择了课程，则从视图模型中的课程列表中检索所选课程。然后，视图模型的 `Enrollments` 属性加载

该课程 `Enrollments` 导航属性中的 Enrollment 实体。

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

修改讲师索引视图

在 `Views/Instructors/Index.cshtml` 中，将模板代码替换为以下代码。突出显示所作更改。

```

@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

 @{
     ViewData["Title"] = "Instructors";
 }

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>


| Last Name | First Name | Hire Date | Office | Courses |  |
|-----------|------------|-----------|--------|---------|--|
|-----------|------------|-----------|--------|---------|--|


```

已对现有代码进行了如下更改：

- 将模型类更改为 `InstructorIndexData`。
- 将页标题从“索引”更改为“讲师”。
- 添加了仅在 `item.OfficeAssignment` 不为 null 时才显示 `item.OfficeAssignment.Location` 的“办公室”列。
(由于这是一对零或一的关系，因此可能没有相关的 OfficeAssignment 实体。)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- 添加了显示每位讲师所授课程的“课程”列。有关此 Razor 语法的详细信息，请参阅[使用 `@:` 进行显式行转换](#)。
- 添加了向所选讲师的 `tr` 元素中动态添加 `class="success"` 的代码。此时会使用 Bootstrap 类为所选行设置背景色。

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- 紧贴每行其他链接的前端添加了标有 Select 的新超链接，从而使所选讲师 ID 发送到 `Index` 方法。

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

运行应用并选择“讲师”选项卡。没有相关 OfficeAssignment 实体时，该页面显示相关 OfficeAssignment 实体的 Location 属性和空表格单元格。

Last Name	First Name	Hire Date	Office	Courses
Abercrombie	Kim	1995-03-11	Smith 17	2021 Composition 2042 Literature Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus Select Edit Details Delete

在 `Views/Instructors/Index.cshtml` 文件中，关闭表格元素(在文件末尾)后，添加以下代码。选择讲师时，此代码显示与讲师相关的课程列表。

```

@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }
    </table>
}

```

此代码读取视图模型的 `courses` 属性以显示课程列表。它还提供 `Select` 超链接，该链接可将所选课程的 ID 发送到 `Index` 操作方法。

刷新页面并选择讲师。此时会出现一个网格，其中显示有分配给所选讲师的课程，且还显示有每个课程的分配系的名称。

Last Name	First Name	Hire Date	Office	Courses
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus

Courses Taught by Selected Instructor

Number	Title	Department
Select 2021	Composition	English
Select 2042	Literature	English

在刚刚添加的代码块后，添加以下代码。选择课程后，代码将显示参与课程的学生列表。

```

@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}

```

此代码读取视图模型的 Enrollment 属性，从而显示参与课程的学生列表。

再次刷新该页并选择讲师。然后选择一门课程，查看参与的学生列表及其成绩。

Instructors					
Create New					
Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

显式加载

在 InstructorsController.cs 中检索讲师列表时，指定了预先加载 CourseAssignments 导航属性。

假设你希望用户在选中讲师和课程时尽量少查看注册情况。此时建议只在有请求时加载注册数据。若要查看如何执行显式加载的示例，请使用以下代码替换 `Index` 方法，这将删除预先加载 `Enrollment` 并显式加载该属性。代码所作更改为突出显示状态。

```

public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}

```

新代码将从检索 Instructor 实体的代码中删除注册数据的 ThenInclude 方法调用。如果选择了讲师和课程，突出显示的代码会检索所选课程的 Enrollment 实体，及每个 Enrollment 的 Student 实体。

运行应用，立即转到“讲师”索引页，尽管已经更改了数据的检索方式，但该页上显示的内容没有任何不同。

总结

现在你已经使用了预先加载和一个查询及多个查询来读取导航属性中的相关数据。下一个教程将介绍如何更新相关数据。

ASP.NET Core MVC 和 EF Core - 更新相关数据 - 第 7 个教程(共 10 个)

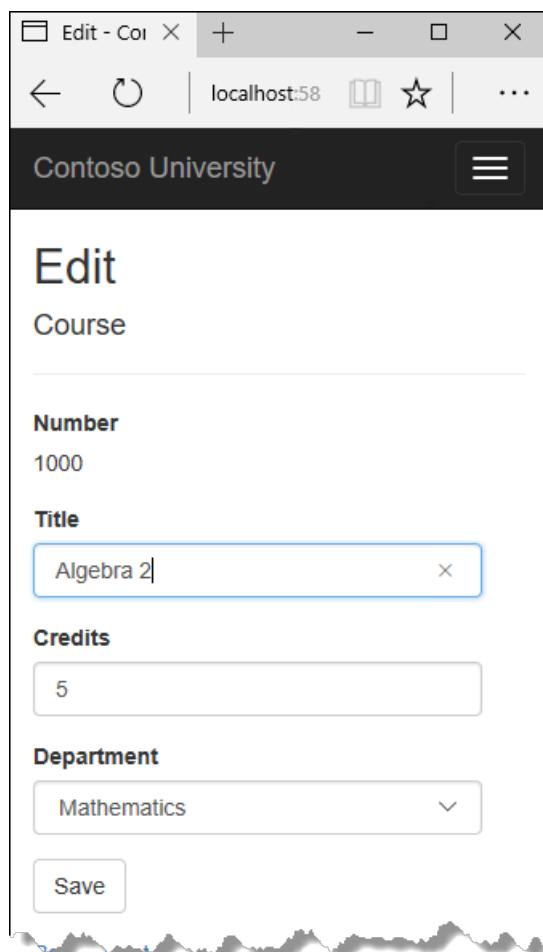
2018/5/17 • 20 min to read • [Edit Online](#)

作者:Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

上一个教程显示出了相关数据, 本教程将通过更新外键字段和导航属性来更新相关数据。

下图是一些将会用到的页面。



Last Name
Abercrombie

First Name
Kim

Hire Date
3/11/1995

Office Location
44/3P

1000 Algebra 2 1045 Calculus 1050 Chemistry
 2021 Composition 2042 Literature 3141 Trigonometry
 4022 Microeconomics 4041 Macroeconomics

Save

自定义课程的创建和编辑页面

创建新的课程实体时，新实体必须与现有院系有关系。为此，基架代码需包括控制器方法、创建视图和编辑视图，且视图中应包括用于选择院系的下拉列表。下拉列表设置了 `Course.DepartmentID` 外键属性，而这正是 Entity Framework 使用适当的 `Department` 实体加载 `Department` 导航属性所需要的。将用到基架代码，但需对其进行稍作更改，以便添加错误处理和对下拉列表进行排序。

在 `CoursesController.cs` 中，删除四种 Create 和 Edit 方法，并将其替换为以下代码：

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

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

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .SingleOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}

```

在 `Edit` `HttpPost` 方法之后，新建一个方法来为下拉列表加载院系信息。

```

private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name",
    selectedDepartment);
}

```

`PopulateDepartmentsDropDownList` 方法获取按名称排序的所有院系的列表，为下拉列表创建 `SelectList` 集合，并将该集合传递给 `ViewBag` 中的视图。该方法可以使用可选的 `selectedDepartment` 参数，而调用的代码可以通过该参数来指定呈现下拉列表时被选择的项。视图将 `DepartmentID` 名称传递给 `<select>` 标记帮助器，该帮助器就知道在 `ViewBag` 对象中查找名为 `DepartmentID` 的 `SelectList`。

`HttpGet` `Create` 方法调用 `PopulateDepartmentsDropDownList` 方法，但不会设置选定项，因为对于新课程而言，其院系尚未建立：

```

public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}

```

`HttpGet` `Edit` 方法根据正在编辑的课程已分配到的院系 ID 设置选定项：

```

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

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

```

`Create` 和 `Edit` 这二者的 `HttpPost` 方法还包括一段代码，用于在错误后重新显示页面时设置选定项。这样可以确保当页面重新显示出现错误消息时，选择的任何院系都将保持选中状态。

将 `.AsNoTracking` 添加到 `Details` 和 `Delete` 方法

为优化“课程详细信息”和“删除”页面的性能，请在 `Details` 和 `HttpGet Delete` 方法中添加 `AsNoTracking` 调用。

```

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

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

```

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

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

修改课程视图

在 Views/Courses/Create.cshtml 中, 向“院系”下拉列表添加一个“选择院系”选项, 将标题从 DepartmentID 更改为 Department, 并添加一条验证消息。

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
```

在 Views/Courses/Edit.cshtml 中, 对“院系”字段进行与 Create.cshtml 中相同的更改。

另外, 在 Views/Courses/Edit.cshtml 中, 在“标题”字段之前添加一个课程编号字段。课程编号是主键, 因此只会显示, 无法更改。

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

“编辑”视图中已有一个隐藏的课程编号字段(`<input type="hidden">`)。添加 `<label>` 标记帮助器后仍然需要该隐藏字段, 因为添加标记帮助器后, 用户在“编辑”页面上单击“保存”时, 已发布数据中并不会包含课程编号。

在 Views/Courses/Delete.cshtml 中, 在顶部添加一个课程编号字段, 并将院系 ID 更改为院系名称。

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

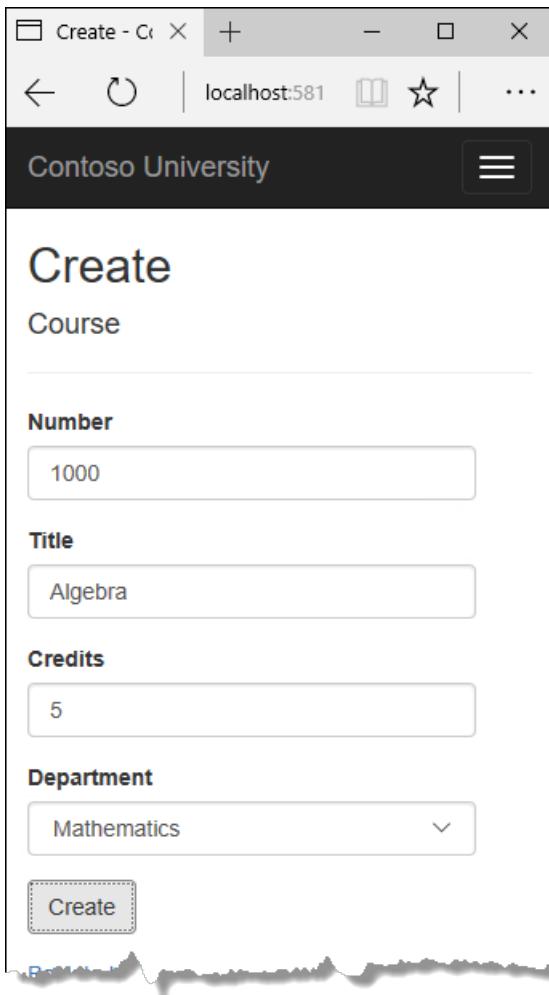
<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

在 Views/Courses/Details.cshtml 中，进行对 Delete.cshtml 所作相同的更改。

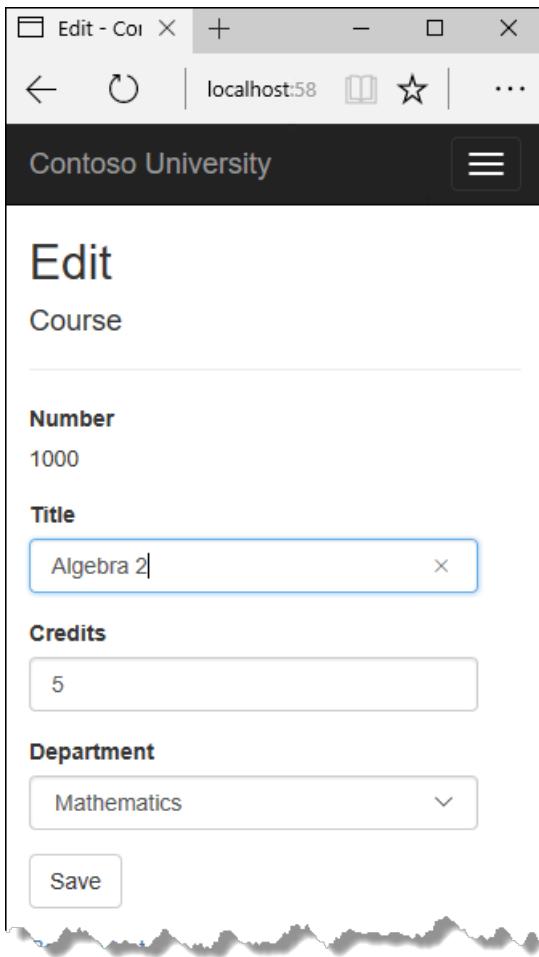
测试“课程”页

运行应用，选择“课程”选项卡，单击“新建”，然后输入新课程的数据：



单击“创建”。课程索引页面随即显示，并且新课程已添加在列表中。索引页列表中的院系名称来自导航属性，表明已正确建立关系。

在课程索引页中的课程上，单击“编辑”。



更改页面上的数据，然后单击“保存”。含有更新后的课程数据的“课程索引”页面随即显示。

添加讲师的编辑页面

编辑讲师记录时，有时希望能更新讲师的办公室分配。Instructor 实体和 OfficeAssignment 实体之间存在一对零或一的关系，这意味着代码必须处理一下情况：

- 如果用户清除了办公室分配，并且办公室分配最初具有一个值，则删除 OfficeAssignment 实体。
- 如果用户输入了办公室分配值，并且该值最初为空，则创建一个新的 OfficeAssignment 实体。
- 如果用户更改了办公室分配的值，则更改现有 OfficeAssignment 实体中的值。

更新讲师控制器

在 InstructorsController.cs 中，更改 `HttpGet Edit` 方法中的代码，使其加载 Instructor 实体的 `OfficeAssignment` 导航属性并调用 `AsNoTracking`：

```

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

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}

```

将 `HttpPost` `Edit` 方法更新为以下代码，以便处理办公室分配更新：

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}

```

该代码执行以下操作：

- 将方法名称更改为 `EditPost`，因为现在的签名与 `HttpGet` `Edit` 方法相同 (`ActionName` 特性指定仍然使用 `/Edit/` URL)。
- 使用 `OfficeAssignment` 导航属性的预先加载从数据库获取当前的 `Instructor` 实体。此操作与在 `HttpGet`

`Edit` 方法中进行的操作相同。

- 将检索出的 Instructor 实体更新为模型绑定器中的值。通过 `TryUpdateModel` 重载可以将想包括的属性列入到允许列表。这样可以防止[第二个教程](#)中所述的过度发布。

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- 如果办公室位置为空，请将 `Instructor.OfficeAssignment` 属性设置为 `NULL`，以便删除 `OfficeAssignment` 表中的相关行。

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

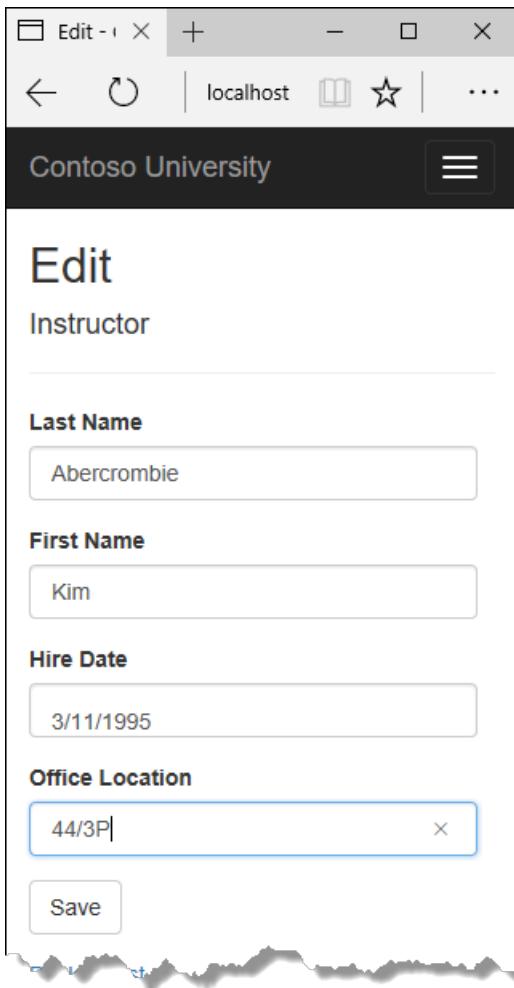
- 将更改保存到数据库。

更新讲师编辑视图

在 `Views/Instructors/Edit.cshtml` 中，在“保存”按钮之前的末尾处，添加一个用于编辑办公室位置的新字段：

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

运行应用，选择“讲师”选项卡，然后单击讲师页面上的“编辑”。更改“办公室位置”，然后单击“保存”。



向“讲师编辑”页添加课程分配

讲师可能教授任意数量的课程。现在可以通过使用一组复选框来更改课程分配，从而增强讲师编辑页面的性能，如以下屏幕截图所示：

The screenshot shows a web browser window titled "Edit - Contoso University". The URL is "localhost:5813/Instruct". The page content is titled "Edit Instructor". It contains fields for "Last Name" (Abercrombie), "First Name" (Kim), "Hire Date" (3/11/1995), and "Office Location" (44/3P). Below these are several checkboxes representing courses: "1000 Algebra 2", "1045 Calculus", "1050 Chemistry", "2021 Composition" (which is checked), "2042 Literature", "3141 Trigonometry", "4022 Microeconomics", and "4041 Macroeconomics". At the bottom left is a "Save" button.

Course 和 Instructor 实体之间是多对多的关系。若要添加和删除关系，可以向 CourseAssignments 联接实体集添加实体和从中删除实体。

用于更改讲师所对应的课程的 UI 是一组复选框。该复选框中会显示数据库中的所有课程，选中讲师当前对应的课程即可。用户可以通过选择或清除复选框来更改课程分配。如果课程的数量过大，建议使用其他方法在视图中呈现数据，但创建或删除关系的方法与操作联接实体的方法相同。

更新讲师控制器

若要为复选框列表的视图提供数据，将使用视图模型类。

在 SchoolViewModels 文件夹中创建 AssignedCourseData.cs，并将现有代码替换为以下代码：

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

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

在 InstructorsController.cs 中，将 `HttpGet Edit` 方法替换为以下代码。突出显示所作更改。

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

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

该代码为 `Courses` 导航属性添加了预先加载，并调用新的 `PopulateAssignedCourseData` 方法使用 `AssignedCourseData` 视图模型类为复选框数组提供信息。

`PopulateAssignedCourseData` 方法中的代码会读取所有 `Course` 实体，以便使用视图模型类加载课程列表。对每门课程而言，该代码都会检查讲师的 `Courses` 导航属性中是否存在该课程。为高效检查某门课程是否被分配给了讲师，可将分配给该讲师的课程放置于 `HashSet` 集合中。对于讲师分配到的课程，`Assigned` 属性则设置为 `true`。视图将使用此属性来确定应将哪些复选框显示为选中状态。最后，该列表会被传递给 `ViewData` 中的视图。

接下来，添加用户单击“保存”时执行的代码。将 `EditPost` 方法替换为以下代码，并添加一个新方法，用于更新 `Instructor` 实体的 `Courses` 导航属性。

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}

```

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}

```

现在的方法签名与 `HttpGet Edit` 方法不同，因此方法名称将从 `EditPost` 变回 `Edit`。

视图没有 `Course` 实体的集合，因此模型绑定器无法自动更新 `CourseAssignments` 导航属性。可在新的 `UpdateInstructorCourses` 方法中更新 `CourseAssignments` 导航属性，而不必使用模型绑定器。为此，需要从模型绑定中排除 `CourseAssignments` 属性。此操作无需对调用 `TryUpdateModel` 的代码进行任何更改，因为使用的是允许列表重载，并且 `CourseAssignments` 不包括在该列表中。

如果未选中任何复选框，则 `UpdateInstructorCourses` 中的代码将使用空集合初始化 `CourseAssignments` 导航属性，并返回以下内容：

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

之后，代码会循环访问数据库中的所有课程，并逐一检查当前分配给讲师的课程和视图中处于选中状态的课程。为便于高效查找，后两个集合存储在 `Instructor.CourseAssignments` 导航属性中。

如果某课程的复选框处于选中状态，但该课程不在 `Instructor.CourseAssignments` 导航属性中，则会将该课程添加到导航属性中的集合中。

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

如果某课程的复选框未处于选中状态，但该课程存在 `Instructor.CourseAssignments` 导航属性中，则会从导航属性中删除该课程。

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.SingleOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

更新讲师视图

在 Views/Instructors/Edit.cshtml 中，通过在“办公室”字段的 `div` 元素之后和“保存”按钮的 `div` 元素之前添加以下代码，以便添加带有一系列复选框的“课程”字段。

注意

将代码粘贴到 Visual Studio 中时，换行符会发生更改，从而导致代码中断。按 `Ctrl+Z` 一次可撤消自动格式设置。这样可以修复换行符，使其看起来如此处所示。缩进不一定要完美，但 `@</tr><tr>`、`@:<td>`、`@:</td>` 和 `@:</tr>` 行必须各成一行（如下所示），否则会出现运行时错误。选中新的代码块后，按 `Tab` 三次，使新代码与现有代码对齐。可在[此处](#)查看此问题的状态。

```

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
                        ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                name="selectedCourses"
                                value="@course.CourseID"
                                @(Html.Raw(course.Assigned ? "checked=\"checked\" : "")) />
                            @course.CourseID @: @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
            </table>
        </div>
    </div>

```

此代码将创建一个具有三列的 HTML 表。每个列中都有一个复选框，随后是一段由课程编号和标题组成的描述文字。所有复选框都具有相同的名称，即 selectedCourses，以告知模型绑定器将它们视为一组。每个复选框的值特性被设置为 `CourseID` 的值。发布页面时，模型绑定器会向控制器传递一个数组，其中只包括所选复选框的 `CourseID` 值。

这些复选框最开始呈现时，对于分配给讲师的课程的复选框，其特性处于选中状态。

运行应用，选择“讲师”选项卡，然后单击讲师页面上的“编辑”以查看“编辑”页面。

The screenshot shows a web browser window titled "Edit - Contoso University". The address bar displays "localhost:5813/Instruct". The main content area is titled "Edit Instructor". It contains the following form fields:

- Last Name:** Abercrombie
- First Name:** Kim
- Hire Date:** 3/11/1995
- Office Location:** 44/3P
- Courses Assigned:**
 - 1000 Algebra 2
 - 1045 Calculus
 - 1050 Chemistry
 - 2021 Composition
 - 2042 Literature
 - 3141 Trigonometry
 - 4022 Microeconomics
 - 4041 Macroeconomics
- Save** button

更改某些课程分配并单击“保存”。所作更改将反映在索引页上。

注意

此处所使用的编辑讲师课程数据的方法适用于数量有限的课程。若是远大于此的集合，则需要使用不同的 UI 和不同的更新方法。

更新“删除”页

在 InstructorsController.cs 中，删除 `DeleteConfirmed` 方法，并在其位置插入以下代码。

```

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}

```

此代码会更改以下内容：

- 对 `CourseAssignments` 导航属性执行预先加载。必须包括此内容，否则 EF 不知道相关的 `CourseAssignment` 实体，也不会删除它们。为避免在此处阅读它们，可以在数据库中配置级联删除。
- 如果要删除的讲师被指派为任何系的管理员，则需从这些系中删除该讲师分配。

向创建页添加办公室位置和课程

在 `InstructorsController.cs` 中，删除 `HttpGet` 和 `HttpPost` `Create` 方法，然后在其位置添加以下代码：

```

public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID =
int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

```

此代码与 `Edit` 方法中所示内容类似，只是最开始未选择任何课程。`HttpGet Create` 方法调用 `PopulateAssignedCourseData` 方法不是因为可能有课程处于选中状态，而是为了在视图中为 `foreach` 循环提供空集合（否则该视图代码将引发空引用异常）。

检查是否存在验证错误并向数据库添加新讲师之前，`HttpPost Create` 方法会将每个选定课程添加到 `CourseAssignments` 导航属性。即使存在模型错误也会添加课程，因此出现模型错误（例如用户键入了无效的日期）并且页面重新显示并出现错误消息时，所作的任何课程选择都会自动还原。

请注意，为了能够向 `CourseAssignments` 导航属性添加课程，必须将该属性初始化为空集合：

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

除了在控制器代码中进行此操作之外，还可以在“导师”模型中进行此操作，方法是将该属性更改为不存在集合时自动创建集合，如以下示例所示：

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

如果通过这种方式修改 `CourseAssignments` 属性，则可以删除控制器中的显式属性初始化代码。

在 `Views/Instructor/Create.cshtml` 中，添加一个办公室位置文本框和课程的复选框，然后按“提交”按钮。与“编辑”页面中一样，**如果粘贴代码时 Visual Studio 重新设置了其格式，则修复该格式**。

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
                    ViewBag.Courses;
                }

                foreach (var course in courses)
                {
                    if (cnt++ % 3 == 0)
                    {
                        @:</tr><tr>
                    }
                    @:<td>
                        <input type="checkbox"
                            name="selectedCourses"
                            value="@course.CourseID"
                            @(Html.Raw(course.Assigned ? "checked=\"checked\"" : ""))
                            @course.CourseID @: @course.Title
                    @:</td>
                }
                @:</tr>
            }
        </table>
    </div>
</div>
```

通过运行应用并创建讲师来进行测试。

处理事务

如 [CRUD 教程](#) 中所述, Entity Framework 隐式实现事务。如果需要更多控制操作(例如, 如果想要在事务中包含在 Entity Framework 外部完成的操作), 请参阅[事务](#)。

总结

处理相关数据的介绍至此已告一段落。下一个教程将介绍如何处理并发冲突。

ASP.NET Core MVC 和 EF Core - 并发 - 第 8 个教程 (共 10 个)

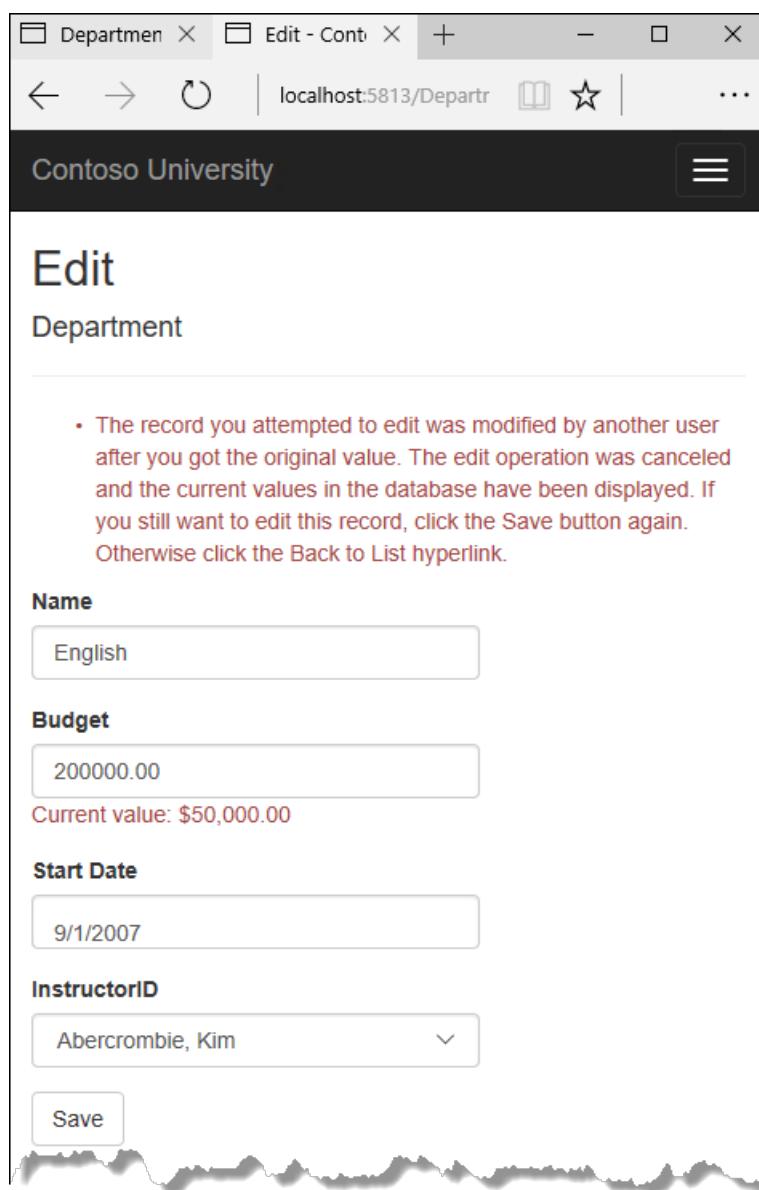
2018/5/17 • 21 min to read • [Edit Online](#)

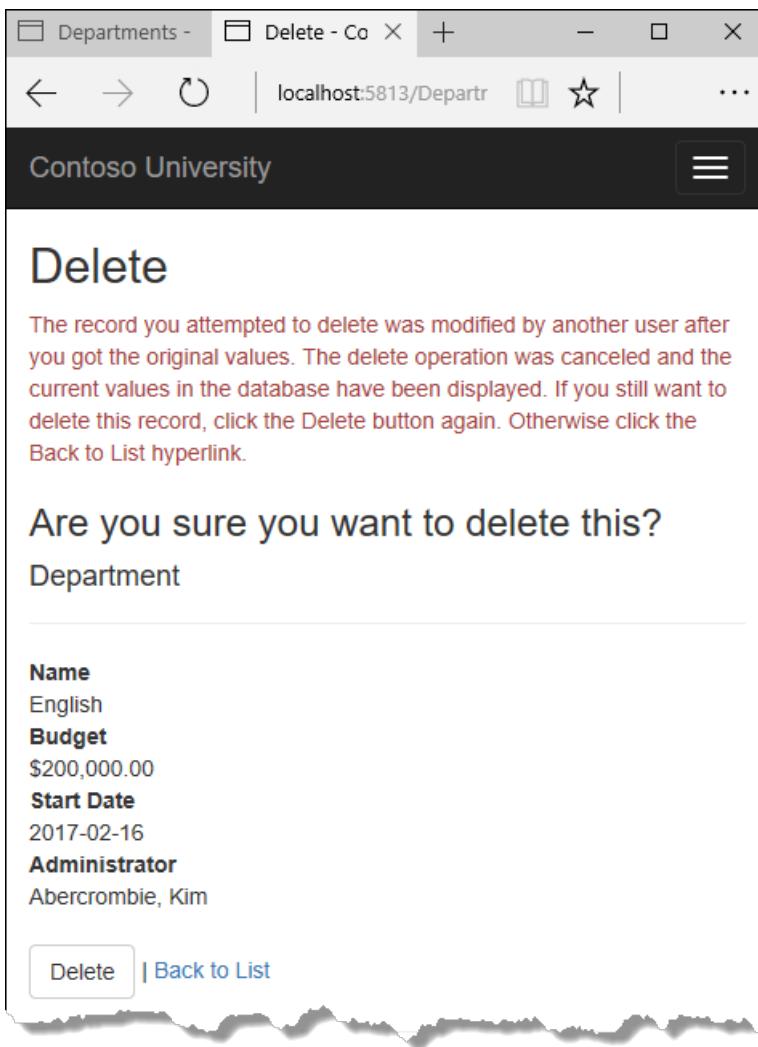
作者: Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

在之前的教程中, 你学习了如何更新数据。本教程介绍如何处理多个用户同时更新同一实体时出现的冲突。

你将创建可处理 Department 实体的 Web 页面并处理并发错误。下图显示了“编辑”和“删除”页面, 包括发生并发冲突时显示的一些消息。





并发冲突

当某用户显示实体数据以对其进行编辑，而另一用户在上一用户的更改写入数据库之前更新同一实体的数据时，会发生并发冲突。如果不启用此类冲突的检测，则最后更新数据库的人员将覆盖其他用户的更改。在许多应用程序中，此风险是可接受的：如果用户很少或更新很少，或者一些更改被覆盖并不重要，则并发编程可能弊大于利。在此情况下，不必配置应用程序来处理并发冲突。

悲观并发(锁定)

如果应用程序确实需要防止并发情况下出现意外数据丢失，一种方法是使用数据库锁定。这称为悲观并发。例如，在从数据库读取一行内容之前，请求锁定为只读或更新访问。如果将一行锁定为更新访问，则其他用户无法将该行锁定为只读或更新访问，因为他们得到的是正在更改的数据的副本。如果将一行锁定为只读访问，则其他人也可将其锁定为只读访问，但不能进行更新。

管理锁定有缺点。编程可能很复杂。它需要大量的数据库管理资源，且随着应用程序用户数量的增加，可能会导致性能问题。由于这些原因，并不是所有的数据库管理系统都支持悲观并发。Entity Framework Core 未提供对它的内置支持，并且本教程不展示其实现方式。

开放式并发

悲观并发的替代方法是乐观并发。悲观并发是指允许发生并发冲突，并在并发冲突发生时作出正确反应。例如，Jane 访问“院系编辑”页面，并将英语系的预算从 350,000.00 美元更改为 0.00 美元。

Edit

Department

Budget

Administrator

Name

Start Date

Save

在 Jane 单击“保存”之前, John 访问了相同页面, 并将开始日期字段从 2007/1/9 更改为 2013/1/9。

Edit

Department

Budget

Administrator

Name

Start Date

Save

Jane 先单击“保存”，并在浏览器返回索引页时看到她的更改。

Name	Budget	Administrator	Start Date	Action
English	\$0.00	Abercrombie, Kim	2007-09-01	Edit Details Delete
F&B	350,000.00	Eakhouri, Fadi	2013-01-09	Edit Details Delete

然后 John 单击“编辑”页面上的“保存”，但页面上预算仍显示为 350,000.00 美元。接下来的情况取决于并发冲突的处理方式。

其中一些选项包括：

- 可以跟踪用户已修改的属性，并仅更新数据库中相应的列。

在示例方案中，不会有数据丢失，因为是由两个用户更新不同的属性。下次有人浏览英语系时，将看到 Jane 和 John 两个人的更改 - 开始日期为 9/1/2013，预算为零美元。这种更新方法可减少可能导致数据丢失的冲突次数，但是如果对实体的同一属性进行竞争性更改，则数据难免会丢失。Entity Framework 是否以这种方式工作取决于更新代码的实现方式。通常不适合在 Web 应用程序中使用，因为它要求保持大量的状态，以便跟踪实体的所有原始属性值以及新值。维护大量的状态可能会影响应用程序的性能，因为它需要服务器资源或必须包含在网页本身（例如隐藏字段）或 Cookie 中。

- 可让 John 的更改覆盖 Jane 的更改。

下次有人浏览英语系时，将看到 2013/1/9 和还原的值 350,000.00 美元。这称为“客户端优先”或“最后一个优先”。（客户端的所有值优先于数据存储的值。）正如本部分的介绍所述，如果不为并发处理编写任何代码，则自动执行此操作。

- 可以阻止在数据库中更新 John 的更改。

通常，将显示一条错误消息，向他显示数据的当前状态，并允许他重新应用其更改（若他仍想更改）。这称为“存储优先”方案。（数据存储值优先于客户端提交的值。）本教程将执行“存储优先”方案。此方法可确保用户在未收到具体发生内容的警报时，不会覆盖任何更改。

检测并发冲突

你可以通过处理 Entity Framework 引发的 `DbConcurrencyException` 异常来解决冲突。为了知道何时引发这些异常，Entity Framework 必须能够检测到冲突。因此，你必须正确配置数据库和数据模型。启用冲突检测的某些选项包括：

- 数据库表中包含一个可用于确定某行更改时间的跟踪列。然后可配置 Entity Framework，将该列包含在 SQL Update 或 Delete 命令的 Where 子句中。

跟踪列的数据类型通常是 `rowversion`。`rowversion` 值是一个序列号，该编号随着每次行的更新递增。在 Update 或 Delete 命令中，Where 子句包含跟踪列的原始值（原始行版本）。如果正在更新的行已被其他用户更改，则 `rowversion` 列中的值与原始值不同，这导致 Update 或 Delete 语句由于 Where 子句而找不到要更新的行。当 Entity Framework 发现 Update 或 Delete 命令没有更新行（即受影响的行数为零）时，便将其解释为并发冲突。

- 配置 Entity Framework，在 Update 或 Delete 命令的 Where 子句中包含表中每个列的原始值。

与第一个选项一样，如果行自第一次读取后发生更改，Where 子句将不返回要更新的行，Entity Framework 会将其解释为并发冲突。对于包含许多列的数据库表，此方法可能导致非常多的 Where 子句，并且可能需要维持大量的状态。如前所述，维持大量的状态会影响应用程序的性能。因此通常不建议使用此方法，并且它也不是本教程中使用的方法。

如果确实想要实现此并发方法，必须通过向要跟踪并发的实体添加 `ConcurrencyCheck` 属性来标记它们的所有非主键属性。所作更改使 Entity Framework 能够将所有列包含在 Update 和 Delete 语句的 SQL Where 子句中。

在本教程的其余部分，将向 Department 实体添加 `rowversion` 跟踪属性，创建控制器和视图，并进行测试以验证是否一切正常工作。

向 Department 实体添加跟踪属性

在 Models/Department.cs 中，添加名为 RowVersion 的跟踪属性：

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

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

`Timestamp` 属性指定此列将包含在发送到数据库的 Update 和 Delete 命令的 Where 子句中。该属性称为 `Timestamp`，因为 SQL Server 的之前版本在 SQL `rowversion` 类型将其替换之前使用 SQL `timestamp` 数据类型。用于 `rowversion` 的 .NET 类型为字节数组。

如果更愿意使用 Fluent API，可使用 `IsConcurrencyToken` 方法（路径为 Data/SchoolContext.cs）指定跟踪属性，如下例所示：

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

通过添加属性，更改了数据库模型，因此需要再执行一次迁移。

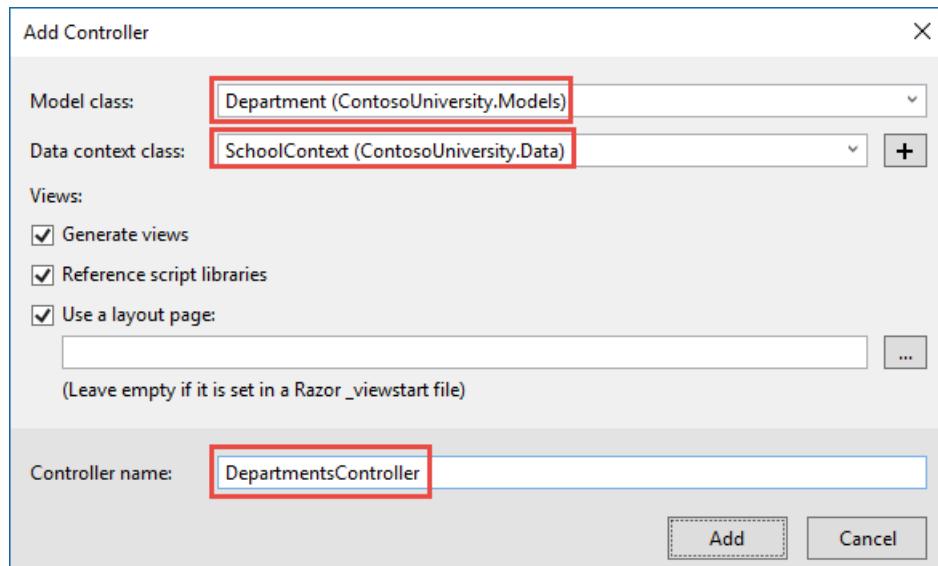
保存更改并生成项目，然后在命令窗口中输入以下命令：

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

创建“院系”控制器和视图

像之前在学生、课程和讲师教程中操作的那样，为“院系”控制器和视图创建基架。



在 DepartmentsController.cs 文件中，将出现的 4 次“FirstMidName”更改为“FullName”，以便院系管理员下拉列表将包含讲师的全名，而不仅仅是姓氏。

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

更新“院系索引”视图

基架引擎在索引视图中创建 RowVersion 列，但不应显示该字段。

将 Views/Departments/Index.cshtml 中的代码替换为以下代码。

```

@model IEnumerable<ContosoUniversity.Models.Department>

 @{
     ViewData["Title"] = "Departments";
 }

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

这会将标题更改为“院系”，删除 RowVersion 列，并显示全名（而非管理员的名字）。

更新“院系”控制器中的编辑方法

在 `HttpGet` `Edit` 方法和 `Details` 方法中，添加 `AsNoTracking`。在 `HttpGet` `Edit` 方法中，为管理员添加预先加载。

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);
```

将 `HttpPost` `Edit` 方法的现有代码替换为以下代码：

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).SingleOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {

```

```

        ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
    }
    if (databaseValues.InstructorID != clientValues.InstructorID)
    {
        Instructor databaseInstructor = await _context.Instructors.SingleOrDefaultAsync(i => i.ID
== databaseValues.InstructorID);
        ModelState.AddModelError("InstructorID", $"Current value:
{databaseInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty, "The record you attempted to edit "
+ "was modified by another user after you got the original value. The "
+ "edit operation was canceled and the current values in the database "
+ "have been displayed. If you still want to edit this record, click "
+ "the Save button again. Otherwise click the Back to List hyperlink.");
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
}
}
}
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
return View(departmentToUpdate);
}

```

代码先尝试读取要更新的院系。如果 `SingleOrDefaultAsync` 方法返回 `NULL`，则该院系已被另一用户删除。此情况下，代码将使用已发布的表单值创建院系实体，以便“编辑”页重新显示错误消息。或者，如果仅显示错误消息而未重新显示院系字段，则不必重新创建 `Department` 实体。

该视图将原始 `RowVersion` 值存储在隐藏字段中，且此方法在 `rowVersion` 参数中接收该值。在调用 `SaveChanges` 之前，必须将该原始 `RowVersion` 属性值置于实体的 `OriginalValues` 集合中。

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

然后，当 Entity Framework 创建 SQL UPDATE 命令时，该命令将包含一个 WHERE 子句，用于查找具有原始 `RowVersion` 值的行。如果没有行受到 UPDATE 命令影响（没有行具有原始 `RowVersion` 值），则 Entity Framework 会引发 `DbUpdateConcurrencyException` 异常。

该异常的 catch 块中的代码获取受影响的 `Department` 实体，该实体具有来自异常对象上的 `Entries` 属性的更新值。

```
var exceptionEntry = ex.Entries.Single();
```

`Entries` 集合将仅包含一个 `EntityEntry` 对象。该对象可用于获取用户输入的新值和当前的数据库值。

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

对于所含的数据库值与用户在“编辑”页上输入的值不同的每一列，该代码均为其添加一个自定义错误消息（为简洁起见，此处仅显示一个字段）。

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
}
```

最后，该代码将 `departmentToUpdate` 的 `RowVersion` 值设置为从数据库中检索到的新值。重新显示“编辑”页时，这个新的 `RowVersion` 值将存储在隐藏字段中，当用户下次单击“保存”时，将只捕获自“编辑”页重新显示起发生的并发错误。

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;  
ModelState.Remove("RowVersion");
```

`ModelState` 具有旧的 `RowVersion` 值，因此需使用 `ModelState.Remove` 语句。在此视图中，当两者都存在时，字段的 `ModelState` 值优先于模型属性值。

更新“院系编辑”视图

在 `Views/Departments/Edit.cshtml` 中，进行以下更改：

- 添加隐藏字段以保存 `RowVersion` 属性值，紧跟在 `DepartmentID` 属性的隐藏字段后面。
- 向下拉列表添加“选择管理员”选项。

```

@model ContosoUniversity.Models.Department

 @{
     ViewData["Title"] = "Edit";
 }

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

测试“编辑”页中的并发冲突

运行应用并转到“院系索引”页。右键单击英语系的“编辑”超链接，并选择“在新选项卡中打开”，然后单击英语系的“编辑”超链接。现在，两个浏览器选项卡显示相同的信息。

在第一个浏览器选项卡中更改一个字段，然后单击“保存”。

The screenshot shows a web browser window with two tabs: "Edit - Contoso" and "Edit - Contoso". The active tab displays a form titled "Edit Department". The form has fields for "Name" (containing "English"), "Budget" (containing "50000.00", which is highlighted with a red border), "Start Date" (containing "9/1/2007"), and "InstructorID" (containing "Abercrombie, Kim"). A "Save" button is at the bottom. The browser interface includes standard controls like back, forward, and search.

Name

English

Budget

50000.00

Start Date

9/1/2007

InstructorID

Abercrombie, Kim

Save

浏览器显示具有更改值的索引页。

在第二个浏览器选项卡中更改一个字段。

The screenshot shows a web browser window with the following details:

- Address Bar:** localhost:5813/Departments
- Title Bar:** Departments - Edit - Contoso University
- Content Area:**
 - Name:** English
 - Budget:** 200000.00 (This field is highlighted with a red border)
 - Start Date:** 9/1/2007
 - InstructorID:** Abercrombie, Kim
- Buttons:** Save

单击“保存”。看见一条错误消息：

The screenshot shows a browser window with two tabs: "Departmen X" and "Edit - Cont X". The address bar shows "localhost:5813/Departm...". The main content area is titled "Edit" and "Department". A red warning message states: "The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink." Below the message are form fields: "Name" (English), "Budget" (200000.00), "Start Date" (9/1/2007), and "InstructorID" (Abercrombie, Kim). A "Save" button is at the bottom.

再次单击“保存”。保存在第二个浏览器选项卡中输入的值。在索引页中出现时，可以看到已保存的值。

更新“删除”页

对于“删除”页，Entity Framework 以类似方式检测其他人编辑院系所引起的并发冲突。当 `HttpGet Delete` 方法显示确认视图时，该视图包含隐藏字段中的原始 `RowVersion` 值。然后，该值可用于在用户确认删除时进行调用的 `HttpPost Delete` 方法。当 Entity Framework 创建 SQL DELETE 命令时，它包含具有原始 `RowVersion` 值的 WHERE 子句。如果该命令不影响任何行（即在显示“删除”确认页面后更改了行），则引发并发异常，调用 `HttpGet Delete` 方法，并将错误标志设置为 true，以重新显示具有一条错误消息的确认页面。任意行均未受影响的原因也可能是，该行已被其他用户删除，因此在此情况下不显示错误消息。

更新“院系”控制器中的删除方法

在 `DepartmentsController.cs` 中，将 `HttpGet Delete` 方法替换为以下代码：

```

public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}

```

该方法接受可选参数，该参数指示是否在并发错误之后重新显示页面。如果此标志为 true 且指定的院系不复存在，则它已被其他用户删除。此情况下，代码将重定向到索引页。如果此标志为 true 且该院系确实存在，则它已被其他用户更改。此情况下，代码将使用 `ViewData` 向视图发送一条错误消息。

将 `HttpPost` `Delete` 方法（名为 `DeleteConfirmed`）中的代码替换为以下代码：

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID });
    }
}

```

在刚替换的基架代码中，此方法仅接受记录 ID：

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

已将此参数更改为由模型绑定器创建的 Department 实体实例。这样，EF 就可访问除记录键之外的 RowVersion 属性值。

```
public async Task<IActionResult> Delete(Department department)
```

你还将操作方法名称从 `DeleteConfirmed` 更改为了 `Delete`。基架代码使用 `DeleteConfirmed` 名称来为 `HttpPost` 方法提供唯一签名。(CLR 要求重载方法具有不同的方法参数。)既然签名是唯一的，就可继续使用 MVC 约定，并为 `HttpPost` 和 `HttpGet` 删除方法使用同一名称。

如果已删除院系，则 `AnyAsync` 方法返回 `false`，应用程序仅返回到 `Index` 方法。

如果捕获到并发错误，代码将重新显示“删除”确认页，并提供一个指示它应显示并发错误消息的标志。

更新“删除”视图

在 `Views/Departments/Delete.cshtml` 中，将基架代码替换为添加 `DepartmentID` 和 `RowVersion` 属性的错误消息字段和隐藏字段的以下代码。突出显示所作更改。

```

@model ContosoUniversity.Models.Department

 @{
     ViewData["Title"] = "Delete";
 }

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>

```

这将进行以下更改：

- 在 `h2` 和 `h3` 标题之间添加错误消息。
- 将“管理员”字段中的 `FirstMidName` 替换为 `FullName`。
- 删除 `RowVersion` 字段。
- 添加 `RowVersion` 属性的隐藏字段。

运行应用并转到“院系索引”页。右键单击英语系的“删除”超链接，并选择“在新选项卡中打开”，然后在第一个选项卡中单击英语系的“编辑”超链接。

在第一个窗口中，更改其中一个值，然后单击“保存”：

Edit

Department

Name

English

Budget

200000.00

Start Date

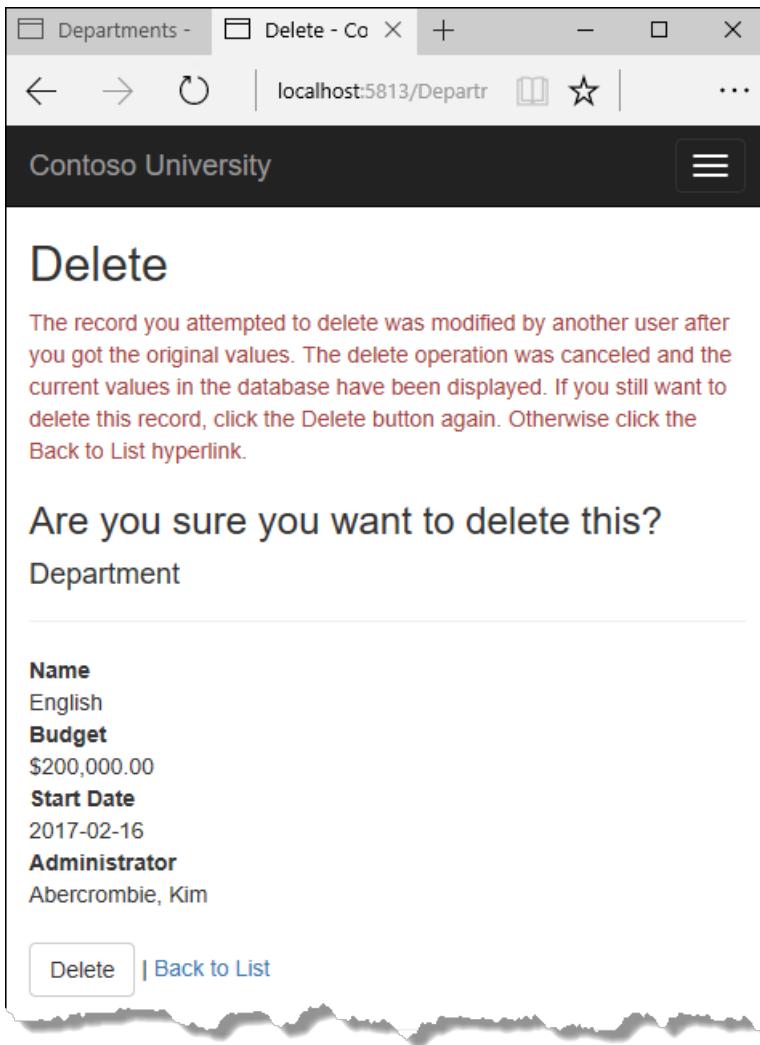
2/16/2017

InstructorID

Abercrombie, Kim

Save

在第二个选项卡中，单击“删除”。你将看到并发错误消息，且已使用数据库中的当前内容刷新了“院系”值。



如果再次单击“删除”，会重定向到已删除显示院系的索引页。

更新“详细信息”和“创建”视图

可选择性地清理“详细信息”和“创建”视图中的基架代码。

替换 Views/Departments/Details.cshtml 中的代码，以删除 RowVersion 列并显示管理员的全名。

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>



<h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>


<div>
    <a href="#">Edit</a> |
    <a href="#">Back to List</a>
</div>
```

替换 Views/Departments/Create.cshtml 中的代码，向下拉列表添加“选择”选项。

```

@model ContosoUniversity.Models.Department

 @{
     ViewData["Title"] = "Create";
 }

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}

```

总结

处理并发冲突已介绍完毕。要深入了解如何处理 EF Core 中的并发，请参阅[并发冲突](#)。下一个教程将介绍如何为 Instructor 和 Students 实体实现“每个层次结构一个表”继承。

ASP.NET Core MVC 和 EF Core - 继承 - 第 9 个教程(共 10 个)

2018/5/17 • 9 min to read • [Edit Online](#)

作者 : Tom Dykstra 和 Rick Anderson

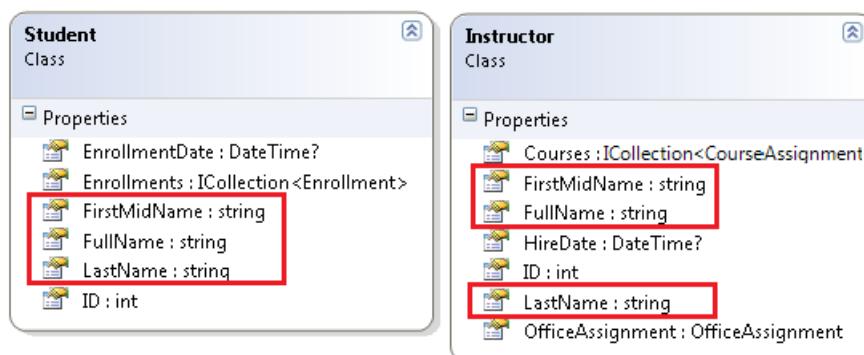
Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解教程系列, 请参阅[本系列中的第一个教程](#)。

在上一个教程中, 已经处理了并发异常。本教程将演示如何在数据模型中实现继承。

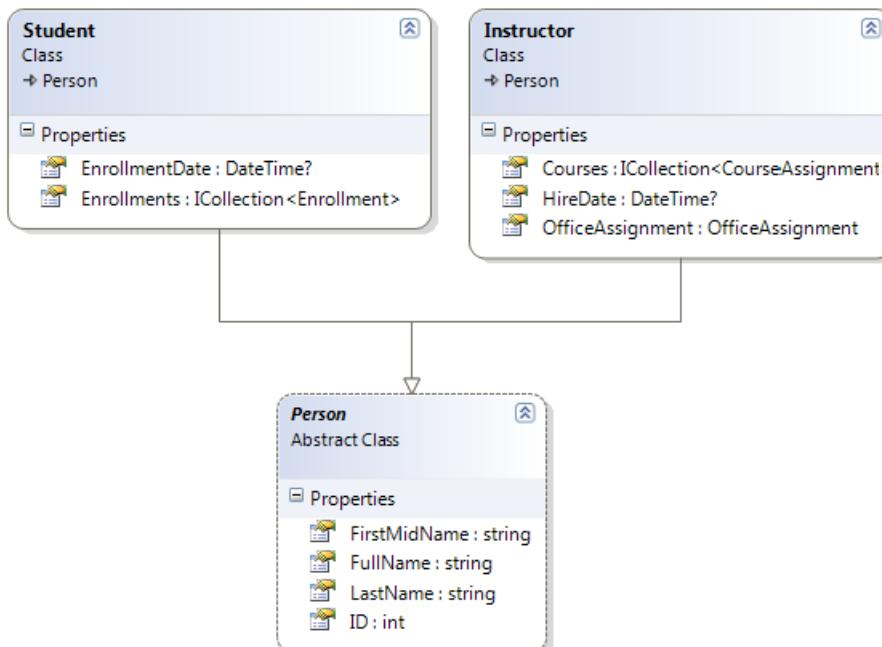
在面向对象的编程中, 可以使用继承以便于重用代码。在本教程中, 将更改 `Instructor` 和 `Student` 类, 以便从 `Person` 基类中派生, 该基类包含教师和学生所共有的属性(如 `LastName`)。不会添加或更改任何网页, 但会更改部分代码, 并将在数据库中自动反映这些更改。

将继承映射到数据库表的选项

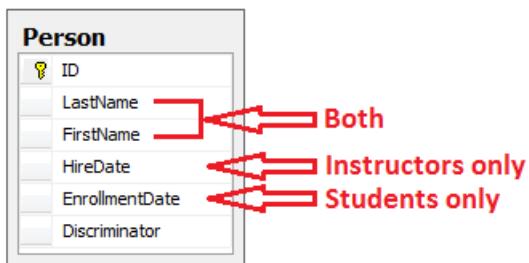
学校数据模型中的 `Instructor` 和 `Student` 类具有多个相同的属性:



假设想要消除由 `Instructor` 和 `Student` 实体共享的属性的冗余代码。或者想要写入可以格式化姓名的服务, 而无需关注该姓名来自教师还是学生。可以创建只包含这些共享属性的 `Person` 基类, 然后使 `Instructor` 和 `Student` 类继承该基类, 如下图所示:

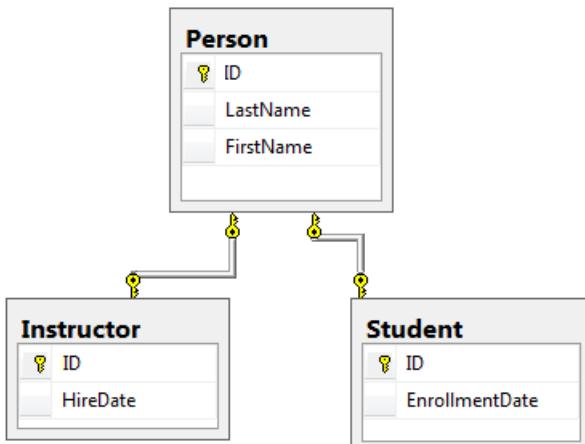


有多种方法可以在数据库中表示此继承结构。可以创建一个 Person 表，将学生和教师的相关信息包含在一个表中。某些列可能仅适用于教师 (HireDate)，某些列仅适用于学生 (EnrollmentDate)，某些列同时适用于两者 (LastName、FirstName)。通常情况下，将有一个鉴别器列来指示每行所代表的类型。例如，鉴别器列可能包含“Instructor”来指示教师，包含“Student”来指示学生。



从单个数据库表生成实体继承结构的模式称为每个层次结构一张 (TPH) 继承。

另一种方法是使数据库看起来更像继承结构。例如，可以仅将姓名字段包含到 Person 表中，在单独的 Instructor 和 Student 表中包含日期字段。



为每个实体类创建数据库表的模式称为每个类型一张表 (TPT) 继承。

另一种方法是将所有非抽象类型映射到单独的表。类的所有属性（包括继承的属性）映射到相应表的列。此模式称为每个具体类一张表 (TPC) 继承。如果为前面所示的 Person、Student 和 Instructor 类实现了 TPC 继承，那么在实现继承之后，Student 和 Instructor 表看起来将与以前没什么不同。

TPC 和 TPH 继承模式的性能通常比 TPT 继承模式好，因为 TPT 模式会导致复杂的联接查询。

本教程将演示如何实现 TPH 继承。TPH 是 Entity Framework Core 唯一支持的继承模式。需要执行的操作是创建 Person 类、将 Instructor 和 Student 类更改为从 Person 派生、将新的类添加到 DbContext，以及创建迁移。

提示

在进行以下更改之前，请考虑保存项目的副本。如果遇到问题并需要重新开始，可以更轻松地从已保存的项目开始，而不用反向操作本教程中的步骤或者返回到整个系列的开始。

创建 Person 类

在 Models 文件夹中，创建 Person.cs 并使用以下代码替换模板代码：

```

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

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}

```

使 Student 和 Instructor 类从 Person 继承

在 Instructor.cs 中，从 Person 类派生 Instructor 类并删除键和姓名字段。代码将如下所示：

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

在 Student.cs 中做出相同更改。

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

将 Person 实体类型添加到数据模型

将 Person 实体类型添加到 SchoolContext.cs。新的行突出显示。

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}

```

以上是 Entity Framework 配置每个层次结构一张表继承所需的全部操作。正如将看到的，更新数据库时，将有一个 Person 表来代替 Student 和 Instructor 表。

创建和自定义迁移代码

保存更改并生成项目。随后在项目文件夹中打开命令窗口并输入以下命令：

```
dotnet ef migrations add Inheritance
```

暂不运行 `database update` 命令。该命令将导致数据丢失，因为它将删除 Instructor 表并将 Student 表重命名为 Person。需要提供自定义代码来保留现有数据。

打开 Migrations/<timestamp>_Inheritance.cs 并使用以下代码替换 `Up` 方法：

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate,
Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS
Discriminator, ID AS OldId FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId =
Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
        name: "IX_Enrollment_StudentID",
        table: "Enrollment",
        column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

此代码负责以下数据库更新任务：

- 删除指向 Student 表的外键约束和索引。
- 将 Instructor 表重命名为 Person，根据需要做出更改以存储学生数据：
- 为学生添加可为 NULL 的 EnrollmentDate。
- 添加鉴别器列来指示行代表学生还是教师。

- HireDate 可为 NULL，因为学生行不会包含聘用日期。
- 添加临时字段，用于更新指向学生的外键。将学生复制到 Person 表时，将获取新的主键值。
- 将数据从 Student 表复制到 Person 表。这将使学生获取分配的新主键值。
- 修复指向学生的外键值。
- 重新创建外键约束和索引，现在将它们指向 Person 表。

(如果已使用 GUID 而不是整数作为主键类型，那么将不需要更改学生主键值，并且可能已省略其中多个步骤。)

运行 `database update` 命令：

```
dotnet ef database update
```

(在生产系统中，可以对 `Down` 方法进行相应更改，以防必须使用该方法返回到以前的数据库版本。本教程中将不使用 `Down` 方法。)

注意

在包含现有数据的数据库中更改架构时，可能会发生其他错误。如果出现无法解决的迁移错误，可以在连接字符串中更改数据库名或者删除数据库。若是新数据库，则没有要迁移的数据，因此在完成更新数据库命令时很可能不会出错。若要删除数据库，请使用 SSOX 或运行 `database drop` CLI 命令。

使用已实现的继承进行测试

运行应用并尝试各种页面。一切都和以前一样。

在“SQL Server 对象资源管理器”中，展开“数据连接/SchoolContext”和“表”，将看到 Student 和 Instructor 表已替换为 Person 表。打开 Person 表设计器，将看到它包含在 Student 和 Instructor 表中使用的所有列。

The screenshot shows the SQL Server Object Explorer with the ContosoUniversity database selected. The Person table is currently being edited in the designer. The table structure includes columns for ID (int, primary key, identity), FirstName (nvarchar(50)), HireDate (datetime2(7)), LastName (nvarchar(50)), EnrollmentDate (datetime2(7)), and Discriminator (nvarchar(128) with a default value of 'N'Instructor')). On the right side of the designer, there are sections for Keys (1), Check Constraints (0), Indexes (0), Foreign Keys (0), and Triggers (0). Below the table structure, the T-SQL script for creating the table is visible:

```

CREATE TABLE [dbo].[Person] (
    [ID] INT IDENTITY (1, 1) NOT NULL,
    [FirstName] NVARCHAR (50) NOT NULL,
    [HireDate] DATETIME2 (7) NULL,
    [LastName] NVARCHAR (50) NOT NULL,
    [EnrollmentDate] DATETIME2 (7) NULL,
    [Discriminator] NVARCHAR (128) DEFAULT (N'Instructor') NOT NULL,
    CONSTRAINT [PK_Instructor] PRIMARY KEY CLUSTERED ([ID] ASC)
);

```

右键单击 Person 表，然后单击“显示表数据”以查看鉴别器列。

The screenshot shows a database table named 'dbo.Person' in SSMS. The table has columns: ID, FirstName, HireDate, LastName, EnrollmentStatus, and Discriminator. The data consists of 12 rows, each representing a person's information. The 'Discriminator' column is used to identify the type of entity (Instructor or Student).

	ID	FirstName	HireDate	LastName	EnrollmentStatus	Discriminator
▶	1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
	2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
	3	Roger	7/1/1998 ...	Harui	NULL	Instructor
	4	Candace	1/15/2001...	Kapoor	NULL	Instructor
	5	Roger	2/12/2004...	Zheng	NULL	Instructor
	7	Nancy	8/17/2016...	Davolio	NULL	Instructor
	8	Carson	NULL	Alexander	9/1/2010 ...	Student
	9	Meredith	NULL	Alonso	9/1/2012 ...	Student
	10	Arturo	NULL	Anand	9/1/2013 ...	Student
	11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
	12	Yan	NULL	Li	9/1/2012	Student

总结

你已经为 `Person`、`Student` 和 `Instructor` 类实现了每个层次结构一张表继承。若要详细了解 Entity Framework Core 中的继承, 请参阅[继承](#)。下一个教程将介绍如何处理各种相对高级的 Entity Framework 方案。

[上一页](#) [下一页](#)

使用 EF Core 创建 ASP.NET Core MVC - 高级 - 第 10 个教程(共 10 个)

2018/5/17 • 15 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Rick Anderson

Contoso 大学示例 web 应用程序演示如何使用 Entity Framework Core 和 Visual Studio 创建 ASP.NET Core MVC web 应用程序。若要了解系列教程, 请参阅[本系列中的第一个教程](#)。

之前的教程中, 已经以每个类一张表的方式实现了继承。本教程将会介绍在掌握开发基础 ASP.NET Core web 应用程序之后使用 Entity Framework Core 开发时需要注意的几个问题。

原生 SQL 查询

使用 Entity Framework 的优点之一是它可避免你编写跟数据库过于耦合的代码。它会自动生成 SQL 查询和命令, 使得你无需自行编写。但有一些特殊情况, 你需要执行手动创建的特定 SQL 查询。对于这些情况下, Entity Framework Code First API 包括直接传递 SQL 命令将到数据库的方法。在 EF Core 1.0 中具有以下选项:

- 使用 `DbSet.FromSql` 返回实体类型的查询方法。返回的对象必须是 `DbSet` 对象期望的类型, 并且它们会自动跟踪数据库上下文中除非你[手动关闭跟踪](#)。
- 对于非查询命令使用 `Database.ExecuteSqlCommand`。

如果需要运行该返回类型不是实体的查询, 你可以使用由 EF 提供的 ADO.NET 中使用数据库连接。数据库上下文不会跟踪返回的数据, 即使你使用该方法来检索实体类型也是如此。

在 Web 应用程序中执行 SQL 命令时, 请务必采取预防措施来保护站点免受 SQL 注入攻击。一种方法是使用参数化查询, 确保不会将网页提交的字符串视为 SQL 命令。在本教程中, 将用户输入集成到查询中时会使用参数化查询。

调用返回实体的查询

`DbSet< TEntity >` 类提供了可用于执行查询并返回 `TEntity` 类型实体的方法。若要查看实现细节, 你需要更改部门控制器中 `Details` 方法的代码。

在 `DepartmentsController.cs` 中的 `Details` 方法, 通过代码调用 `FromSql` 方法检索一个部门, 如以下高亮代码所示:

```

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

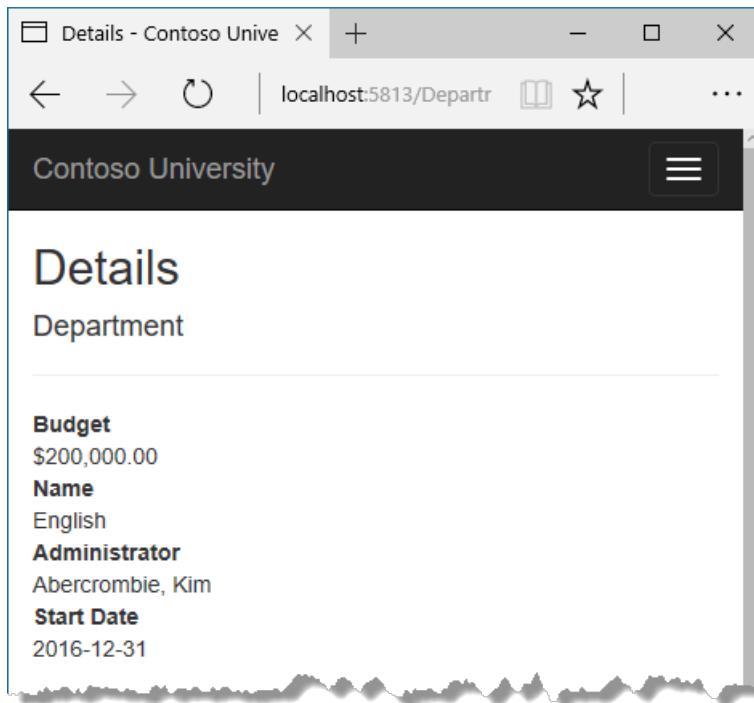
    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}

```

为了验证新代码是否工作正常,请选择**Department**选项卡,然后点击某个部门的**Detail**。



调用返回其他类型的查询

之前你在“关于”页面创建了一个学生统计信息网格,显示每个注册日期的学生数量。可以从学生实体集中获取数据(`_context.Students`),使用 LINQ 将结果投影到 `EnrollmentDateGroup` 视图模型对象的列表。假设你想要 SQL 本身编写,而不使用 LINQ。需要运行 SQL 查询中返回实体对象之外的内容。在 EF Core 1.0 中,执行该操作的另一种方法是编写 ADO.NET 代码,并从 EF 获取数据库连接。

在 `HomeController.cs` 中,将 `About` 方法替换为以下代码:

```

public async Task<ActionResult> About()
{
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

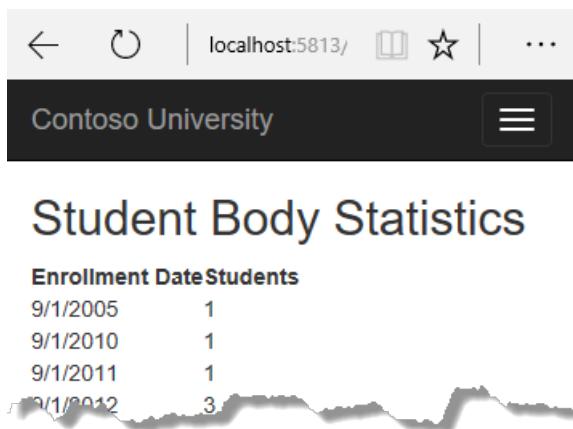
            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount
= reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}

```

添加 using 语句：

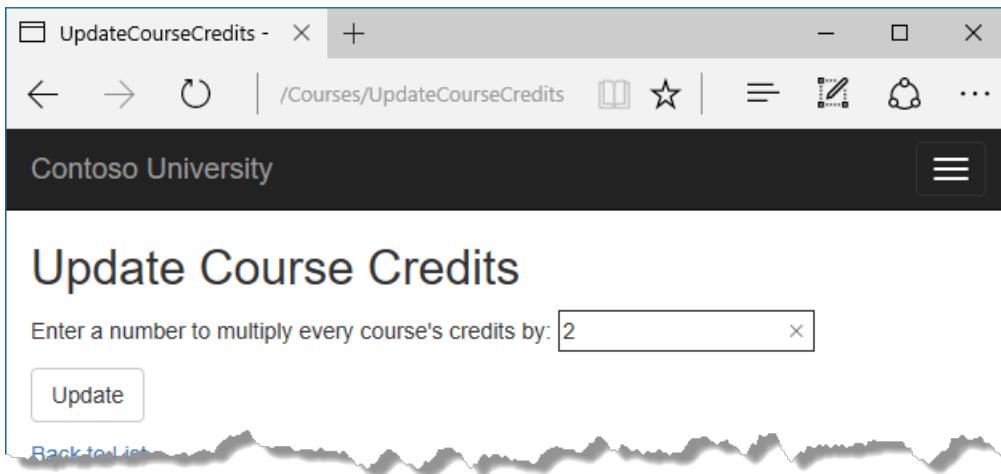
```
using System.Data.Common;
```

运行应用并转到“关于”页面。显示的数据和之前一样。



调用更新查询

假设 Contoso 大学管理员想要在数据库中执行全局更改，例如如更改的每个课程的可修读人数。如果该大学有大量的课程，检索所有实体并单独更改会降低效率。在本节中，你将实现一个页面，使用户能够指定一个参数，通过这个参数可以更改所有课程的可修读人数，在这里你会通过执行 SQL UPDATE 语句来进行更改。页面外观类似于下图：



在 CoursesController.cs 中，为 `HttpGet` 和 `HttpPost` 添加 `UpdateCourseCredits` 方法：

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

[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

当控制器处理 `HttpGet` 请求时，`ViewData["RowsAffected"]` 中不会返回任何东西，并且在视图中显示一个空文本框和提交按钮，如上图所示。

当单击 **Update** 按钮时，将调用 `HttpPost` 方法，且从文本框中输入的值获取乘数。代码接着执行 SQL 语句更新课程，并向视图的 `ViewData` 返回受影响的行数。当视图获取 `RowsAffected` 值，它将显示更新的行数。

在“解决方案资源管理器”中，右键单击“Views/Courses”文件夹，然后依次单击“添加”和“新建项”。

在添加新项对话框中，在已安装下 **单击 ASP.NET**，在左窗格中，单击 **MVC** 视图页，并将新视图命名为 `UpdateCourseCredits.cshtml`。

在 `Views/Courses/UpdateCourseCredits.cshtml` 中，将模板代码替换为以下代码：

```

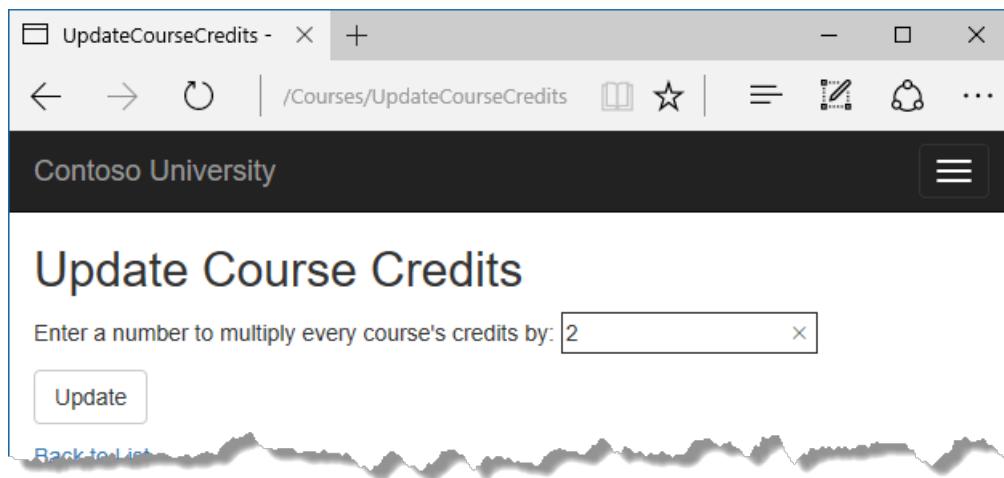
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

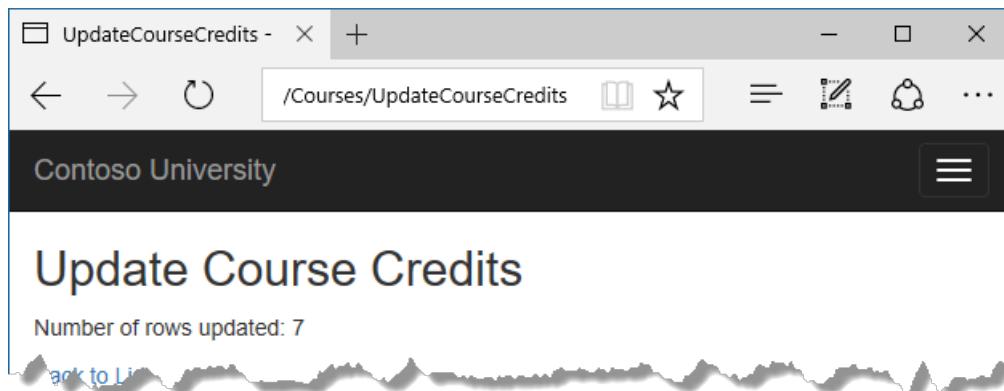
@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

通过选择**Courses**选项卡运行 `UpdateCourseCredits` 方法，然后在浏览器地址栏中 URL 的末尾添加"/`UpdateCourseCredits`"到（例如：<http://localhost:5813/Courses/UpdateCourseCredits>）。在文本框中输入数字：



单击**Update**。你会看到受影响的行数：



单击**Back To List**可以看到课程列表，其中可修读人数已经替换成修改后的数字。

请注意生产代码将确保更新最终得到有效的数据。此处所示的简化代码会使得相乘后可修读人数大于 5。(Credits 属性具有 [Range(0, 5)] 特性。)更新查询将起作用, 但无效的数据会导致意外的结果, 例如在系统的其他部分中加入可修读人数为 5 或更少可能会导致意外的结果。

有关原生 SQL 查询的详细信息, 请参阅[原生 SQL 查询](#)。

检查发送到数据库的 SQL

有时能够以查看发送到数据库的实际 SQL 查询对于开发者来说是很有用的。EF Core 自动使用 ASP.NET Core 的内置日志记录功能来编写包含 SQL 查询和更新的日志。在本部分中, 你将看到记录 SQL 日志的一些示例。

打开 `StudentsController.cs` 并在 `Details` 方法的 `if (student == null)` 语句上设置断点。

在调试模式下运行应用, 并转到某位学生的“详细信息”页面。

转到输出窗口显示调试输出, 就可以看到查询语句:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=@__id_0='?'], CommandType='Text', CommandTimeout='30'
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=@__id_0='?'], CommandType='Text', CommandTimeout='30'
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

你会注意到一些可能会让你觉得惊讶的操作: SQL 从 Person 表最多选择 2 行 (`TOP(2)`)。`SingleOrDefaultAsync` 方法服务器上不会解析为 1 行。原因是:

- 如果查询返回多个行, 该方法会返回 `null`。
- 如果想知道查询是否返回多个行, EF 必须检查是否至少返回 2。

请注意, 你不必使用调试模式, 并在断点处停止, 然后在输出窗口获取日志记录。这种方法非常便捷, 只需在想查看输出时停止日志记录即可。如果不进行此操作, 程序将继续进行日志记录, 要查看感兴趣的部分则必须向后滚动。

存储库和工作单元模式

许多开发人员编写代码实现存储库和工作模式单元以作为使用 Entity Framework 代码的包装器。这些模式用于在应用程序的数据访问层和业务逻辑层之间创建抽象层。实现这些模式可让你的应用程序对数据存储介质的更改不敏感, 而且很容易进行自动化单元测试和进行测试驱动开发 (TDD)。但是, 编写附加代码以实现这些模式对于使用 EF 的应用程序并不总是最好的选择, 原因有以下几个:

- EF 上下文类可以为使用 EF 的数据库更新充当工作单位类。
- 对于使用 EF 进行的数据库更新, EF 上下文类可充当工作单元类。
- EF 包括用于无需编写存储库代码就实现 TDD 的功能。

有关如何实现存储库和工作单元模式的详细信息, 请参阅[本系列教程的 Entity Framework 5 版本](#)。

Entity Framework Core 的实现可用于测试的内存数据库驱动程序。有关详细信息, 请参阅[测试以及 InMemory](#)。

自动脏值检测

Entity Framework 通过比较的实体的当前值与原始值来判断更改实体的方式 (因此需要发送更新到数据库)。查询或附加该实体时会存储的原始值。如下方法会导致自动脏值:

- `DbContext.SaveChanges`
- `DbContext.Entry`
- `ChangeTracker.Entries`

如果正在跟踪大量实体, 并且这些方法之一在循环中多次调用, 通过使用 `ChangeTracker.AutoDetectChangesEnabled` 属性暂时关闭自动脏值检测, 能够显著改进性能。例如:

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

Entity Framework Core 源代码与开发计划

Entity Framework Core 源位于 <https://github.com/aspnet/EntityFrameworkCore>。仓库中除了有源代码, 还可包括每夜生成、问题跟踪、功能规范、设计会议备忘录和[将来的开发路线图](#)。你可以归档或查找 bug 并进行更改。

尽管源代码处于开源状态, Entity Framework Core 是由 Microsoft 完全支持的产品。Microsoft Entity Framework 团队将控制接受哪些贡献和测试所有的代码更改, 以确保每个版本的质量。

现有数据库逆向工程

若想要通过对现有数据库中的实体类反向工程得出数据模型, 可以使用[scaffold-dbcontext](#)。可以参阅[入门教程](#)。

使用动态 LINQ 来简化对所选内容排序的代码

[本系列的第三个教程](#)演示如何通过在 `switch` 语句中对列名称进行硬编码来编写 LINQ。如果只有两列可供选择, 这种方法可行, 但是如果拥有许多列, 代码可能会变得冗长。要解决该问题, 可使用 `EF.Property` 方法将属性名称指定为一个字符串。要尝试此方法, 请将 `StudentsController` 中的 `Index` 方法替换为以下代码。

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] =
        String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
    ViewData["DateSortParm"] =
        sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                  select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                           || s.FirstMidName.Contains(searchString));
    }

    if (string.IsNullOrEmpty(sortOrder))
    {
        sortOrder = "LastName";
    }

    bool descending = false;
    if (sortOrder.EndsWith("_desc"))
    {
        sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
        descending = true;
    }

    if (descending)
    {
        students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));
    }
    else
    {
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        page ?? 1, pageSize));
}

```

后续步骤

在这将完成在 ASP.NET MVC 应用程序中使用 Entity Framework Core 这一系列教程。

有关 EF Core 的详细信息, 请参阅 [Entity Framework 的Core文档](#)。另外也可参阅 [Entity Framework Core in Action](#)(实际运用 Entity Framework Core)一书。

有关如何部署 web 应用程序的信息, 请参阅[托管和部署](#)。

有关 ASP.NET Core MVC 相关的其他主题 (如身份验证与授权) 的信息, 请参阅[ASP.NET Core 文档](#)。

鸣谢

Tom Dykstra 和 Rick Anderson (twitter @RickAndMSFT) 共同编写了本教程。Rowan Miller、Diego Vega 和 Entity Framework 团队的其他成员协助代码评审和帮助解决编写教程代码时出现的调试问题。

常见错误

ContosoUniversity.dll 被另一个进程使用

错误消息:

```
无法打开"...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- "进程无法访问文件"...\\bin\\Debug\\netcoreapp1.0\\ContosoUniversity.dll", 因为它正在由其他进程使用。
```

解决方案:

停止 IIS Express 中的站点。请转到 Windows 系统任务栏中, 找到 IIS Express 并右键单击其图标、选择 Contoso 大学站点, 然后单击[停止站点](#)。

迁移基架的 Up 和 Down 方法中没有代码

可能的原因:

EF CLI 命令不会自动关闭并保存代码文件。如果在运行 `migrations add` 命令时, 你未保存更改, EF 将找不到所做的更改。

解决方案:

运行 `migrations remove` 命令, 保存你更改的代码并重新运行 `migrations add` 命令。

运行数据库更新时出错

在有数据的数据库中进行架构更改时, 很有可能发生其他错误。如果遇到无法解决的迁移错误, 你可以更改连接字符串中的数据库名称, 或删除数据库。若要迁移, 创建新的数据库, 在数据库尚没有数据时使用更新数据库命令更有希望完成且不发生错误。

最简单方法是在 `appsettings.json` 中重命名数据库。下次运行 `database update` 时, 会创建一个新数据库。

若要在 SSOX 中删除数据库, 右键单击数据库, 单击[删除](#), 然后在[删除数据库](#)对话框框中, 选择[关闭现有连接](#), 单击[确定](#)。

若要使用 CLI 删除数据库, 可以运行 `database drop` CLI 命令:

```
dotnet ef database drop
```

定位 SQL Server 实例出错

错误消息:

```
建立到 SQL Server 的连接时出现与网络相关或特定于实例的错误。未找到或无法访问服务器。请验证实例名称是否正确, SQL Server 是否已配置为允许远程连接。(提供程序:SQL 网络接口, 错误:26 - 定位指定的服务器/实例时出错)
```

解决方案:

请检查连接字符串。如果你已手动删除数据库文件, 更改数据库的构造字符串中数据库的名称, 然后从头开始使

用新的数据库。

[上一篇](#)

ASP.NET Core 教程

2018/5/14 • 1 min to read • [Edit Online](#)

下面的分步指南已可用于开发 ASP.NET Core 应用程序：

生成 Web 应用

[Razor 页面](#) 是使用 ASP.NET Core 2.0 及更高版本创建新 Web UI 应用时建议使用的方法。

- [ASP.NET Core 中的 Razor 页面介绍](#)
- 使用 ASP.NET Core 创建 Razor 页面 Web 应用
 - [macOS 上的 Razor 页面](#)
 - [使用 VS Code 创建 Razor 页面](#)
- 创建 ASP.NET Core MVC Web 应用
 - [使用 Visual Studio for Mac 创建 Web 应用](#)
 - [在 macOS 或 Linux 上使用 Visual Studio Code 创建 Web 应用](#)

生成 Web API

- 使用 ASP.NET Core 创建 Web API
 - [使用 Visual Studio for Mac 创建 Web API](#)
 - [使用 Visual Studio Code 创建 Web API](#)

在带有 Visual Studio for Mac 的 macOS 上使用 ASP.NET Core 创建 Razor 页面 Web 应用

2018/4/27 • 1 min to read • [Edit Online](#)

我们正在撰写此系列的文章。

此系列介绍了在 macOS 上使用 ASP.NET Core 生成 Razor 页面 Web 应用的基础知识。

1. [macOS 上的 Razor 页面入门](#)
2. [向 Razor 页面应用添加模型](#)
3. [已搭建基架的 Razor 页面](#)
4. [使用 SQLite](#)
5. [更新页面](#)
6. [添加搜索](#)

在后续部分完成前, 请先按照 Visual Studio for Windows 版本操作。

1. [添加新字段](#)
2. [添加验证](#)

借助 Visual Studio for Mac 在 macOS 上的 ASP.NET Core 中开始使用 Razor 页面

2018/5/17 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍构建 ASP.NET Core Razor Pages Web 应用的基础知识。建议在学习本教程前先查看 [Razor 页面介绍](#)。Razor 页面是在 ASP.NET Core 中为 Web 应用程序生成 UI 时建议使用的方法。

系统必备

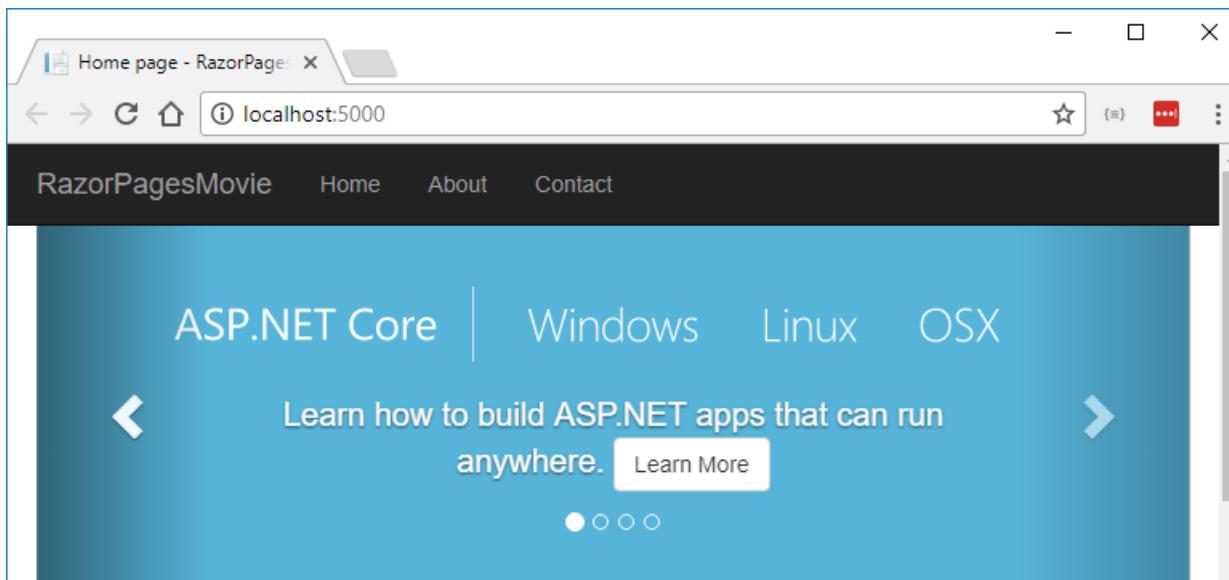
[Visual Studio for Mac](#)

创建 Razor Web 应用

从终端运行以下命令:

```
dotnet new razor -o RazorPagesMovie  
cd RazorPagesMovie  
dotnet run
```

上述命令使用 [.NET Core CLI](#) 创建并运行 Razor 页面项目。打开浏览器，转到 <http://localhost:5000> 查看应用程序。



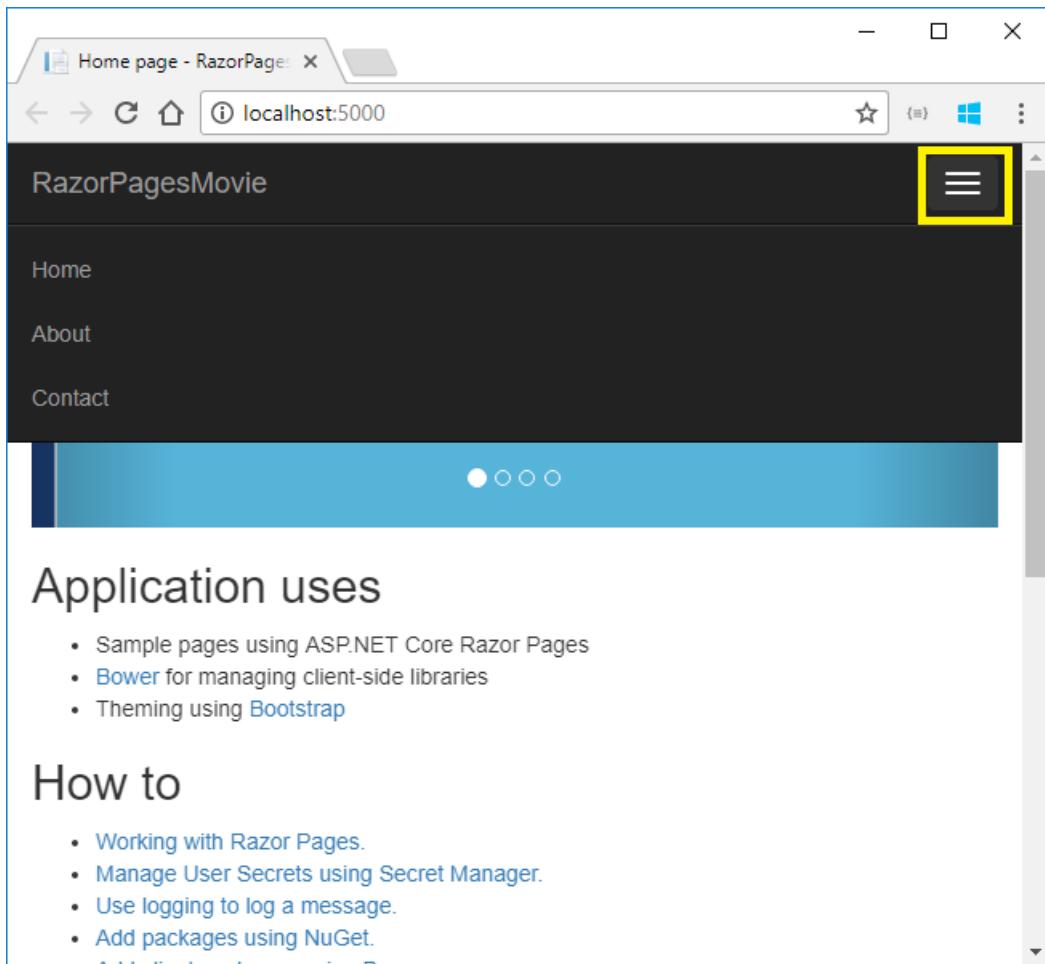
Application uses

- Sample pages using ASP.NET Core Razor Pages
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

[Working with Razor Pages](#)

默认模板创建“RazorPagesMovie”、“主页”、“关于”和“联系人”链接和页面。可能需要单击导航图标才能显示这些链接，具体取决于浏览器窗口的大小。



Application uses

- Sample pages using ASP.NET Core Razor Pages
- [Bower](#) for managing client-side libraries
- Theming using [Bootstrap](#)

How to

- [Working with Razor Pages.](#)
- [Manage User Secrets using Secret Manager.](#)
- [Use logging to log a message.](#)
- [Add packages using NuGet.](#)

测试链接。“RazorPagesMovie”和“主页”链接转到“索引”页。“关于”和“联系人”链接分别转到 [About](#) 和 [Contact](#) 页。面。

项目文件和文件夹

下表列出了项目中的文件和文件夹。对于本教程而言，Startup.cs 是最有必要了解的文件。无需查看下面提供的每一个链接。需要详细了解项目中的某个文件或文件夹时，可参考此处提供的链接。

文件或文件夹	目标
wwwroot	包含静态文件。请参阅 使用静态文件 。
页数	Razor Pages 的文件夹。
appsettings.json	配置
Program.cs	托管 ASP.NET Core 应用。
Startup.cs	配置服务和请求管道。请参阅 启动 。

“页面”文件夹

_Layout.cshtml 文件包含常见的 HTML 元素(脚本和样式表)，并设置应用程序的布局。例如，单击“RazorPagesMovie”、“主页”、“关于”或“联系人”时，将看到相同的元素。常见的元素包括顶部的导航菜单和窗口底部的标题。请参阅[布局](#)了解详细信息。

_ViewStart.cshtml 将 Razor 页面 [Layout](#) 属性设置为使用 _Layout.cshtml 文件。请参阅[布局](#)了解详细信息。

_ViewImports.cshtml 文件包含要导入每个 Razor 页面的 Razor 指令。请参阅[导入共享指令](#)了解详细信息。

_ValidationScriptsPartial.cshtml 文件提供对 [jQuery](#) 验证脚本的引用。在本教程的后续部分中添加 [Create](#) 和 [Edit](#) 页面时，将使用 _ValidationScriptsPartial.cshtml 文件。

[About](#)、[Contact](#) 和 [Index](#) 页面是基本页面，可用于启动应用。[Error](#) 页面用于显示错误信息。

打开项目

按 [Ctrl+C](#) 关闭应用程序。

在 Visual Studio 中，选择“文件”>“打开”，然后选择“RazorPagesMovie.csproj”文件。

启动应用

在 Visual Studio 中，选择“运行”>“开始执行(不调试)”以启动应用。Visual Studio 启动 [Kestrel](#)，启动浏览器并导航到 <http://localhost:5000>。

在下一个教程中，我们将向项目添加模型。

[下一篇：添加模型](#)

使用 Visual Studio for Mac 将模型添加到 ASP.NET Core Razor 页面应用

2018/5/14 • 5 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中将添加用于管理数据库中的电影的类。可以结合使用这些类和 [Entity Framework Core \(EF Core\)](#) 来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化必须要编写的数据访问代码。

要创建的模型类称为 POCO 类(源自“简单传统 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们定义数据库中存储的数据属性。

在本教程中，首先要编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍):[从现有数据库生成模型类](#)。

[查看或下载示例。](#)

添加数据模型

- 在解决方案资源管理器中，右键单击“RazorPagesMovie”项目，然后选择“添加” > “新建文件夹”。将文件夹命名为“Models”。
- 右键单击“Models”文件夹，然后选择“添加” > “新建文件”。
- 在“新建文件”对话框中：
 - 在左侧窗格中，选择“常规”。
 - 在中间窗格中，选择“空类”。
 - 将此类命名为“Movie”，然后选择“新建”。

向 `Movie` 类添加以下属性：

```
using System;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段作为主键。

添加数据库上下文类

将以下 `MovieContext.cs` 类添加到“模型”文件夹:[!code-csharp]

前面的代码为实体集创建 `DbSet` 属性。在实体框架术语中，实体集通常与数据库表相对应，实体与表中的行相对应。

添加数据库连接字符串

将连接字符串添加到 appsettings.json 文件。

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "ConnectionStrings": {  
        "MovieContext": "Data Source=MvcMovie.db"  
    }  
}
```

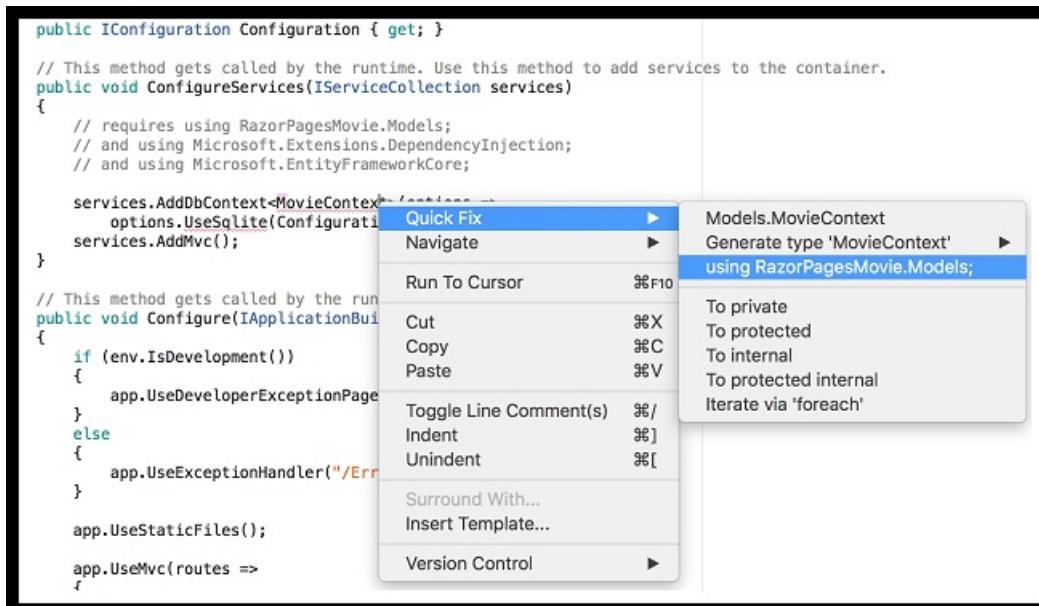
注册数据库上下文

使用 Startup.cs 文件中的[依存关系注入](#)容器注册数据库上下文。

```
public void ConfigureServices(IServiceCollection services)  
{  
    // requires  
    // using RazorPagesMovie.Models;  
    // using Microsoft.EntityFrameworkCore;  
  
    services.AddDbContext<MovieContext>(options =>  
        options.UseSqlite(Configuration.GetConnectionString("MovieContext")));  
    services.AddMvc();  
}
```

右键单击红色波浪线，例如，行 `services.AddDbContext<MovieContext>(options =>` 中的 `MovieContext`。选择“快速修复”>“使用 RazorPagesMovie.Models;”。Visual Studio 添加 using 语句。

生成项目以验证有没有任何错误存在。

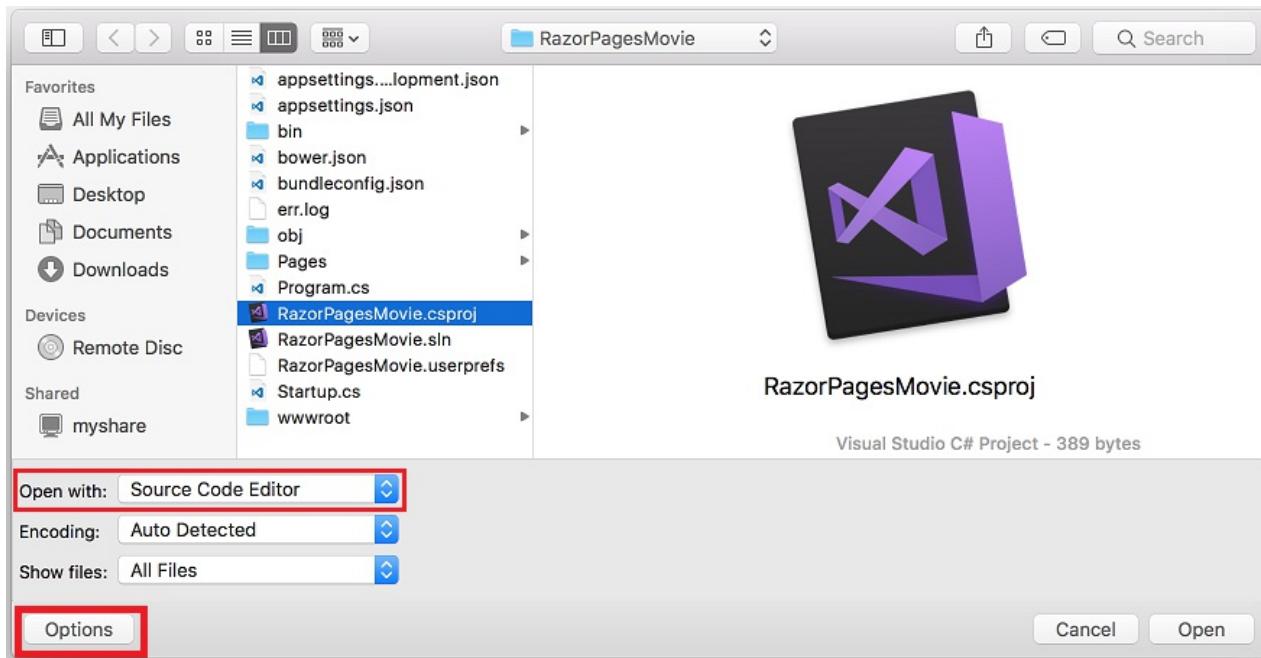


用于进行迁移的 Entity Framework Core NuGet 包

[Microsoft.EntityFrameworkCore.Tools.DotNet](#) 中提供了适用于命令行接口 (CLI) 的 EF 工具。单击 [Microsoft.EntityFrameworkCore.Tools.DotNet](#) 链接以获取要使用的版本号。若要安装此包，请将它添加到 .csproj 文件中的 `DotNetCliToolReference` 集合。注意：必须通过编辑 .csproj 文件来安装此包；不能使用 `install-package` 命令或程序包管理器 GUI。

若要编辑.csproj文件：

- 选择“文件”>“打开”，然后选择.csproj文件。
- 选择“选项”。
- 将“打开方式”更改为“源代码编辑器”。



将 Microsoft.EntityFrameworkCore.Tools.DotNet 工具引用添加至第二个 <ItemGroup>：

```
<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.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.2" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />
  </ItemGroup>
</Project>
```

以下代码中显示的版本号在编写时是正确的。

添加基架工具并执行初始迁移

从命令行运行以下 .NET Core CLI 命令：

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet restore
dotnet ef migrations add InitialCreate
dotnet ef database update
```

add package 命令安装运行基架引擎所需的工具。

ef migrations add InitialCreate 命令生成用于创建初始数据库架构的代码。此架构以（Models/MovieContext.cs 文件中的）DbContext 中指定的模型为基础。InitialCreate 参数用于为迁移命名。可以使用任意名称，但是按照惯例应选择描述迁移的名称。有关详细信息，请参阅[迁移简介](#)。

ef database update 命令在用于创建数据库的 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 Up 方法。

搭建“电影”模型的基架

- 从命令行(在包含 Program.cs、Startup.cs 和 .csproj 文件的项目目录中)中运行如下命令：

```
dotnet aspnet-codegenerator razorpage -m Movie -dc MovieContext -udl -outDir Pages/Movies --referenceScriptLibraries
```

如果收到错误：

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

打开一个到项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)的命令 shell。

如果收到错误：

```
The process cannot access the file  
'RazorPagesMovie/bin/Debug/netcoreapp2.0/RazorPagesMovie.dll'  
because it is being used by another process.
```

退出 Visual Studio, 然后重新运行命令。

下表详细说明了 ASP.NET Core 代码生成器的参数：

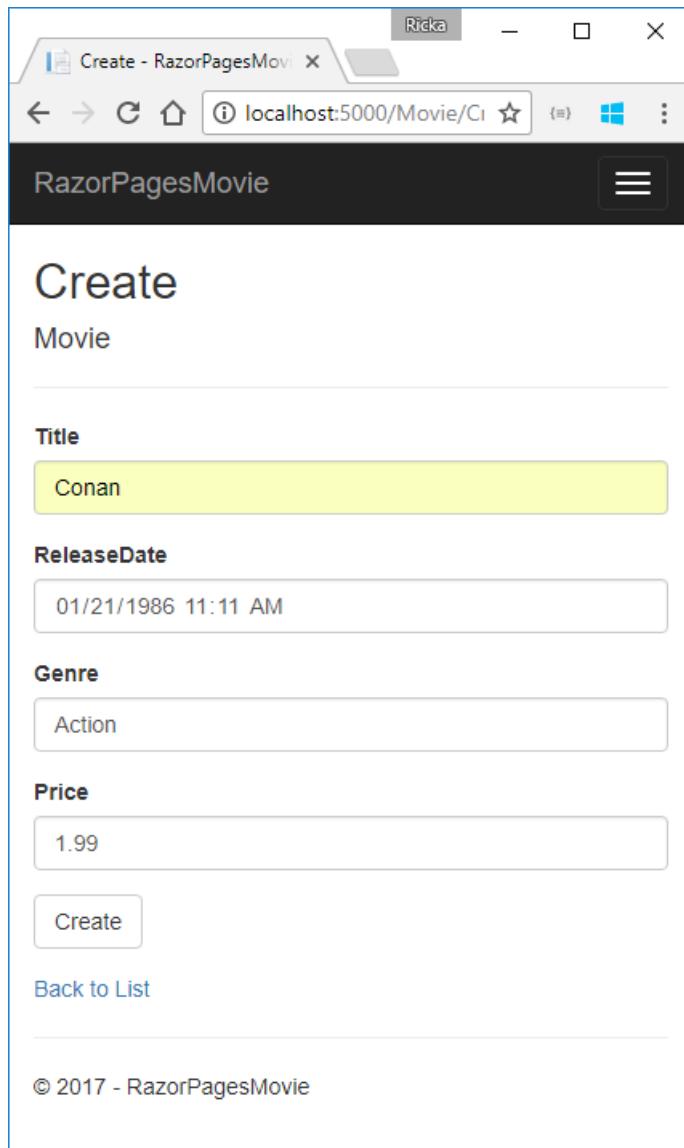
参数	描述
-m	模型的名称。
-dc	数据上下文。
-udl	使用默认布局。
-outDir	用于创建视图的相对输出文件夹路径。
--referenceScriptLibraries	向“编辑”和“创建”页面添加 <code>_ValidationScriptsPartial</code>

使用 `h` 开关获取 `aspnet-codegenerator razorpage` 命令方面的帮助：

```
dotnet aspnet-codegenerator razorpage -h
```

测试应用

- 运行应用并将 `/Movies` 追加到浏览器中的 URL (`http://localhost:port/movies`)。
- 测试“创建”链接。



- 测试“编辑”、“详细信息”和“删除”链接。

如果收到类似如下的错误，则验证是否已运行迁移并更新数据库：

```
An unhandled exception occurred while processing the request.  
'no such table: Movie'.
```

将页面/电影文件添加到项目

- 在 Visual Studio 中，右键单击“页面”文件夹，然后选择“添加”>“添加现有文件夹”。
- 选择“电影”文件夹。
- 在“选择要包含在项目中的文件”对话框中，选择“包括所有”。

下一个教程介绍由基架创建的文件。

[上一篇：入门](#)

[下一篇：已搭建基架的 RAZOR 页](#)

面

ASP.NET Core 中已搭建基架的 Razor 页面

2018/5/14 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍上一教程中通过搭建基架创建的 Razor 页面。

[查看或下载示例。](#)

“**创建**”、“**删除**”、“**详细信息**”和“**编辑**”页。

检查 Pages/Movies/Index.cshtml.cs 页面模型:[!code-csharp]

Razor 页面派生自 `PageModel`。按照约定, `PageModel` 派生的类称为 `<PageName>Model`。此构造函数使用[依赖关系注入](#)将 `MovieContext` 添加到页。所有已搭建基架的页面都遵循此模式。请参阅[异步代码](#), 了解有关使用实体框架的异步编程的详细信息。

对页面发出请求时, `OnGetAsync` 方法向 Razor 页面返回影片列表。在 Razor 页面上调用 `OnGetAsync` 或 `OnGet` 以初始化页面状态。在这种情况下, `OnGetAsync` 将获得影片列表并显示出来。

当 `OnGet` 返回 `void` 或 `OnGetAsync` 返回 `Task` 时, 不使用任何返回方法。当返回类型是 `IActionResult` 或 `Task<IActionResult>` 时, 必须提供返回语句。例如, Pages/Movies/Create.cshtml.cs `OnPostAsync` 方法:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

检查 Pages/Movies/Index.cshtml Razor 页面:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
    </tbody>
</table>

```

Razor 可以从 HTML 转换为 C# 或 Razor 特定标记。当 `@` 符号后跟 **Razor 保留关键字** 时，它会转换为 Razor 特定标记，否则会转换为 C#。

`@page` Razor 指令将文件转换为一个 MVC 操作 —，这意味着它可以处理请求。`@page` 必须是页面上的第一个 Razor 指令。`@page` 是转换到 Razor 特定标记的一个示例。有关详细信息，请参阅 [Razor 语法](#)。

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.Movie[0].Title))
```

`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。检查 Lambda 表达式(而非求值)。这意味着当 `model`、`model.Movie` 或 `model.Movie[0]` 为 `null` 或为空时, 不会存在任何访问冲突。对 Lambda 表达式求值时(例如, 使用 `@Html.DisplayNameFor(modelItem => item.Title)`) , 将求得该模型的属性值。

@model 指令

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel
```

`@model` 指令指定传递给 Razor 页面的模型类型。在前面的示例中, `@model` 行使 `PageModel` 派生的类可用于 Razor 页面。在页面上的 `@Html.DisplayNameFor` 和 `@Html.DisplayName` HTML 帮助程序中使用该模型。

ViewData 和布局

考虑下列代码:

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel  
  
{@  
    ViewData["Title"] = "Index";  
}
```

前面突出显示的代码是 Razor 转换为 C# 的一个示例。`{` 和 `}` 字符括住 C# 代码块。

`PageModel` 基类具有 `ViewData` 字典属性, 可用于添加要传递到某个视图的数据。可以使用键/值模式将对象添加到 `ViewData` 字典。在前面的示例中, “Title”属性被添加到 `ViewData` 字典。“Title”属性用于 Pages/_Layout.cshtml 文件。以下标记显示 Pages/_Layout.cshtml 文件的前几行。

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <title>@ViewData["Title"] - RazorPagesMovie</title>  
  
    {*Markup removed for brevity.*@
```

行 `{*Markup removed for brevity.*@` 为 Razor 注释。与 HTML 注释不同 (`<!-- -->`), Razor 注释不会发送到客户端。

运行应用并测试项目中的链接(“主页”、“关于”、“联系人”、“创建”、“编辑”和“删除”)。每个页面都设置有标题, 可以在浏览器选项卡中看到标题。将某个页面加入书签时, 标题用于该书签。Pages/Index.cshtml 和 Pages/Movies/Index.cshtml 当前具有相同的标题, 但可以修改它们以具有不同的值。

在 Pages/_ViewStart.cshtml 文件中设置 `Layout` 属性:

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

前面的标记针对所有 Razor 文件将布局文件设置为 Pages 文件夹下的 Pages/_Layout.cshtml。请参阅[布局](#)了解详细信息。

更新布局

更改 Pages/_Layout.cshtml 文件中的 `<title>` 元素以使用较短的字符串。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Movie</title>
```

查找 Pages/_Layout.cshtml 文件中的以下定位点元素。

```
<a asp-page="/Index" class="navbar-brand">RazorPagesMovie</a>
```

将前面的元素替换为以下标记。

```
<a asp-page="/Movies/Index" class="navbar-brand">RpMovie</a>
```

前面的定位点元素是一个[标记帮助程序](#)。此处它是[定位点标记帮助程序](#)。`asp-page="/Movies/Index"` 标记帮助程序属性和值可以创建指向 `/Movies/Index` Razor 页面的链接。

保存所做的更改，并通过单击“RpMovie”链接测试应用。请参阅 GitHub 中的 [_Layout.cshtml](#) 文件。

“创建”页面模型

检查 Pages/Movies/Create.cshtml.cs 页面模型：

```

// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public CreateModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

`OnGet` 方法初始化页面所需的任何状态。“创建”页没有任何要初始化的状态。`Page` 方法创建用于呈现 `Create.cshtml` 页的 `PageResult` 对象。

`Movie` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。当“创建”表单发布表单值时，ASP.NET Core 运行时将发布的值绑定到 `Movie` 模型。

当页面发布表单数据时，运行 `OnPostAsync` 方法：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

如果不存在任何模型错误，将重新显示表单，以及发布的任何表单数据。在发布表单前，可以在客户端捕获到大部分模型错误。模型错误的一个示例是，发布的日期字段值无法转换为日期。我们将在本教程后面的内容中讨论有关

客户端验证和模型验证的更多信息。

如果不存在模型错误，将保存数据，并且浏览器会重定向到索引页。

创建 Razor 页面

检查 Pages/Movies/Create.cshtml Razor 页面文件：

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
```

<form method="post"> 元素是一个表单标记帮助程序。表单标记帮助程序会自动包含防伪令牌。

基架引擎在模型中为每个字段(ID 除外)创建 Razor 标记，如下所示：

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

验证标记帮助程序 (`<div asp-validation-summary>` 和 ``) 显示验证错误。本系列后面的部分将更详细地讨论有关验证的信息。

标签标记帮助程序 (`<label asp-for="Movie.Title" class="control-label"></label>`) 生成标签描述和 `Title` 属性的 `for` 特性。

输入标记帮助程序 (`<input asp-for="Movie.Title" class="form-control" />`) 使用 `DataAnnotations` 属性并在客户端生成 jQuery 验证所需的 HTML 属性。

下一教程将介绍 SQLite 和数据库的种子设定。

[上一篇：添加模型](#)

[下一篇：SQLITE](#)

使用 SQLite 和 Razor 页面

2018/5/8 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

`MovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using RazorPagesMovie.Models;
    // using Microsoft.EntityFrameworkCore;

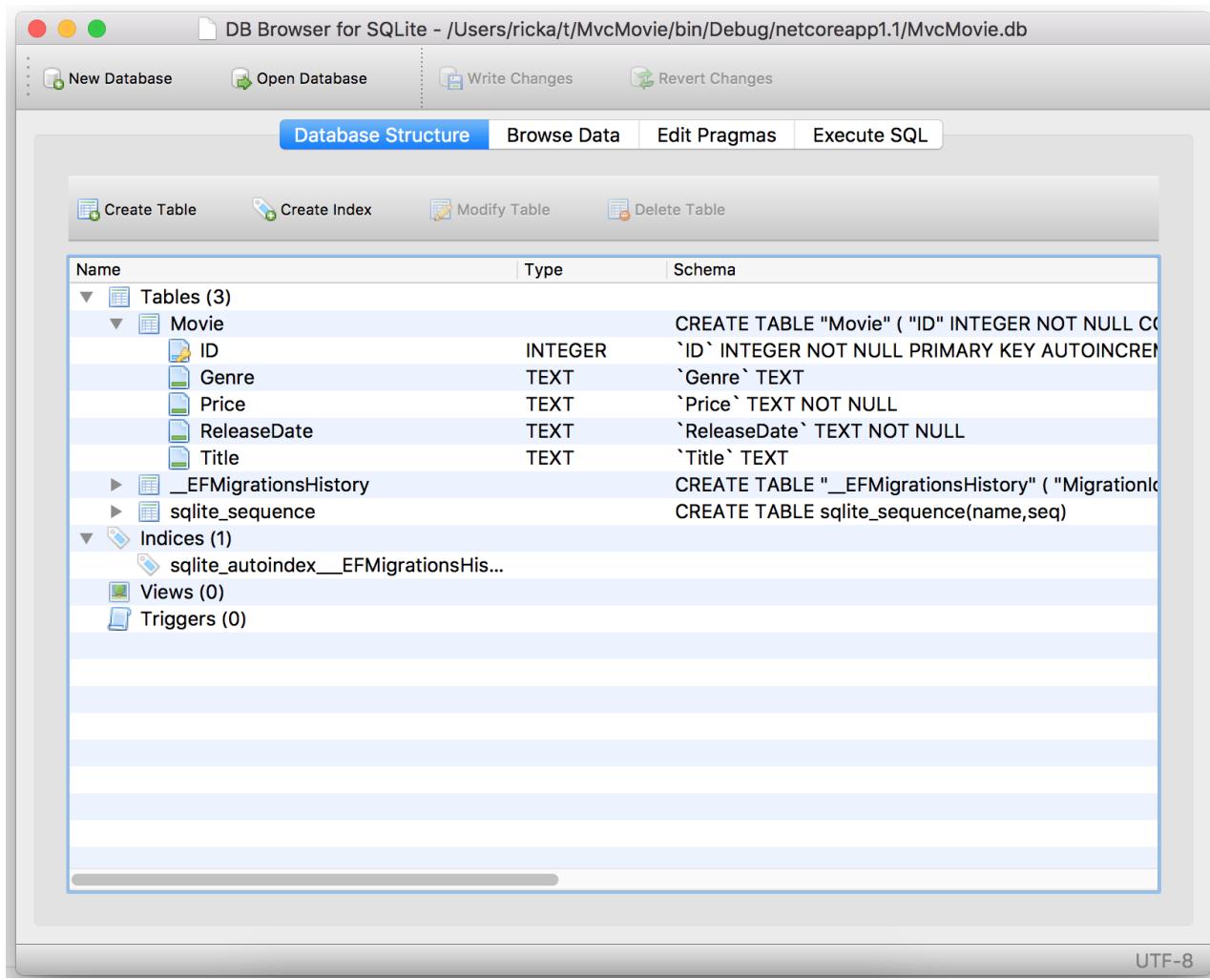
    services.AddDbContext<MovieContext>(options =>
        options.UseSqlite(Configuration.GetConnectionString("MovieContext")));
    services.AddMvc();
}
```

SQLite

[SQLite](#) 网站上表示:

SQLite 是一个自包含、高可靠性、嵌入式、功能完整、公共域的 SQL 数据库引擎。SQLite 是世界上使用最多的数据引擎。

可以下载许多第三方工具来管理并查看 SQLite 数据库。下面的图片来自 [DB Browser for SQLite](#)。如果你有最喜欢的 SQLite 工具, 请发表评论以分享你喜欢的方面。



设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}

```

如果 DB 中没有任何电影，则会返回种子初始值设定项。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法：

```
// Unused usings removed.
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<MovieContext>();
                    // requires using Microsoft.EntityFrameworkCore;
                    context.Database.Migrate();
                    // Requires using RazorPagesMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

删除 DB 中的所有记录(使种子方法运行)。停止并启动应用以设定数据库种子。

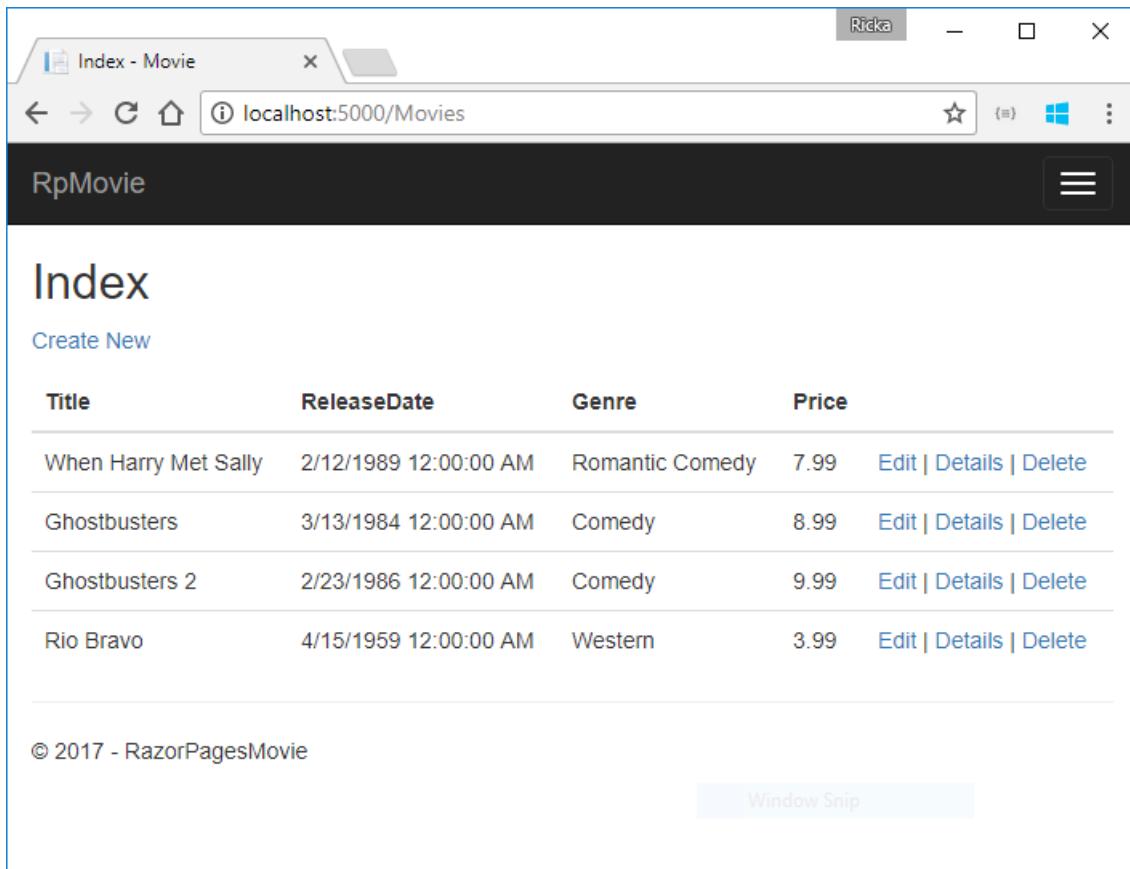
应用将显示设定为种子的数据。

更新生成的页面

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(如下图所示的 12:00:00 AM), 并且“ReleaseDate”应为“Release Date”(两个词)。



The screenshot shows a Microsoft Edge browser window titled "Index - Movie". The address bar displays "localhost:5000/Movies". The main content area is titled "RpMovie" and contains the word "Index". Below it is a "Create New" button. A table lists four movies:

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	2/12/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

At the bottom of the page, there is a copyright notice: "© 2017 - RazorPagesMovie".

更新生成的代码

打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

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

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

我们将在下一教程中介绍 DataAnnotations。Display 特性指定要显示的字段名称的内容(本例中应为“Release Date”，而不是“ReleaseDate”)。DataType 属性指定数据的类型(日期)，使字段中存储的时间信息不会显示。

浏览到 Pages/Movies，并将鼠标悬停在“编辑”链接上以查看目标 URL。

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959	Western	3.99	Edit Details Delete

© 2017 - RazorPagesMovie

localhost:5000/Movies/Edit?id=2

“编辑”、“详细信息”和“删除”链接是在 Pages/Movies/Index.cshtml 文件中由[定位标记帮助程序](#)生成的。

```
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
```

[标记帮助程序](#)使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在前面的代码中，

`AnchorTagHelper` 从 Razor 页面(路由是相对的)、`asp-page` 和路由 ID (`asp-route-id`) 动态生成 HTML `href` 特性值。有关详细信息，请参阅[页面的 URL 生成](#)。

在最喜欢的浏览器中使用“查看源”来检查生成的标记。生成的 HTML 的一部分如下所示：

```
<td>
    <a href="/Movies/Edit?id=1">Edit</a> |
    <a href="/Movies/Details?id=1">Details</a> |
    <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

动态生成的链接通过查询字符串传递电影 ID(例如 `http://localhost:5000/Movies/Details?id=2`)。

更新“编辑”、“详细信息”和“删除”Razor 页面以使用“`{id:int?}`”路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int?}"`。运行应用，然后查看源。生成的 HTML 会将 ID 添加到 URL 的路径部分：

```
<td>
    <a href="/Movies/Edit/1">Edit</a> |
    <a href="/Movies/Details/1">Details</a> |
    <a href="/Movies/Delete/1">Delete</a>
</td>
```

如果对具有“`{id:int?}`”路由模板的页面进行的请求中不包含整数，则将返回 HTTP 404(未找到)错误。例如，

`http://localhost:5000/Movies/Details` 将返回 404 错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

更新并发异常处理

在 Pages/Movies/Edit.cshtml.cs 文件中更新 `OnPostAsync` 方法。下列突出显示的代码显示了更改：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!(_context.Movie.Any(e => e.ID == Movie.ID)))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
}

return RedirectToPage("./Index");
}
```

之前的代码仅检测当第一个并发客户端删除电影以及第二个并发客户端对电影发布更改时的并发异常。

测试 `catch` 块：

- 在 `catch (DbUpdateConcurrencyException)` 上设置断点
- 编辑电影。

- 在其他浏览器窗口中，选择同一电影的“删除”链接，然后删除此电影。
- 在之前的浏览器窗口中，将更改发布到电影。

当两个或更多客户端同时更新记录时，生产代码通常将检测到并发冲突。有关详细信息，请参阅[处理并发冲突](#)。

发布和绑定审阅

检查 `Pages/Movies/Edit.cshtml.cs` 文件：[!code-csharp]

当对 `Movies/Edit` 页面进行 HTTP GET 请求时（例如 `http://localhost:5000/Movies/Edit/2`）：

- `OnGetAsync` 方法从数据库提取电影并返回 `Page` 方法。
- `Page` 方法呈现“`Pages/Movies/Edit.cshtml`”Razor 页面。`Pages/Movies/Edit.cshtml` 文件包含模型指令（`@model RazorPagesMovie.Pages.Movies.EditModel`），这使电影模型在页面上可用。
- “编辑”表单中会显示电影的值。

当发布 `Movies/Edit` 页面时：

- 此页面上的表单值将绑定到 `Movie` 属性。`[BindProperty]` 特性会启用[模型绑定](#)。

```
[BindProperty]  
public Movie Movie { get; set; }
```

- 如果模型状态中存在错误（例如，`ReleaseDate` 无法被转换为日期），则会使用已提交的值再次发布表单。
- 如果没有模型错误，则电影已保存。

“索引”、“创建”和“删除”Razor 页面中的 HTTP GET 方法遵循一个类似的模式。“创建”Razor 页面中的 HTTP POST `OnPostAsync` 方法遵循的模式类似于“编辑”Razor 页面中的 `OnPostAsync` 方法所遵循的模式。

在下一教程中将添加搜索。

上一步：使用

添加搜索

SQLITE

将搜索添加到 Razor 页面应用

2018/5/8 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本文档中，将向索引页面添加搜索功能以实现按“流派”或“名称”搜索电影。

使用以下代码更新索引页面的 `OnGetAsync` 方法：

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

```
public async Task OnGetAsync(string searchString)  
{  
    var movies = from m in _context.Movie  
                 select m;  
  
    if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
  
    Movie = await movies.ToListAsync();  
}
```

`OnGetAsync` 方法的第一行创建了 [LINQ](#) 查询用于选择电影：

```
var movies = from m in _context.Movie  
             select m;
```

此时仅对查询进行了定义，它还不会针对数据库运行。

如果 `searchString` 参数包含一个字符串，电影查询则会被修改为根据搜索字符串进行筛选：

```
if (!String.IsNullOrEmpty(searchString))  
{  
    movies = movies.Where(s => s.Title.Contains(searchString));  
}
```

`s => s.Title.Contains()` 代码是 [Lambda 表达式](#)。Lambda 在基于方法的 [LINQ](#) 查询中用作标准查询运算符方法的参数，如 [Where](#) 方法或 `Contains`（前面的代码中所使用的）。在对 LINQ 查询进行定义或通过调用方法（如 `Where`、`Contains` 或 `OrderBy`）进行修改后，此查询不会被执行。相反，会延迟执行查询。这意味着表达式的计算会延迟，直到循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关详细信息，请参阅 [Query Execution](#)（查询执行）。

注意：`Contains` 方法在数据库中运行，而不是在 C# 代码中。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上，`Contains` 映射到 [SQL LIKE](#)，这是不区分大小写的。在 SQLite 中，由于使用了默认排序规则，因此需要区分大小写。

导航到电影页面，并向 URL 追加一个如 `?searchString=Ghost` 的查询字符串（例如 `http://localhost:5000/Movies?searchString=Ghost`）。筛选的电影将显示出来。

Index

Create New

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

© 2017 - RazorPagesMovie

如果向索引页面添加了以下路由模板，搜索字符串则可作为 URL 段传递（例如 `http://localhost:5000/Movies/ghost`）。

```
@page "{searchString?}"
```

前面的路由约束允许按路由数据（URL 段）搜索标题，而不是按查询字符串值进行搜索。`"{searchString?}"` 中的 `?` 表示这是可选路由参数。

Index

Create New

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

© 2017 - RazorPagesMovie

但是，不能指望用户修改 URL 来搜索电影。在此步骤中，会添加 UI 来筛选电影。如果已添加路由约束 `"{searchString?}"`，请将它删除。

打开 Pages/Movies/Index.cshtml 文件，并添加以下代码中突出显示的 `<form>` 标记：

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@

```

HTML `<form>` 标记使用[表单标记帮助程序](#)。提交表单时，筛选器字符串将发送到 Pages/Movies/Index 页面。保存更改并测试筛选器。

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

按流派搜索

将以下突出显示的属性添加到 Pages/Movies/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.MovieContext _context;

    public IndexModel(RazorPagesMovie.Models.MovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    public SelectList Genres { get; set; }
    public string MovieGenre { get; set; }
```

`SelectList Genres` 包含流派列表。这使用户能够从列表中选择一种流派。

`MovieGenre` 属性包含用户选择的特定流派(例如“西部”)。

使用以下代码更新 `OnGetAsync` 方法：

```
// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task OnGetAsync(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

下面的代码是一种 LINQ 查询，可从数据库中检索所有流派。

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;
```

流派的 `SelectList` 是通过投影截然不同的流派创建的。

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

添加“按流派搜索”

更新 `Index.cshtml`, 如下所示：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

通过按流派或/和电影标题搜索来测试应用。

[上一篇：更新页面](#) [下一篇：添加新字段](#)

使用 ASP.NET Core 和 Visual Studio Code 创建 Razor 页面 Web 应用

2018/4/27 • 1 min to read • [Edit Online](#)

我们正在撰写此系列的文章。

此系列介绍了使用 Visual Studio Code 在 ASP.NET Core 中生成 Razor 页面 Web 应用的基础知识。

1. [通过 VS Code 开始使用 Razor 页面](#)
2. [向 Razor 页面应用添加模型](#)
3. [已搭建基架的 Razor 页面](#)
4. [使用 SQLite](#)
5. [更新页面](#)
6. [添加搜索](#)

在后续部分完成前, 请先按照 Visual Studio for Windows 版本操作。

1. [添加新字段](#)
2. [添加验证](#)

在 Visual Studio Code 中开始使用 ASP.NET Core Razor 页面

2018/5/17 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍构建 ASP.NET Core Razor 页面 Web 应用的基础知识。我们建议在学习本教程前先阅读 [Razor 页面介绍](#)。Razor 页面是在 ASP.NET Core 中为 Web 应用程序生成 UI 时建议使用的方法。

系统必备

Install the following:

- [.NET Core SDK 2.0 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)
- [.NET Core SDK 2.1 RC1 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)

创建 Razor Web 应用

从终端运行以下命令：

```
dotnet new razor -o RazorPagesMovie  
cd RazorPagesMovie  
dotnet run
```

上述命令使用 [.NET Core CLI](#) 创建并运行 Razor 页面项目。打开浏览器，转到 <http://localhost:5000> 查看应用程序。

The screenshot shows a web browser window with the title "Home page - RazorPages" and the URL "localhost:5000". The page has a dark header with the text "RazorPagesMovie" and navigation links for "Home", "About", and "Contact". Below the header is a large blue banner with the text "ASP.NET Core" and "Windows", "Linux", "OSX". A central call-to-action button says "Learn how to build ASP.NET apps that can run anywhere." with a "Learn More" link. At the bottom of the banner is a navigation bar with four items: a left arrow, a right arrow, and two circular icons. The main content area below the banner contains sections for "Application uses" and "How to".

Application uses

- Sample pages using ASP.NET Core Razor Pages
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

- Working with Razor Pages

默认模板创建“RazorPagesMovie”、“主页”、“关于”和“联系人”链接和页面。可能需要单击导航图标才能显示这些链接，具体取决于浏览器窗口的大小。

The screenshot shows a web browser window with the title "Home page - RazorPages" and the URL "localhost:5000". The page has a dark header with the text "RazorPagesMovie" and navigation links for "Home", "About", and "Contact". A yellow box highlights the three-line navigation icon in the top right corner of the header. Below the header is a large blue banner with the text "ASP.NET Core" and "Windows", "Linux", "OSX". A central call-to-action button says "Learn how to build ASP.NET apps that can run anywhere." with a "Learn More" link. At the bottom of the banner is a navigation bar with four items: a left arrow, a right arrow, and two circular icons. The main content area below the banner contains sections for "Application uses" and "How to".

Application uses

- Sample pages using ASP.NET Core Razor Pages
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

- Working with Razor Pages.
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet.

测试链接。“RazorPagesMovie”和“主页”链接转到“索引”页。“关于”和“联系人”链接分别转到 [About](#) 和 [Contact](#) 页

面。

项目文件和文件夹

下表列出了项目中的文件和文件夹。对于本教程而言，Startup.cs 是最有必要了解的文件。无需查看下面提供的每一个链接。需要详细了解项目中的某个文件或文件夹时，可参考此处提供的链接。

文件或文件夹	目标
wwwroot	包含静态文件。请参阅 使用静态文件 。
页数	Razor Pages 的文件夹。
appsettings.json	配置
Program.cs	托管 ASP.NET Core 应用。
Startup.cs	配置服务和请求管道。请参阅 启动 。

“页面”文件夹

_Layout.cshtml 文件包含常见的 HTML 元素(脚本和样式表)，并设置应用程序的布局。例如，单击“RazorPagesMovie”、“主页”、“关于”或“联系人”时，将看到相同的元素。常见的元素包括顶部的导航菜单和窗口底部的标题。请参阅[布局](#)了解详细信息。

_ViewStart.cshtml 将 Razor 页面 `Layout` 属性设置为使用 _Layout.cshtml 文件。请参阅[布局](#)了解详细信息。

_ViewImports.cshtml 文件包含要导入每个 Razor 页面的 Razor 指令。请参阅[导入共享指令](#)了解详细信息。

_ValidationScriptsPartial.cshtml 文件提供对 [jQuery](#) 验证脚本的引用。在本教程的后续部分中添加 `Create` 和 `Edit` 页面时，将使用 _ValidationScriptsPartial.cshtml 文件。

`About`、`Contact` 和 `Index` 页面是基本页面，可用于启动应用。`Error` 页面用于显示错误信息。

打开项目

按 `Ctrl+C` 关闭应用程序。

在 Visual Studio Code (VS Code) 中，选择“文件”>“打开文件夹”，然后选择“RazorPagesMovie”文件夹。

- 对警告消息““RazorPagesMovie”中缺少进行生成和调试所需的资产。是否添加它们？”选择“是”。
- 对信息性消息“存在未解析的依赖项”选择“还原”。

启动应用

按 `Ctrl+F5` 启动应用而不进行调试。或者，从“调试”菜单中选择“开始执行(不调试)”。

在下一个教程中，我们将向项目添加模型。

[下一篇：添加模型](#)

使用 Visual Studio Code 将模型添加到 ASP.NET Core Razor 页面应用

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中将添加用于管理数据库中的电影的类。可以结合使用这些类和 [Entity Framework Core \(EF Core\)](#) 来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化必须要编写的数据访问代码。

要创建的模型类称为 POCO 类(源自“简单传统 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们定义数据库中存储的数据属性。

在本教程中，首先要编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍):[从现有数据库生成模型类](#)。

[查看或下载示例。](#)

添加数据模型

- 添加名为“Models”的文件夹。
- 将类添加到名为“Movie.cs”的“Models”文件夹。
- 将以下代码添加到“Models/Movie.cs”文件：

向 `Movie` 类添加以下属性：

```
using System;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段作为主键。

添加数据库上下文类

将以下 `MovieContext.cs` 类添加到“模型”文件夹:[!code-csharp]

前面的代码为实体集创建 `DbSet` 属性。在实体框架术语中，实体集通常与数据库表相对应，实体与表中的行相对应。

添加数据库连接字符串

将连接字符串添加到 `appsettings.json` 文件。

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "ConnectionStrings": {  
        "MovieContext": "Data Source=MvcMovie.db"  
    }  
}
```

注册数据库上下文

使用 Startup.cs 文件中的[依存关系注入](#)容器注册数据库上下文。

```
public void ConfigureServices(IServiceCollection services)  
{  
    // requires  
    // using RazorPagesMovie.Models;  
    // using Microsoft.EntityFrameworkCore;  
  
    services.AddDbContext<MovieContext>(options =>  
        options.UseSqlite(Configuration.GetConnectionString("MovieContext")));  
    services.AddMvc();  
}
```

生成项目以确定没有任何错误。

用于进行迁移的 Entity Framework Core NuGet 包

Microsoft.EntityFrameworkCore.Tools.DotNet 中提供了适用于命令行接口 (CLI) 的 EF 工具。若要安装此包, 请将它添加到 .csproj 文件中的 `DotNetCliToolReference` 集合。注意: 必须通过编辑 .csproj 文件来安装此包; 不能使用 `install-package` 命令或程序包管理器 GUI。

编辑 RazorPagesMovie.csproj 文件:

- 选择“文件”>“打开文件”, 然后选择 RazorPagesMovie.csproj 文件。
- 将 `Microsoft.EntityFrameworkCore.Tools.DotNet` 的工具引用添加至第二个 `<ItemGroup>`:

```
<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.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.2" />  
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.1" />  
  </ItemGroup>  
</Project>
```

添加基架工具并执行初始迁移

从命令行运行以下 .NET Core CLI 命令:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design  
dotnet restore  
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

`add package` 命令安装运行基架引擎所需的工具。

`ef migrations add InitialCreate` 命令生成用于创建初始数据库架构的代码。此架构以（`Models/MovieContext.cs` 文件中的）`DbContext` 中指定的模型为基础。`InitialCreate` 参数用于为迁移命名。可以使用任意名称，但是按照惯例应选择描述迁移的名称。有关详细信息，请参阅[迁移简介](#)。

`ef database update` 命令在用于创建数据库的 `Migrations/<time-stamp>_InitialCreate.cs` 文件中运行 `Up` 方法。

搭建“电影”模型的基架

- 打开项目目录（包含 `Program.cs`、`Startup.cs` 和 `.csproj` 文件的目录）中的命令窗口。
- 运行下面的命令：

注意：请在 **Windows** 上运行以下命令。对于 **MacOS** 和 **Linux**，请参阅下一个命令

```
dotnet aspnet-codegenerator razorpage -m Movie -dc MovieContext -udl -outDir Pages\Movies --  
referenceScriptLibraries
```

- 在 **MacOS** 和 **Linux** 上，请运行以下命令：

```
dotnet aspnet-codegenerator razorpage -m Movie -dc MovieContext -udl -outDir Pages/Movies --  
referenceScriptLibraries
```

如果收到错误：

```
The process cannot access the file  
'RazorPagesMovie/bin/Debug/netcoreapp2.0/RazorPagesMovie.dll'  
because it is being used by another process.
```

退出 Visual Studio，然后重新运行命令。

下表详细说明了 ASP.NET Core 代码生成器的参数：

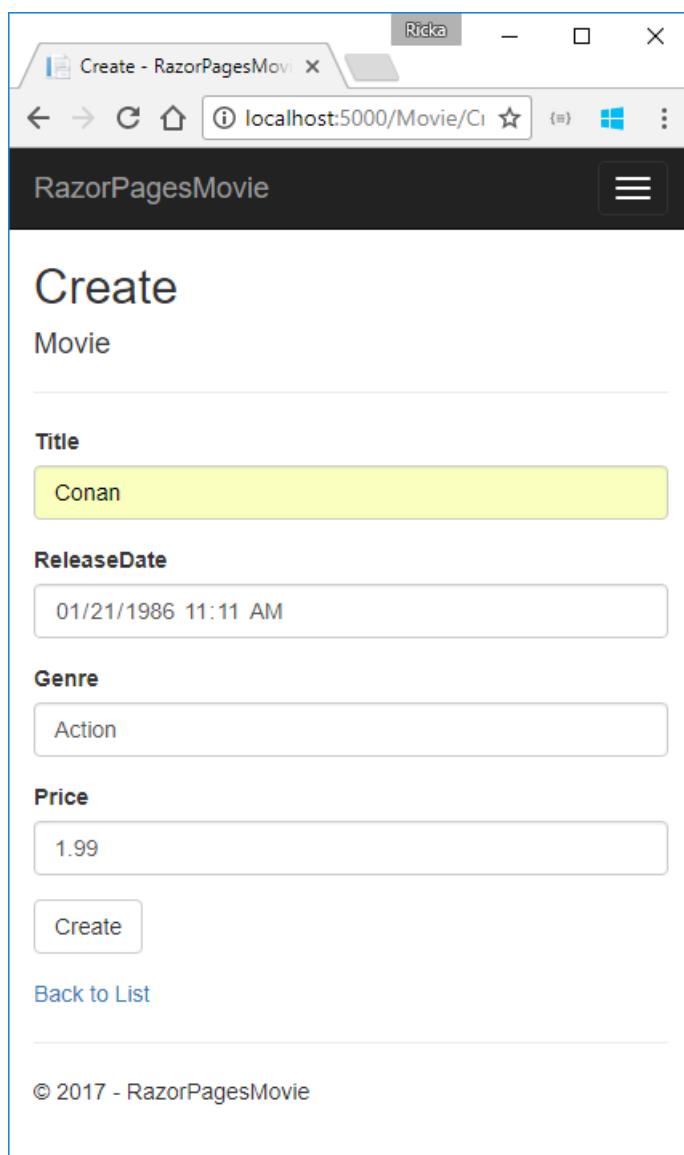
参数	描述
<code>-m</code>	模型的名称。
<code>-dc</code>	数据上下文。
<code>-udl</code>	使用默认布局。
<code>-outDir</code>	用于创建视图的相对输出文件夹路径。
<code>--referenceScriptLibraries</code>	向“编辑”和“创建”页面添加 <code>_ValidationScriptsPartial</code>

使用 `h` 开关获取 `aspnet-codegenerator razorpage` 命令方面的帮助：

```
dotnet aspnet-codegenerator razorpage -h
```

测试应用

- 运行应用并将 `/Movies` 追加到浏览器中的 URL (`http://localhost:port/movies`)。
- 测试“创建”链接。



- 测试“编辑”、“详细信息”和“删除”链接。

如果收到类似如下的错误，则验证是否已运行迁移并更新数据库：

```
An unhandled exception occurred while processing the request.  
'no such table: Movie'.
```

下一个教程介绍由基架创建的文件。

[上一篇：入门](#)

[下一篇：已搭建基架的 RAZOR 页](#)

ASP.NET Core 中已搭建基架的 Razor 页面

2018/5/14 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程介绍上一教程中通过搭建基架创建的 Razor 页面。

[查看或下载示例。](#)

“**创建**”、“**删除**”、“**详细信息**”和“**编辑**”页。

检查 Pages/Movies/Index.cshtml.cs 页面模型:[!code-csharp]

Razor 页面派生自 `PageModel`。按照约定, `PageModel` 派生的类称为 `<PageName>Model`。此构造函数使用[依赖关系注入](#)将 `MovieContext` 添加到页。所有已搭建基架的页面都遵循此模式。请参阅[异步代码](#), 了解有关使用实体框架的异步编程的详细信息。

对页面发出请求时, `OnGetAsync` 方法向 Razor 页面返回影片列表。在 Razor 页面上调用 `OnGetAsync` 或 `OnGet` 以初始化页面状态。在这种情况下, `OnGetAsync` 将获得影片列表并显示出来。

当 `OnGet` 返回 `void` 或 `OnGetAsync` 返回 `Task` 时, 不使用任何返回方法。当返回类型是 `IActionResult` 或 `Task<IActionResult>` 时, 必须提供返回语句。例如, Pages/Movies/Create.cshtml.cs `OnPostAsync` 方法:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

检查 Pages/Movies/Index.cshtml Razor 页面:

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
    </tbody>
</table>

```

Razor 可以从 HTML 转换为 C# 或 Razor 特定标记。当 `@` 符号后跟 **Razor 保留关键字** 时，它会转换为 Razor 特定标记，否则会转换为 C#。

`@page` Razor 指令将文件转换为一个 MVC 操作 —，这意味着它可以处理请求。`@page` 必须是页面上的第一个 Razor 指令。`@page` 是转换到 Razor 特定标记的一个示例。有关详细信息，请参阅 [Razor 语法](#)。

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.Movie[0].Title))
```

`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。检查 Lambda 表达式(而非求值)。这意味着当 `model`、`model.Movie` 或 `model.Movie[0]` 为 `null` 或为空时, 不会存在任何访问冲突。对 Lambda 表达式求值时(例如, 使用 `@Html.DisplayNameFor(modelItem => item.Title)`) , 将求得该模型的属性值。

@model 指令

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel
```

`@model` 指令指定传递给 Razor 页面的模型类型。在前面的示例中, `@model` 行使 `PageModel` 派生的类可用于 Razor 页面。在页面上的 `@Html.DisplayNameFor` 和 `@Html.DisplayName` HTML 帮助程序中使用该模型。

ViewData 和布局

考虑下列代码:

```
@page  
@model RazorPagesMovie.Pages.Movies.IndexModel  
  
{@  
    ViewData["Title"] = "Index";  
}
```

前面突出显示的代码是 Razor 转换为 C# 的一个示例。`{` 和 `}` 字符括住 C# 代码块。

`PageModel` 基类具有 `ViewData` 字典属性, 可用于添加要传递到某个视图的数据。可以使用键/值模式将对象添加到 `ViewData` 字典。在前面的示例中, “Title”属性被添加到 `ViewData` 字典。“Title”属性用于 Pages/_Layout.cshtml 文件。以下标记显示 Pages/_Layout.cshtml 文件的前几行。

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
    <title>@ViewData["Title"] - RazorPagesMovie</title>  
  
    {*Markup removed for brevity.*@
```

行 `*Markup removed for brevity.*@` 为 Razor 注释。与 HTML 注释不同 (`<!-- -->`), Razor 注释不会发送到客户端。

运行应用并测试项目中的链接(“主页”、“关于”、“联系人”、“创建”、“编辑”和“删除”)。每个页面都设置有标题, 可以在浏览器选项卡中看到标题。将某个页面加入书签时, 标题用于该书签。Pages/Index.cshtml 和 Pages/Movies/Index.cshtml 当前具有相同的标题, 但可以修改它们以具有不同的值。

在 Pages/_ViewStart.cshtml 文件中设置 `Layout` 属性:

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

前面的标记针对所有 Razor 文件将布局文件设置为 Pages 文件夹下的 Pages/_Layout.cshtml。请参阅[布局](#)了解详细信息。

更新布局

更改 Pages/_Layout.cshtml 文件中的 `<title>` 元素以使用较短的字符串。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - Movie</title>
```

查找 Pages/_Layout.cshtml 文件中的以下定位点元素。

```
<a asp-page="/Index" class="navbar-brand">RazorPagesMovie</a>
```

将前面的元素替换为以下标记。

```
<a asp-page="/Movies/Index" class="navbar-brand">RpMovie</a>
```

前面的定位点元素是一个[标记帮助程序](#)。此处它是[定位点标记帮助程序](#)。`asp-page="/Movies/Index"` 标记帮助程序属性和值可以创建指向 `/Movies/Index` Razor 页面的链接。

保存所做的更改，并通过单击“RpMovie”链接测试应用。请参阅 GitHub 中的 [_Layout.cshtml](#) 文件。

“创建”页面模型

检查 Pages/Movies/Create.cshtml.cs 页面模型：

```

// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.MovieContext _context;

        public CreateModel(RazorPagesMovie.Models.MovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}

```

`OnGet` 方法初始化页面所需的任何状态。“创建”页没有任何要初始化的状态。`Page` 方法创建用于呈现 `Create.cshtml` 页的 `PageResult` 对象。

`Movie` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。当“创建”表单发布表单值时，ASP.NET Core 运行时将发布的值绑定到 `Movie` 模型。

当页面发布表单数据时，运行 `OnPostAsync` 方法：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

```

如果不存在任何模型错误，将重新显示表单，以及发布的任何表单数据。在发布表单前，可以在客户端捕获到大部分模型错误。模型错误的一个示例是，发布的日期字段值无法转换为日期。我们将在本教程后面的内容中讨论有关

客户端验证和模型验证的更多信息。

如果不存在模型错误，将保存数据，并且浏览器会重定向到索引页。

创建 Razor 页面

检查 Pages/Movies/Create.cshtml Razor 页面文件：

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @await Html.RenderPartialAsync("_ValidationScriptsPartial");
}
```

<form method="post"> 元素是一个表单标记帮助程序。表单标记帮助程序会自动包含防伪令牌。

基架引擎在模型中为每个字段(ID 除外)创建 Razor 标记，如下所示：

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

验证标记帮助程序 (`<div asp-validation-summary>` 和 ``) 显示验证错误。本系列后面的部分将更详细地讨论有关验证的信息。

标签标记帮助程序 (`<label asp-for="Movie.Title" class="control-label"></label>`) 生成标签描述和 `Title` 属性的 `for` 特性。

输入标记帮助程序 (`<input asp-for="Movie.Title" class="form-control" />`) 使用 `DataAnnotations` 属性并在客户端生成 jQuery 验证所需的 HTML 属性。

下一教程将介绍 SQLite 和数据库的种子设定。

[上一篇：添加模型](#)

[下一篇：SQLITE](#)

使用 SQLite 和 Razor 页面

2018/5/8 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

`MovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using RazorPagesMovie.Models;
    // using Microsoft.EntityFrameworkCore;

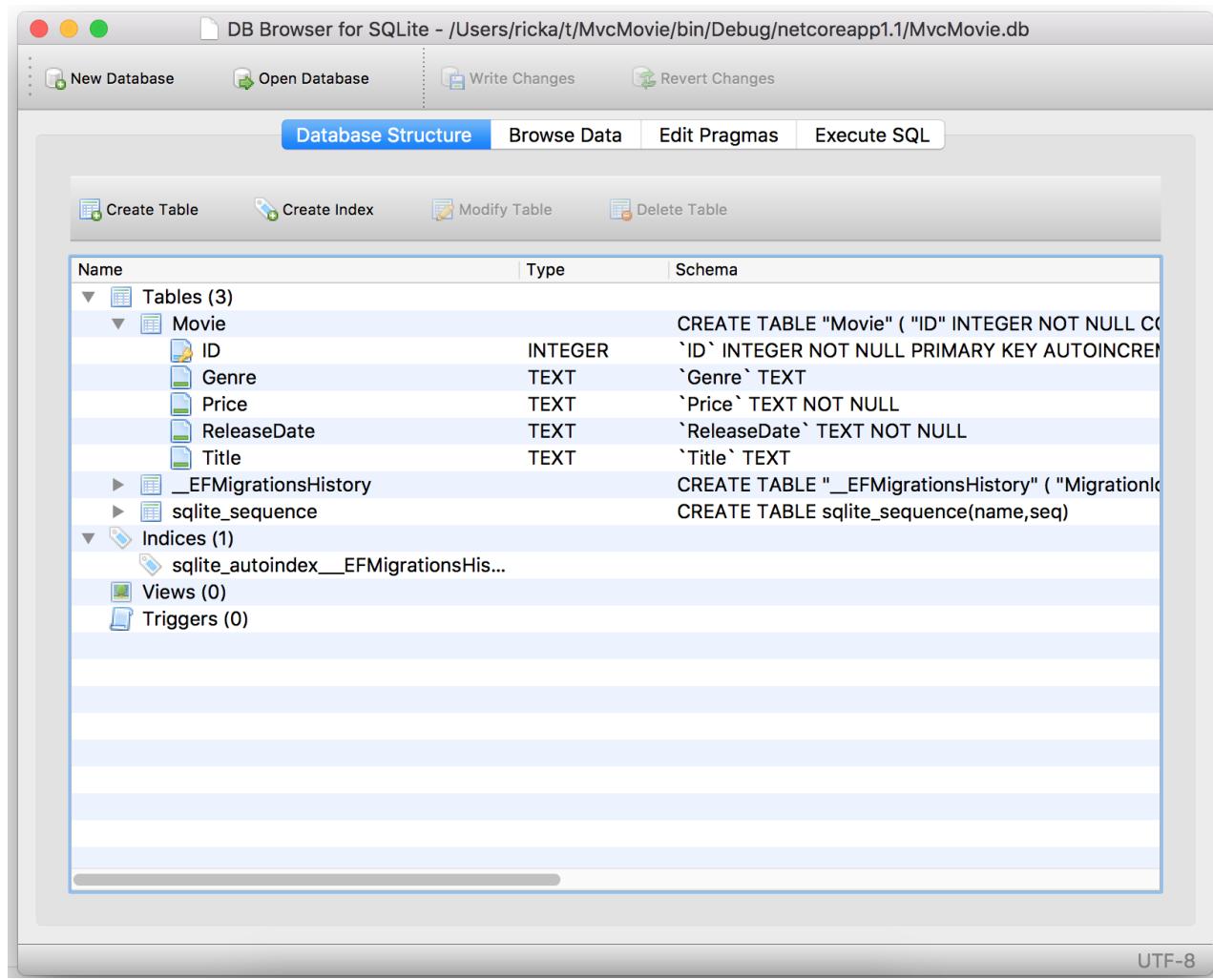
    services.AddDbContext<MovieContext>(options =>
        options.UseSqlite(Configuration.GetConnectionString("MovieContext")));
    services.AddMvc();
}
```

SQLite

[SQLite](#) 网站上表示:

SQLite 是一个自包含、高可靠性、嵌入式、功能完整、公共域的 SQL 数据库引擎。SQLite 是世界上使用最多的数据引擎。

可以下载许多第三方工具来管理并查看 SQLite 数据库。下面的图片来自 [DB Browser for SQLite](#)。如果你有最喜欢的 SQLite 工具, 请发表评论以分享你喜欢的方面。



设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}

```

如果 DB 中没有任何电影，则会返回种子初始值设定项。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法：

```
// Unused usings removed.
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<MovieContext>();
                    // requires using Microsoft.EntityFrameworkCore;
                    context.Database.Migrate();
                    // Requires using RazorPagesMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

删除 DB 中的所有记录(使种子方法运行)。停止并启动应用以设定数据库种子。

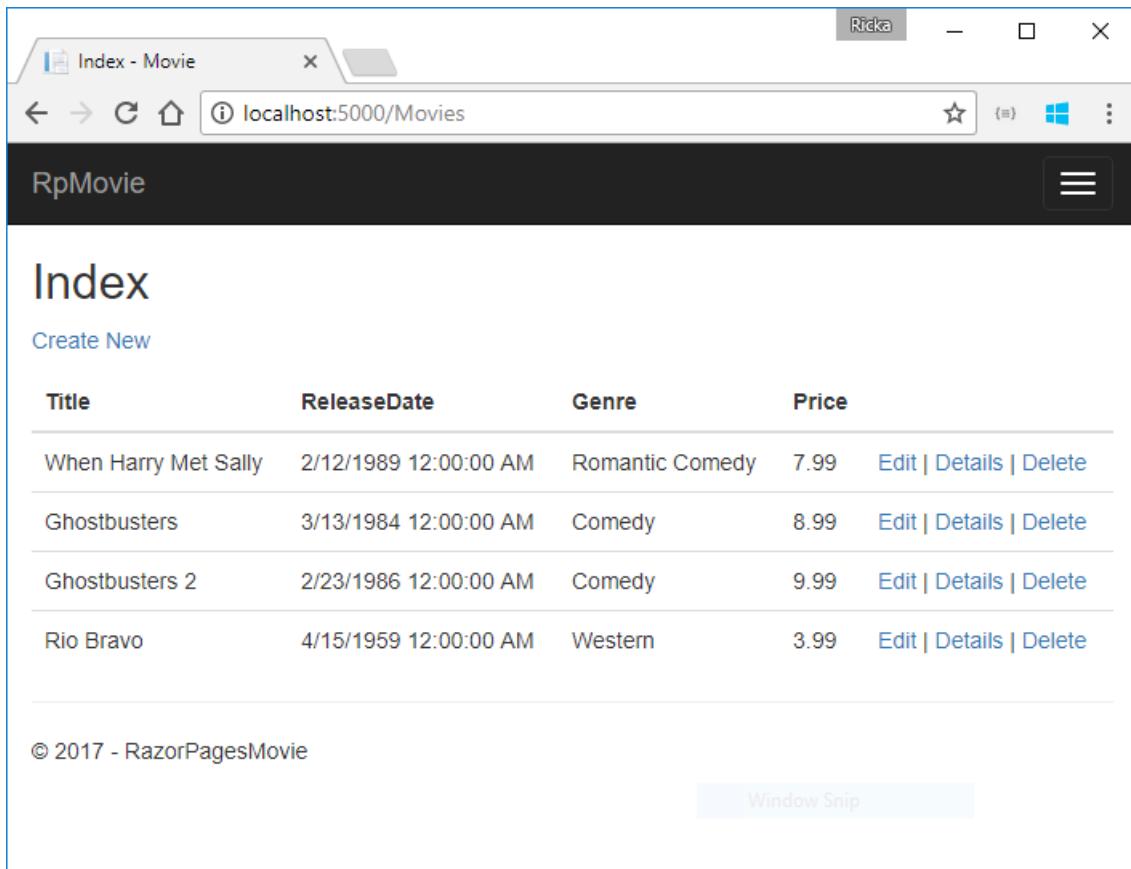
应用将显示设定为种子的数据。

更新生成的页面

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(如下图所示的 12:00:00 AM), 并且“ReleaseDate”应为“Release Date”(两个词)。



The screenshot shows a Microsoft Edge browser window titled "Index - Movie". The address bar displays "localhost:5000/Movies". The main content area is titled "RpMovie" and contains the word "Index". Below it is a "Create New" button. A table lists four movies:

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	2/12/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

At the bottom left, there is a copyright notice: "© 2017 - RazorPagesMovie". At the bottom right, there is a watermark: "Window Snip".

更新生成的代码

打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

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

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

我们将在下一教程中介绍 `DataAnnotations`。`Display` 特性指定要显示的字段名称的内容(本例中应为“Release Date”，而不是“ReleaseDate”)。`DataType` 属性指定数据的类型(日期)，使字段中存储的时间信息不会显示。

浏览到 Pages/Movies，并将鼠标悬停在“编辑”链接上以查看目标 URL。

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959	Western	3.99	Edit Details Delete

© 2017 - RazorPagesMovie

localhost:5000/Movies/Edit?id=2

“编辑”、“详细信息”和“删除”链接是在 Pages/Movies/Index.cshtml 文件中由[定位标记帮助程序](#)生成的。

```
@foreach (var item in Model.Movie) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            <a asp-page=".Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-page=".Details" asp-route-id="@item.ID">Details</a> |
            <a asp-page=".Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
```

[标记帮助程序](#)使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在前面的代码中，

`AnchorTagHelper` 从 Razor 页面(路由是相对的)、`asp-page` 和路由 ID (`asp-route-id`) 动态生成 HTML `href` 特性值。有关详细信息，请参阅[页面的 URL 生成](#)。

在最喜欢的浏览器中使用“查看源”来检查生成的标记。生成的 HTML 的一部分如下所示：

```
<td>
    <a href="/Movies/Edit?id=1">Edit</a> |
    <a href="/Movies/Details?id=1">Details</a> |
    <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

动态生成的链接通过查询字符串传递电影 ID(例如 `http://localhost:5000/Movies/Details?id=2`)。

更新“编辑”、“详细信息”和“删除”Razor 页面以使用“`{id:int?}`”路由模板。将上述每个页面的页面指令从 `@page` 更改为 `@page "{id:int?}"`。运行应用，然后查看源。生成的 HTML 会将 ID 添加到 URL 的路径部分：

```
<td>
    <a href="/Movies/Edit/1">Edit</a> |
    <a href="/Movies/Details/1">Details</a> |
    <a href="/Movies/Delete/1">Delete</a>
</td>
```

如果对具有“`{id:int?}`”路由模板的页面进行的请求中不包含整数，则将返回 HTTP 404(未找到)错误。例如，`http://localhost:5000/Movies/Details` 将返回 404 错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

更新并发异常处理

在 Pages/Movies/Edit.cshtml.cs 文件中更新 `OnPostAsync` 方法。下列突出显示的代码显示了更改：

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!(_context.Movie.Any(e => e.ID == Movie.ID)))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }
}

return RedirectToPage("./Index");
}
```

之前的代码仅检测当第一个并发客户端删除电影以及第二个并发客户端对电影发布更改时的并发异常。

测试 `catch` 块：

- 在 `catch (DbUpdateConcurrencyException)` 上设置断点
- 编辑电影。

- 在其他浏览器窗口中，选择同一电影的“删除”链接，然后删除此电影。
- 在之前的浏览器窗口中，将更改发布到电影。

当两个或更多客户端同时更新记录时，生产代码通常将检测到并发冲突。有关详细信息，请参阅[处理并发冲突](#)。

发布和绑定审阅

检查 `Pages/Movies/Edit.cshtml.cs` 文件：[!code-csharp]

当对 `Movies/Edit` 页面进行 HTTP GET 请求时（例如 `http://localhost:5000/Movies/Edit/2`）：

- `OnGetAsync` 方法从数据库提取电影并返回 `Page` 方法。
- `Page` 方法呈现“`Pages/Movies/Edit.cshtml`”Razor 页面。`Pages/Movies/Edit.cshtml` 文件包含模型指令（`@model RazorPagesMovie.Pages.Movies.EditModel`），这使电影模型在页面上可用。
- “编辑”表单中会显示电影的值。

当发布 `Movies/Edit` 页面时：

- 此页面上的表单值将绑定到 `Movie` 属性。`[BindProperty]` 特性会启用[模型绑定](#)。

```
[BindProperty]  
public Movie Movie { get; set; }
```

- 如果模型状态中存在错误（例如，`ReleaseDate` 无法被转换为日期），则会使用已提交的值再次发布表单。
- 如果没有模型错误，则电影已保存。

“索引”、“创建”和“删除”Razor 页面中的 HTTP GET 方法遵循一个类似的模式。“创建”Razor 页面中的 HTTP POST `OnPostAsync` 方法遵循的模式类似于“编辑”Razor 页面中的 `OnPostAsync` 方法所遵循的模式。

在下一教程中将添加搜索。

[上一篇：使用](#)

[添加搜索](#)

[SQLITE](#)

将搜索添加到 Razor 页面应用

2018/5/8 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本文档中，将向索引页面添加搜索功能以实现按“流派”或“名称”搜索电影。

使用以下代码更新索引页面的 `OnGetAsync` 方法：

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

```
public async Task OnGetAsync(string searchString)  
{  
    var movies = from m in _context.Movie  
                 select m;  
  
    if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
  
    Movie = await movies.ToListAsync();  
}
```

`OnGetAsync` 方法的第一行创建了 [LINQ](#) 查询用于选择电影：

```
var movies = from m in _context.Movie  
             select m;
```

此时仅对查询进行了定义，它还不会针对数据库运行。

如果 `searchString` 参数包含一个字符串，电影查询则会被修改为根据搜索字符串进行筛选：

```
if (!String.IsNullOrEmpty(searchString))  
{  
    movies = movies.Where(s => s.Title.Contains(searchString));  
}
```

`s => s.Title.Contains()` 代码是 [Lambda 表达式](#)。Lambda 在基于方法的 [LINQ](#) 查询中用作标准查询运算符方法的参数，如 [Where](#) 方法或 `Contains`（前面的代码中所使用的）。在对 [LINQ](#) 查询进行定义或通过调用方法（如 `Where`、`Contains` 或 `OrderBy`）进行修改后，此查询不会被执行。相反，会延迟执行查询。这意味着表达式的计算会延迟，直到循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关详细信息，请参阅 [Query Execution](#)（查询执行）。

注意：`Contains` 方法在数据库中运行，而不是在 C# 代码中。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上，`Contains` 映射到 [SQL LIKE](#)，这是不区分大小写的。在 SQLite 中，由于使用了默认排序规则，因此需要区分大小写。

导航到电影页面，并向 URL 追加一个如 `?searchString=Ghost` 的查询字符串（例如 `http://localhost:5000/Movies?searchString=Ghost`）。筛选的电影将显示出来。

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

© 2017 - RazorPagesMovie

如果向索引页面添加了以下路由模板，搜索字符串则可作为 URL 段传递（例如 `http://localhost:5000/Movies/ghost`）。

```
@page "{searchString?}"
```

前面的路由约束允许按路由数据（URL 段）搜索标题，而不是按查询字符串值进行搜索。`"{searchString?}"` 中的 `?` 表示这是可选路由参数。

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit Details Delete

© 2017 - RazorPagesMovie

但是，不能指望用户修改 URL 来搜索电影。在此步骤中，会添加 UI 来筛选电影。如果已添加路由约束 `"{searchString?}"`，请将它删除。

打开 Pages/Movies/Index.cshtml 文件，并添加以下代码中突出显示的 `<form>` 标记：

```

@page
@model RazorPagesMovie.Pages.Movies.IndexModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@

```

HTML `<form>` 标记使用[表单标记帮助程序](#)。提交表单时，筛选器字符串将发送到 Pages/Movies/Index 页面。保存更改并测试筛选器。

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	\$8.99	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	\$9.99	Edit Details Delete

按流派搜索

将以下突出显示的属性添加到 Pages/Movies/Index.cshtml.cs：

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.MovieContext _context;

    public IndexModel(RazorPagesMovie.Models.MovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    public SelectList Genres { get; set; }
    public string MovieGenre { get; set; }
```

`SelectList Genres` 包含流派列表。这使用户能够从列表中选择一种流派。

`MovieGenre` 属性包含用户选择的特定流派(例如“西部”)。

使用以下代码更新 `OnGetAsync` 方法：

```
// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task OnGetAsync(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

下面的代码是一种 LINQ 查询，可从数据库中检索所有流派。

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;
```

流派的 `SelectList` 是通过投影截然不同的流派创建的。

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

添加“按流派搜索”

更新 `Index.cshtml`, 如下所示：

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

通过按流派或/和电影标题搜索来测试应用。

[上一篇：更新页面](#) [下一篇：添加新字段](#)

在带有 Visual Studio for Mac 的 macOS 上使用 ASP.NET Core MVC 创建 Web 应用

2018/4/10 • 1 min to read • [Edit Online](#)

下面的一系列教程介绍使用 Visual Studio for Mac 生成 ASP.NET Core MVC Web 应用所涉及的基础知识。

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#)，再使用 MVC 版本。[Razor 页面教程](#)：

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面](#) 版本选择本教程，请在[此 GitHub 问题](#)中说明原因。

1. [入门](#)
2. [添加控制器](#)
3. [添加视图](#)
4. [添加模型](#)
5. [SQLite](#)
6. [控制器方法和视图](#)
7. [添加搜索](#)
8. [添加新字段](#)
9. [添加验证](#)
10. [检查 Details 和 Delete 方法](#)

ASP.NET Core MVC 和 Visual Studio for Mac 入门

2018/5/14 • 2 min to read • [Edit Online](#)

作者: Rick Anderson

本教程介绍使用 [Visual Studio for Mac](#) 生成 ASP.NET Core MVC Web 应用所涉及的基础知识。

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#)，再使用 MVC 版本。[Razor 页面教程](#):

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面版本](#) 选择本教程，请在[此 GitHub 问题](#)中说明原因。

本教程提供 3 个版本：

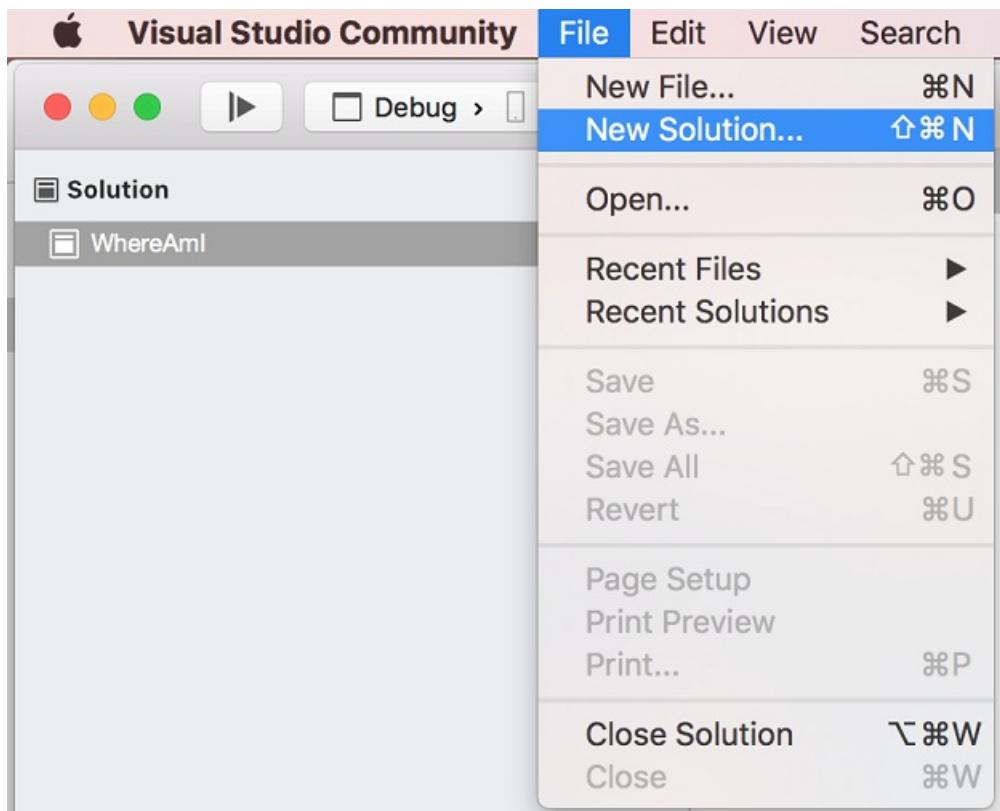
- macOS: [使用 Visual Studio for Mac 生成 ASP.NET Core MVC 应用](#)
- Windows: [使用 Visual Studio 生成 ASP.NET Core MVC 应用](#)
- Linux、macOS 和 Windows: [使用 Visual Studio Code 生成 ASP.NET Core MVC 应用](#)

系统必备

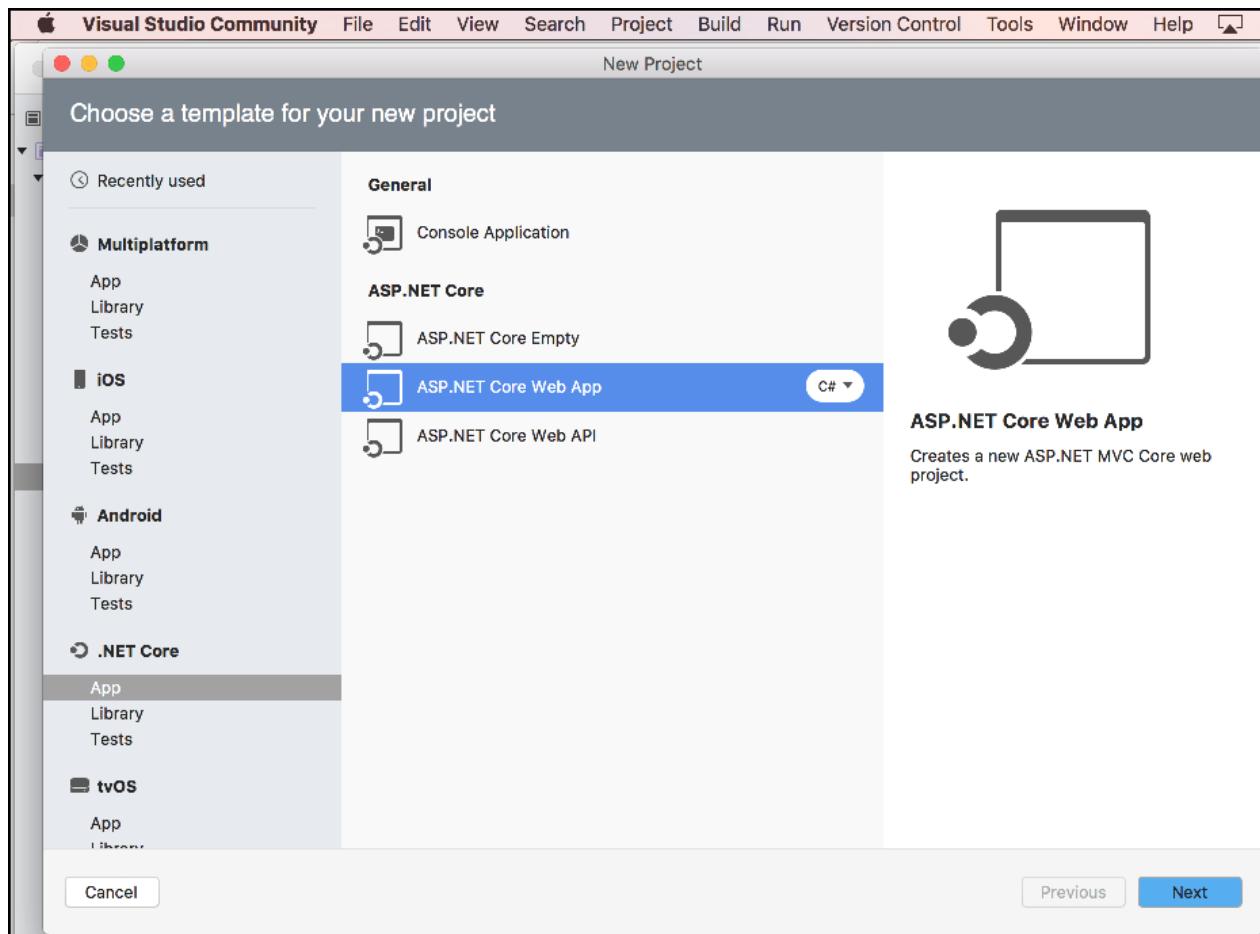
[Visual Studio for Mac](#)

创建 Web 应用

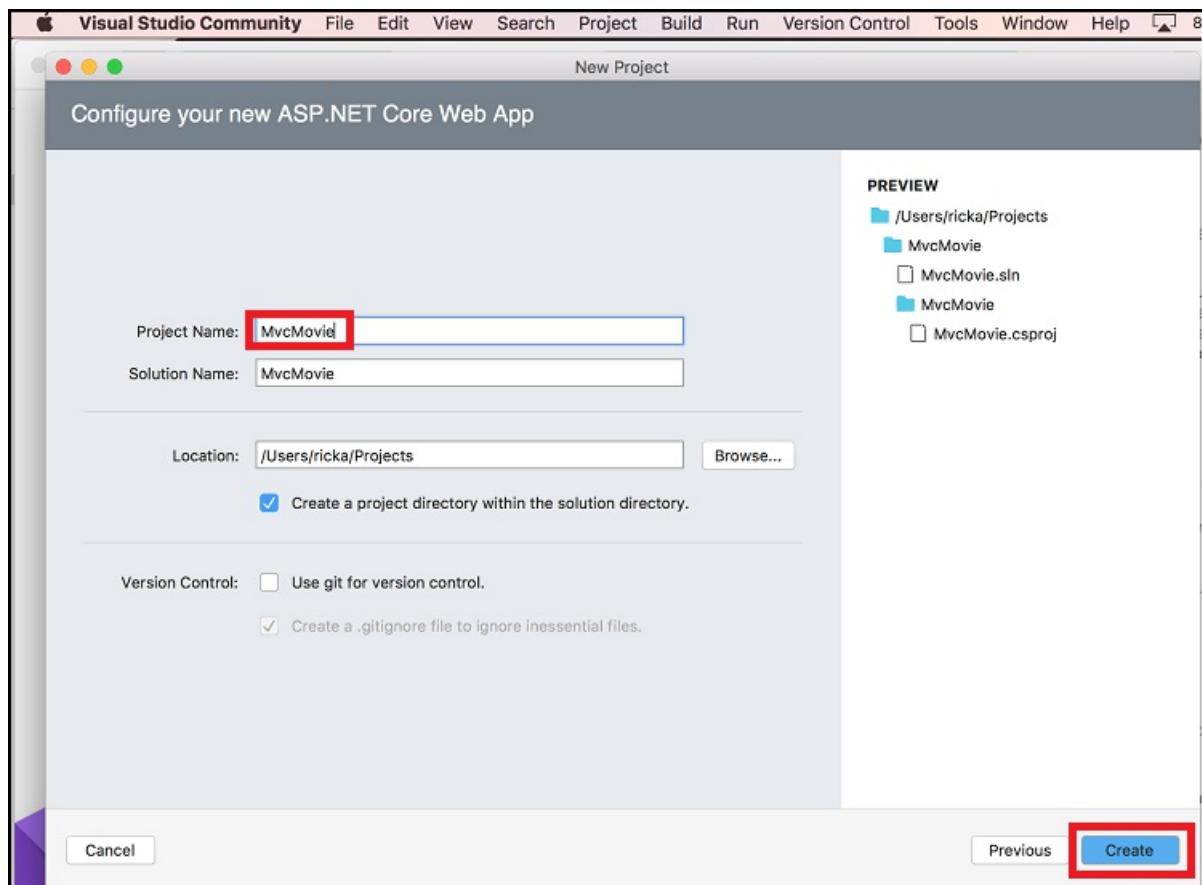
在 Visual Studio 中，选择“文件”>“新建解决方案”。



选择“.NET Core App”>“ASP.NET Core”>“Web 应用”>“下一步”。



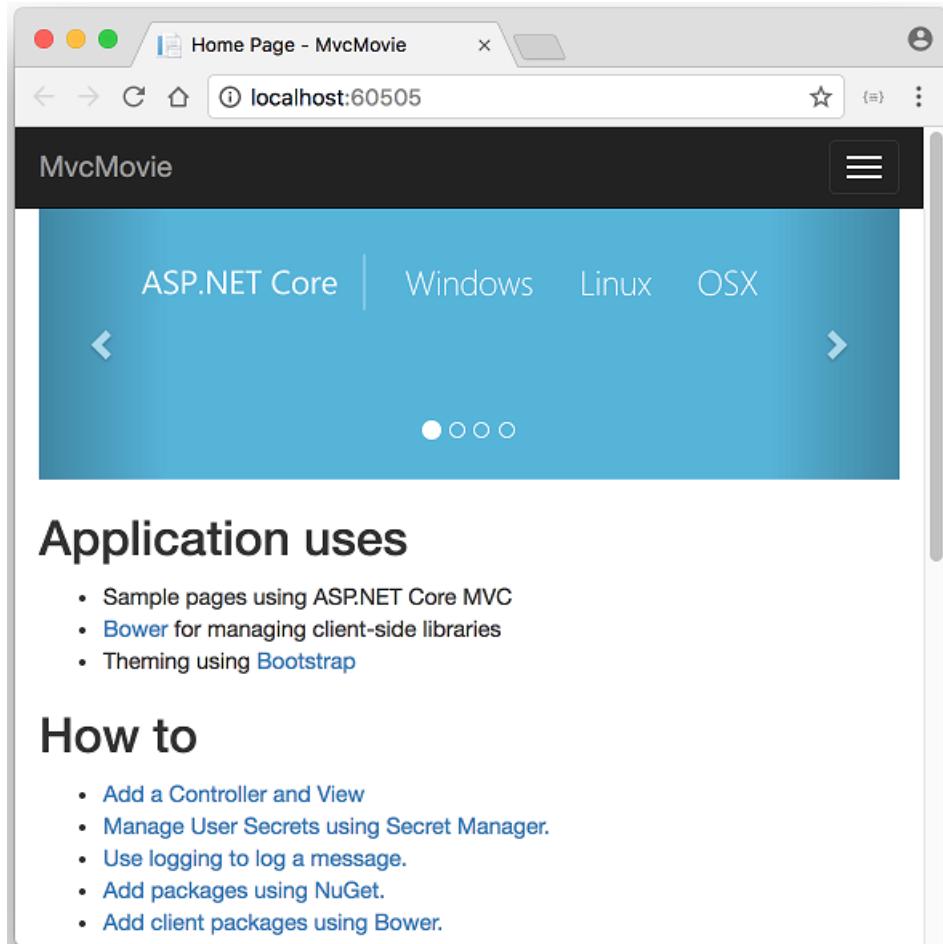
将项目命名为“MvcMovie”，然后选择“创建”。



启动应用

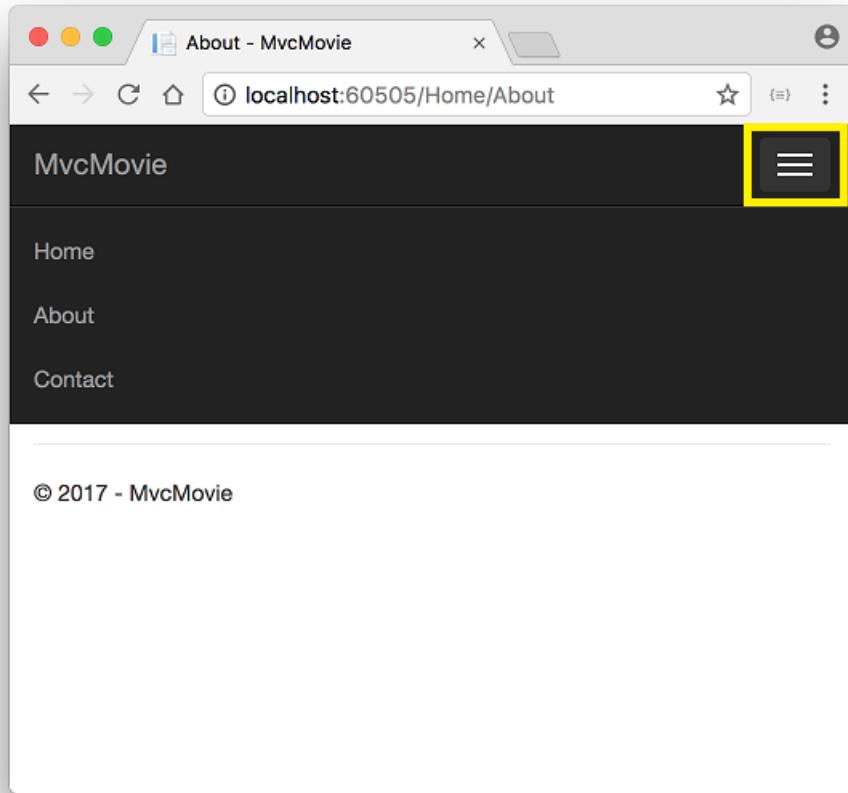
在 Visual Studio 中，选择“运行”>“开始执行(不调试)”以启动应用。Visual Studio 启动 Kestrel，启动浏览器并导航

到 `http://localhost:port`，其中的“端口”是随机选择的端口号。



- 地址栏显示 `localhost:port#`，而不是显示 `example.com`。这是因为 `localhost` 是本地计算机的标准主机名。Visual Studio 创建 Web 项目时，Web 服务器使用的是随机端口。运行应用时，将看到不同的端口号。
- 可以从“运行”菜单中以调试或非调试模式启动应用。

默认模板提供了“主页”、“关于”和“联系人”的链接。上面的浏览器图像未显示这些链接。根据浏览器的大小，可能需要单击导航图标才能显示这些链接。



在本教程的下一部分中，你将了解 MVC 并开始撰写一些代码。

[下一篇](#)

使用 Visual Studio for Mac 将控制器添加到 ASP.NET Core MVC 应用

2018/5/14 • 7 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

模型-视图-控制器 (MVC) 体系结构模式将应用分成 3 个主要组件: 模型 (M)、视图 (V) 和控制器 (C)。MVC 模式有助于创建比传统单片应用更易于测试和更新的应用。基于 MVC 的应用包含:

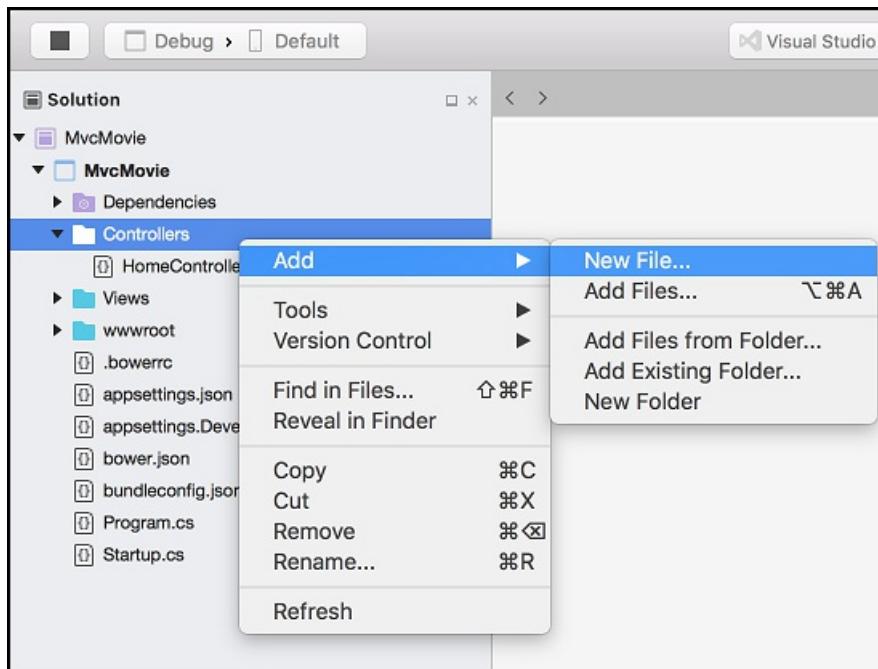
- **模式:** 表示应用数据的类。模型类使用验证逻辑来对该数据强制实施业务规则。通常, 模型对象检索模型状态并将其存储在数据库中。本教程中, `Movie` 模型将从数据库中检索电影数据, 并将其提供给视图或对其进行更新。更新后的数据将写入到数据库。
- **视图:** 视图是显示应用用户界面 (UI) 的组件。此 UI 通常会显示模型数据。
- **控制器:** 处理浏览器请求的类。它们检索模型数据并调用返回响应的视图模板。在 MVC 应用中, 视图仅显示信息; 控制器处理并响应用户输入和交互。例如, 控制器处理路由数据和查询字符串值, 并将这些值传递给模型。该模型可使用这些值查询数据库。例如, `http://localhost:1234/Home/About` 具有 `Home` (控制器) 的路由数据和 `About` (在 `Home` 控制器上调用的操作方法)。`http://localhost:1234/Movies/Edit/5` 是一个请求, 用于通过电影控制器编辑 ID 为 5 的电影。本教程稍后将探讨路由数据。

MVC 模式可帮助创建分隔不同应用特性(输入逻辑、业务逻辑和 UI 逻辑)的应用, 同时让这些元素之间实现松散耦合。该模式可指定应用中每种逻辑的位置。UI 逻辑位于视图中。输入逻辑位于控制器中。业务逻辑位于模型中。这种隔离有助于控制构建应用时的复杂程度, 因为它可用于一次处理一个实现特性, 而不影响其他特性的代码。例如, 处理视图代码时不必依赖业务逻辑代码。

本教程系列中介绍了这些概念, 并展示了如何使用它们构建电影应用。MVC 项目包含“控制器”和“视图”文件夹。

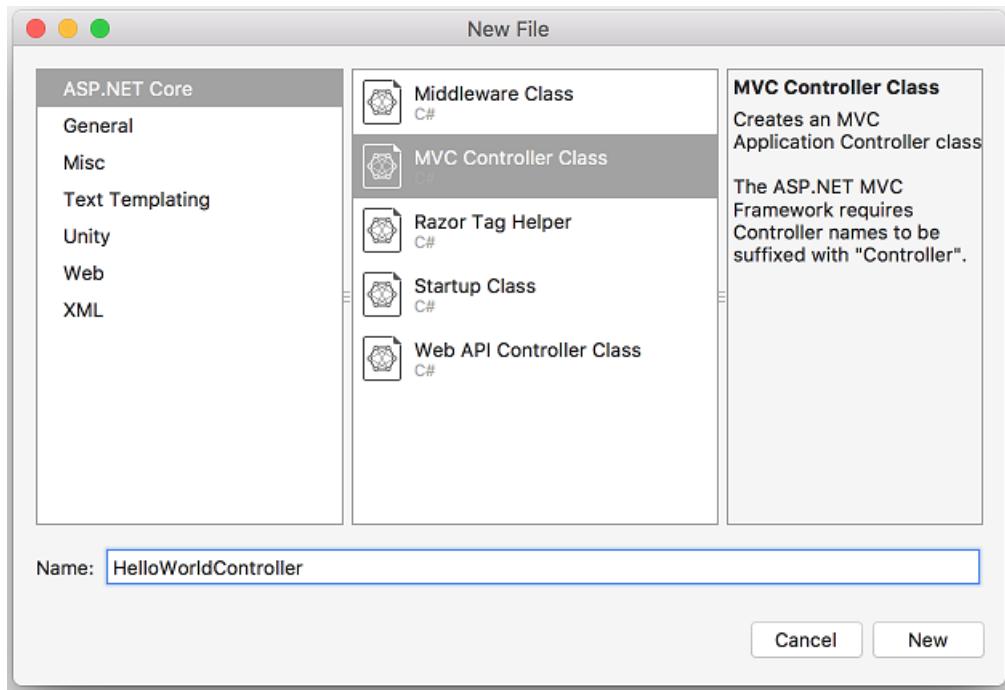
添加控制器

在“解决方案资源管理器”中, 右键单击“控制器”, 选择“添加”>“新文件”。



选择“ASP.NET Core”和“MVC 控制器类”。

将控制器命名为“HelloWorldController”。



将“Controllers/HelloWorldController.cs”的内容替换为以下内容：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

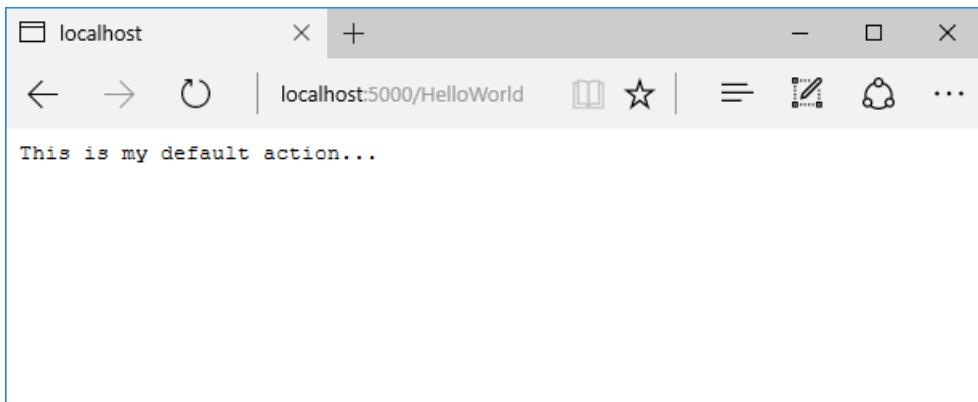
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

控制器中的每个 `public` 方法均可作为 HTTP 终结点调用。上述示例中，两种方法均返回一个字符串。请注意每个方法前面的注释。

HTTP 终结点是 Web 应用程序中可定向的 URL(例如 `http://localhost:1234/HelloWorld`)，其中结合了所用的协议 `HTTP`、`TCP` 端口等 Web 服务器的网络位置 `localhost:1234`，以及目标 URI `HelloWorld`。

第一条注释指出这是一个 `HTTP GET` 方法，它通过向基 URL 追加“/HelloWorld/”进行调用。第二条注释指定一个 `HTTP GET` 方法，它通过向 URL 追加“/HelloWorld/Welcome/”进行调用。本教程稍后将使用基架引擎生成 `HTTP POST` 方法。

在非调试模式下运行应用，并将“HelloWorld”追加到地址栏中的路径。`Index` 方法返回一个字符串。



MVC 根据入站 URL 调用控制器类(及其中的操作方法)。MVC 所用的默认 URL 路由逻辑使用如下格式来确定调用的代码:

```
/[Controller]/[ActionName]/[Parameters]
```

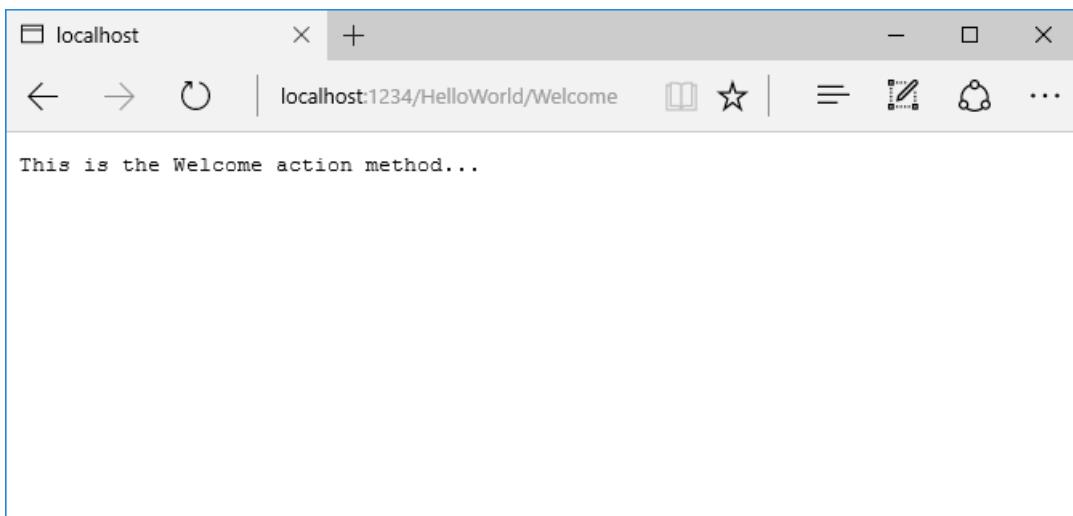
在 Startup.cs 文件的 `Configure` 方法中设置路由的格式。

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

如果运行应用且不提供任何 URL 段, 它将默认为上面突出显示的模板行中指定的“Home”控制器和“Index”方法。

第一个 URL 段决定要运行的控制器类。因此 `localhost:xxxx/HelloWorld` 映射到 `HelloWorldController` 类。该 URL 段的第二部分决定类上的操作方法。因此 `localhost:xxxx/HelloWorld/Index` 将触发 `HelloWorldController` 类的 `Index` 运行。请注意, 只需浏览到 `localhost:xxxx/HelloWorld`, 而 `Index` 方法默认调用。原因是 `Index` 是默认方法, 如果未显式指定方法名称, 则将在控制器上调用它。URL 段的第三部分 (`id`) 针对的是路由数据。稍后可在本教程中看到路由数据。

浏览到 `http://localhost:xxxx/HelloWorld/Welcome`。`Welcome` 方法运行并返回字符串“这是 Welcome 操作方法...”。对于此 URL, 采用 `HelloWorld` 控制器和 `Welcome` 操作方法。目前尚未使用 URL 的 `[Parameters]` 部分。



修改代码, 将一些参数信息从 URL 传递到控制器。例如 `/HelloWorld/Welcome?name=Rick&numtimes=4`。更改 `Welcome` 方法以包括以下代码中显示的两个参数:

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

前面的代码：

- 使用 C# 可选参数功能指示，未为 `numTimes` 参数传递值时该参数默认为 1。
- 使用 `HtmlEncoder.Default.Encode` 防止恶意输入（即 JavaScript）损害应用。
- 使用内插字符串。

运行应用并浏览到：

`http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

（将 xxxx 替换为端口号。）可在 URL 中对 `name` 和 `numtimes` 使用其他值。MVC 模型绑定系统可将命名参数从地址栏中的查询字符串自动映射到方法中的参数。有关详细信息，请参阅[模型绑定](#)。

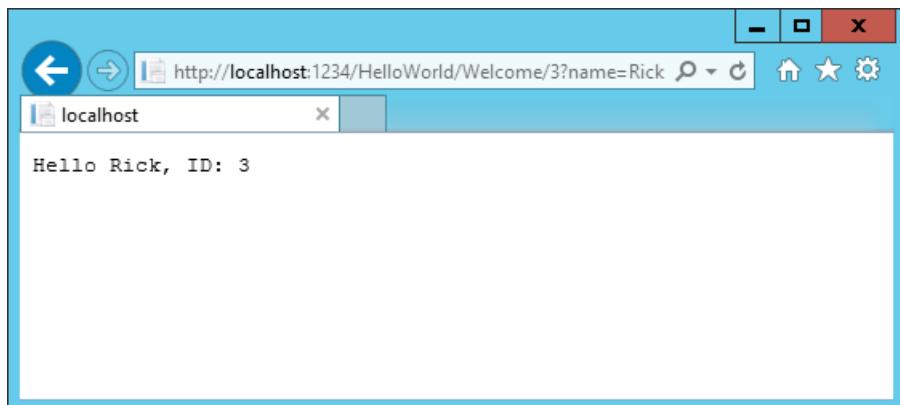


在上图中，未使用 URL 段 (`Parameters`)，且 `name` 和 `numTimes` 参数作为[查询字符串](#)进行传递。上述 URL 中的 `?` (问号) 为分隔符，后接查询字符串。`&` 字符用于分隔查询字符串。

将 `Welcome` 方法替换为以下代码：

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

运行应用并输入以下 URL：`http://localhost:xxx/HelloWorld/Welcome/3?name=Rick`



此时，第三个 URL 段与路由参数 `id` 相匹配。`Welcome` 方法包含 `MapRoute` 方法中匹配 URL 模板的参数 `id`。后

面的 `? (id?)` 表示 `id` 参数可选。

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

上述示例中，控制器始终执行 MVC 的“VC”部分，即视图和控制器工作。控制器将直接返回 HTML。通常不希望控制器直接返回 HTML，因为编码和维护非常繁琐。通常，需使用单独的 Razor 视图模板文件来帮助生成 HTML 响应。可在下一教程中执行该操作。

[上一页](#)

[下一页](#)

将视图添加到 ASP.NET Core MVC 应用

2018/5/14 • 9 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将修改 `HelloWorldController` 类, 进而使用 Razor 视图模板文件来顺利封装为客户端生成 HTML 响应的过程。

使用 Razor 创建视图模板文件。基于 Razor 的模板具有".cshtml"文件扩展名。它们提供了一种巧妙的方法来使用 C# 创建 HTML 输出。

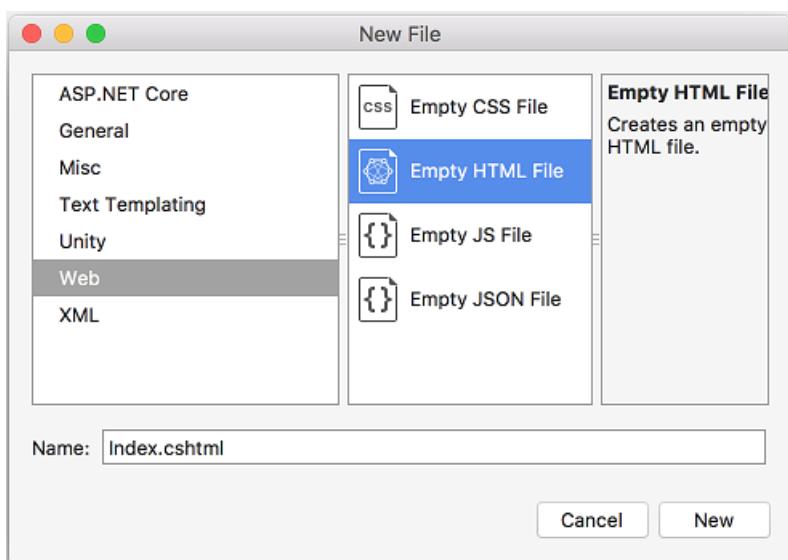
当前, `Index` 方法返回带有在控制器类中硬编码的消息的字符串。在 `HelloWorldController` 类中, 将 `Index` 方法替换为以下代码:

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

上述代码返回 `View` 对象。它使用视图模板对浏览器生成 HTML 响应。类似上述 `Index` 方法的控制器方法(也称为操作方法)通常返回 `IActionResult`(或派生自 `ActionResult` 的类), 而非类似字符串的类型。

添加视图

- 右键单击“视图”文件夹, 然后单击“添加”>“新文件夹”, 并将文件夹命名为“HelloWorld”。
- 右键单击“Views/HelloWorld”文件夹, 然后单击“添加”>“新文件”。
- 在“新建文件”对话框中:
 - 在左窗格中, 选择“Web”。
 - 在中间窗格中, 选择“空 HTML 文件”。
 - 在“名称”框中键入“Index.cshtml”。
 - 选择“新建”。



使用以下内容替换 Razor 视图文件 `Views/HelloWorld/Index.cshtml` 的内容:

```

@{
    ViewData["Title"] = "Index";
}

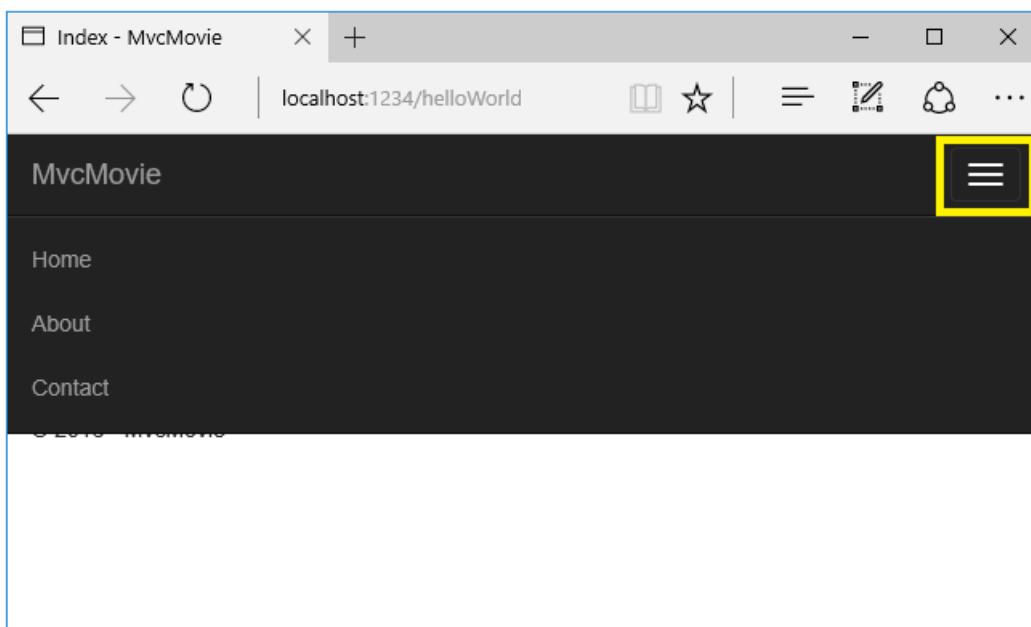
<h2>Index</h2>

<p>Hello from our View Template!</p>

```

导航到

如果浏览器窗口较小（例如在移动设备上），则可能需要在右上角切换（点击）Bootstrap 导航按钮以查看“首页”、“关于”和“联系人”链接。



更改视图和布局页面

点击菜单链接（“MvcMovie”、“首页”、“关于”）。每页显示相同的菜单布局。菜单布局是在 `Views/Shared/_Layout.cshtml` 文件中实现的。打开 `Views/Shared/_Layout.cshtml` 文件。

布局 模板使你能够在一个位置指定网站的 HTML 容器布局，然后将它应用到网站中的多个页面。查找 `@RenderBody()` 行。`RenderBody` 是显示创建的所有特定于视图的页面的占位符，已包装在布局页面中。例如，如果

选择“关于”链接，Views/Home/About.cshtml 视图将在 `RenderBody` 方法中呈现。

更改布局文件中的标题和菜单链接

在标题元素中，将 `MvcMovie` 更改为 `Movie App`。将布局模板中的定位文本从 `MvcMovie` 更改为 `Movie App`，并将控制器从 `Home` 更改为 `Movies`，如下所示：

注意：ASP.NET Core 2.0 版本略有不同。它不包含 `@inject ApplicationInsights` 和 `@Html.Raw(JavaScriptSnippet.FullScript)`。

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href("~/css/site.css") />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
              value="absolute" />
        <link rel="stylesheet" href("~/css/site.min.css" asp-append-version="true" />
    </environment>
    @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Movies" asp-action="Index" class="navbar-brand">Movie App</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>© 2017 - MvcMovie</p>
        </footer>
    </div>

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src "~/js/site.js" asp-append-version="true"></script>
    </environment>
```

```

<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfhII9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-Tc5IQib027qvyyjSMfHjOMaLkfUWVxZxUPnCJA712mCWNIPG9mGCD8wGNICPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

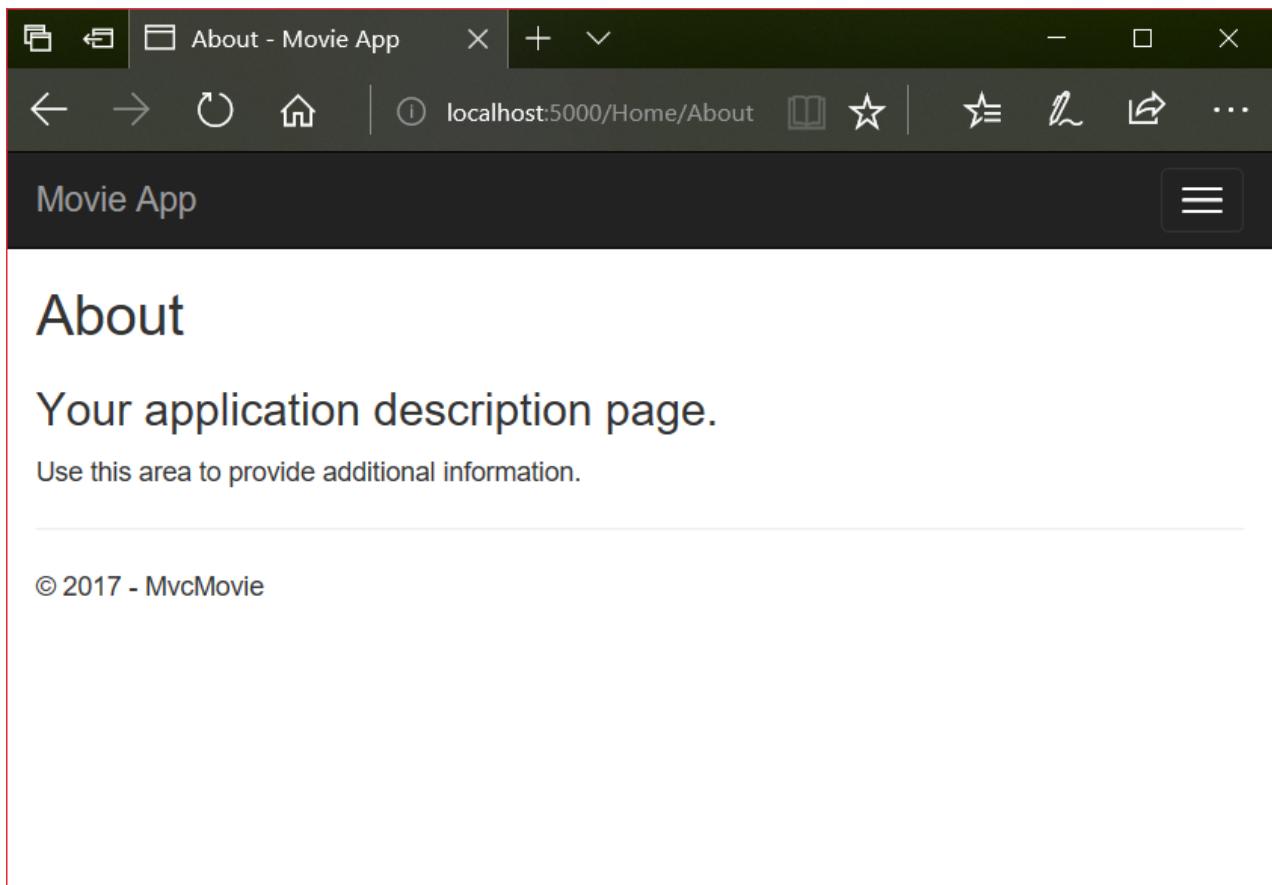
@RenderSection("Scripts", required: false)
</body>
</html>

```

警告

我们尚未实现 `Movies` 控制器，所以如果单击此链接，将收到 404(未找到)错误。

保存更改并点击“关于”链接。请注意浏览器选项卡上的标题现在显示的是“关于 - 电影应用”，而不是“关于 - **Mvc 电影**”：



点击“联系人”链接，请注意，标题和定位文本还会显示“电影应用”。我们能够在布局模板中进行一次更改，让网站上的所有页面都反映新的链接文本和新标题。

检查 `Views/_ViewStart.cshtml` 文件：

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

Views/_ViewStart.cshtml 文件将 Views/Shared/_Layout.cshtml 文件引入到每个视图中。可以使用 `Layout` 属性设置不同的布局视图，或将它设置为 `null`，这样将不会使用任何布局文件。

更改 `Index` 视图的标题。

打开 Views/HelloWorld/Index.cshtml。有两处需要更改的地方：

- 浏览器标题中显示的文字。
- 辅助标题(`<h2>` 元素)。

稍稍对它们进行一些更改，这样可以看出哪一段代码更改了应用的哪部分。

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

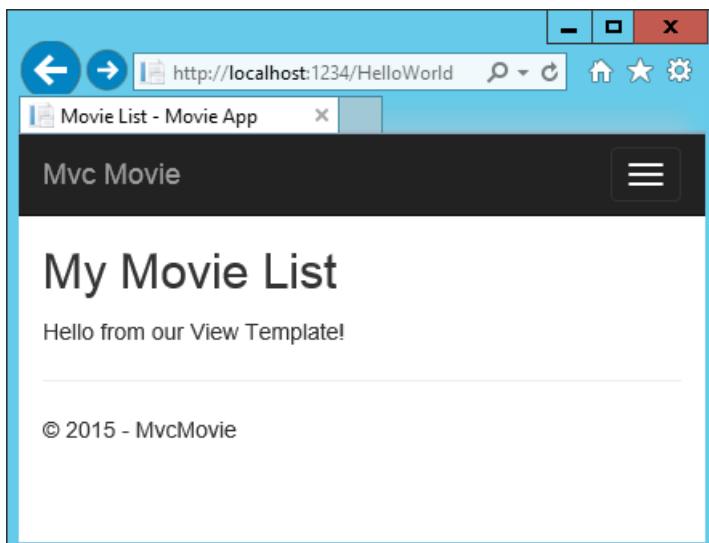
<p>Hello from our View Template!</p>
```

上述代码中的 `ViewData["Title"] = "Movie List";` 将 `ViewData` 字典的 `Title` 属性设置为“Movie List”。`Title` 属性在布局页面中的 `<title>` HTML 元素中使用：

```
<title>@ViewData["Title"] - Movie App</title>
```

保存更改并导航到 `http://localhost:xxxx/HelloWorld`。请注意，浏览器标题、主标题和辅助标题已更改。(如果没有在浏览器中看到更改，则可能正在查看缓存的内容。在浏览器中按 `Ctrl + F5` 强制加载来自服务器的响应。)浏览器标题是使用我们在 `Index.cshtml` 视图模板中设置的 `ViewData["Title"]` 以及在布局文件中添加的额外“- Movie App”创建的。

还要注意，`Index.cshtml` 视图模板中的内容是如何与 `Views/Shared/_Layout.cshtml` 视图模板合并的，并且注意单个 HTML 响应被发送到了浏览器。凭借布局模板可以很容易地对应用程序中所有页面进行更改。若要了解更多信息，请参阅[布局](#)。



但我们一点点“数据”(在此示例中为“Hello from our View Template!”消息)是硬编码的。MVC 应用程序有一个“V”(视图)，而你已有一个“C”(控制器)，但还没有“M”(模型)。

将数据从控制器传递给视图

控制器操作会被调用以响应传入的 URL 请求。控制器类是编写处理传入浏览器请求的代码的地方。控制器从数据源检索数据，并决定将哪些类型的响应发送回浏览器。可以从控制器使用视图模板来生成并格式化对浏览器的 HTML 响应。

控制器负责提供所需的数据，使视图模板能够呈现响应。最佳做法：视图模板不应该直接执行业务逻辑或与数据库进行交互。相反，视图模板应仅使用由控制器提供给它的数据。保持此“关注点分离”有助于保持代码干净、可测试性和可维护性。

目前，`HelloWorldController` 类中的 `Welcome` 方法采用 `name` 和 `ID` 参数，然后将值直接输出到浏览器。应将控制器更改为使用视图模板，而不是使控制器将此响应呈现为字符串。视图模板会生成动态响应，这意味着必须将适当的数据位从控制器传递给视图以生成响应。为此，可以让控制器将视图模板所需的动态数据（参数）放置在视图模板稍后可以访问的 `ViewData` 字典中。

返回到 `HelloWorldController.cs` 文件，并更改 `Welcome` 方法以将 `Message` 和 `NumTimes` 值添加到 `ViewData` 字典。`ViewData` 字典是一个动态对象，这意味着你可以将任何所需的内容放在其中；只有将内容放在其中后 `ViewData` 对象才具有定义的属性。[MVC 模型绑定系统](#)自动将命名参数（`name` 和 `numTimes`）从地址栏中的查询字符串映射到方法中的参数。完整的 `HelloWorldController.cs` 文件如下所示：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

`ViewData` 字典对象包含将传递给视图的数据。

创建一个名为 `Views/HelloWorld/Welcome.cshtml` 的 `Welcome` 视图模板。

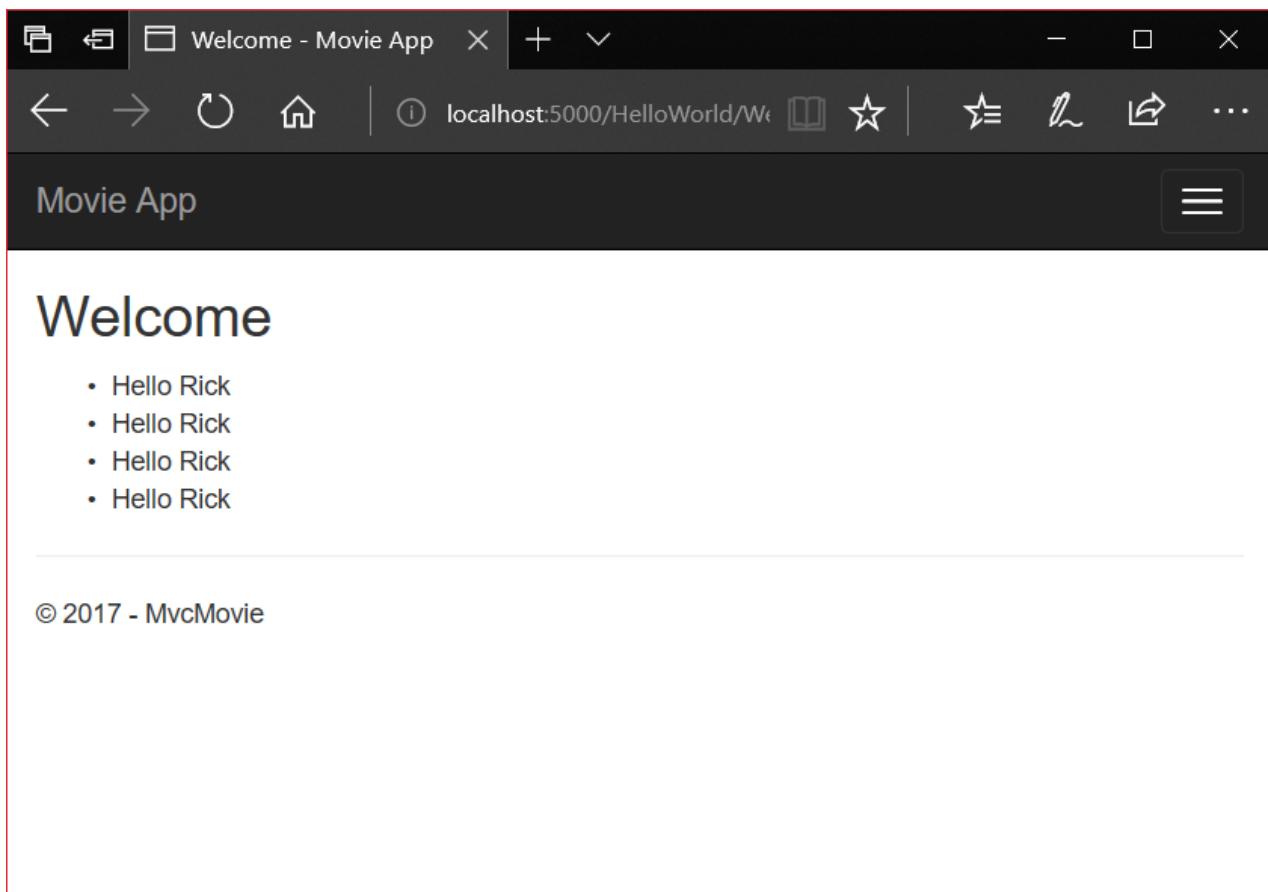
在 `Welcome.cshtml` 视图模板中创建一个循环，显示“Hello”`NumTimes`。使用以下内容替换 `Views/HelloWorld/Welcome.cshtml` 的内容：

```
@{  
    ViewData["Title"] = "Welcome";  
}  
  
<h2>Welcome</h2>  
  
<ul>  
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)  
    {  
        <li>@ViewData["Message"]</li>  
    }  
</ul>
```

保存更改并浏览到以下 URL：

```
http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4
```

数据取自 URL，并传递给使用 [MVC 模型绑定器](#) 的控制器。控制器将数据打包到 `ViewData` 字典中，并将该对象传递给视图。然后，视图将数据作为 HTML 呈现给浏览器。



在上面的示例中，我们使用 `ViewData` 字典将数据从控制器传递给视图。稍后在本教程中，我们将使用视图模型将数据从控制器传递给视图。传递数据的视图模型方法通常比 `ViewData` 字典方法更为优先。有关详细信息，请参阅 [ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC](#) (MVC 中 ViewModel、ViewData、ViewBag、TempData 和 Session 之间的比较)。

当然，这是模型的一种“M”类型，而不是数据库类。让我们用学到的内容来创建一个电影数据库。

使用 Visual Studio for Mac 将模型添加到 ASP.NET Core MVC 应用

2018/5/14 • 6 min to read • [Edit Online](#)

将模型添加到 ASP.NET Core MVC 应用

作者: [Rick Anderson](#) 和 [Tom Dykstra](#)

在本部分中，将添加用于管理数据库中电影的一些类。这些类将是 MVC 应用的“Model”部分。

可以结合 [Entity Framework Core](#) (EF Core) 使用这些类来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化需要编写的数据访问代码。[EF Core 支持许多数据库引擎](#)。

要创建的模型类称为 POCO 类(源自“普通旧 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们只定义将存储在数据库中的数据的属性。

在本教程中，首先将编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍)是从已存在的数据库生成模型类。有关此方法的信息，请参阅 [ASP.NET Core - Existing Database](#)(ASP.NET Core - 现有数据库)。

添加数据模型类

- 右键单击“Models”文件夹，然后选择“添加”>“新建文件”。
- 在“新建文件”对话框中：
 - 在左侧窗格中，选择“常规”。
 - 在中间窗格中，选择“空类”。
 - 将此类命名为“Movie”，然后选择“新建”。

向 `Movie` 类添加以下属性：

```
using System;

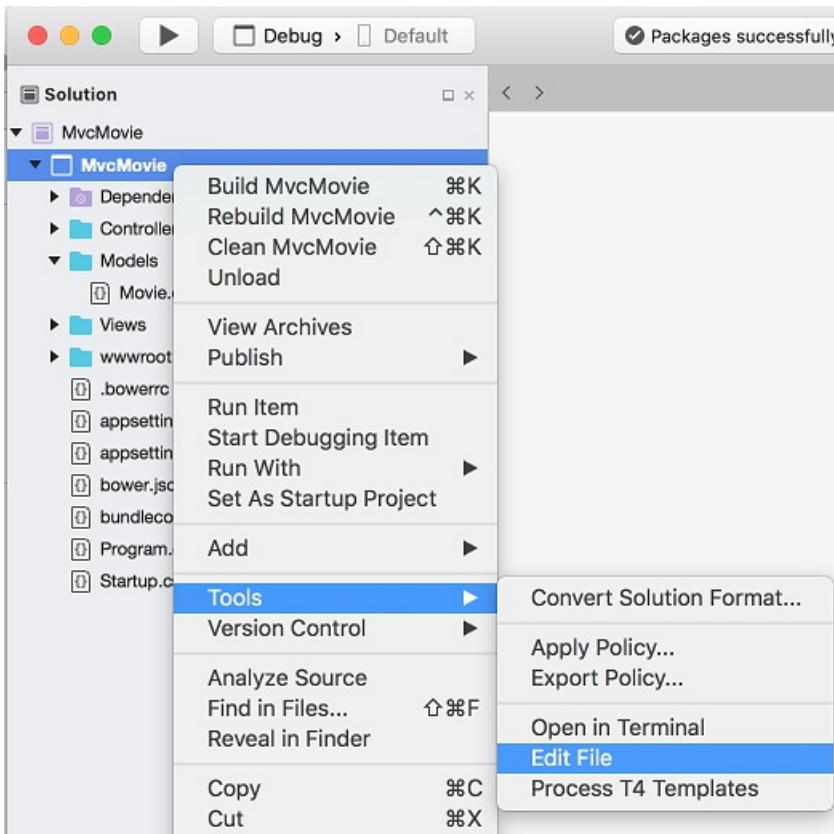
namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段以获取主键。

生成项目以验证有没有任何错误存在。现在 MVC 应用中已具有模型。

准备项目以搭建基架

- 右键单击项目文件，然后选择“工具”>“编辑文件”。



- 将以下突出显示的 NuGet 包添加到 MvcMovie.csproj 文件:

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>

```

- 保存该文件。
- 创建 Models/MvcMovieContext.cs 文件并添加以下 `MvcMovieContext` 类:[!code-csharp]
- 打开 Startup.cs 文件并添加两个 using:[!code-csharp]
- 将数据库上下文添加到 Startup.cs 文件:

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlite("Data Source=MvcMovie.db"));
}

```

这会告诉实体框架，数据模型中包含哪些模型类。现在定义一个 Movie 对象实体集，此实体集会表示为数

据库中的一个 Movie 表。

- 生成项目并确定没有任何错误。

为 MovieController 搭建基架

在项目文件夹中打开终端窗口，然后运行以下命令：

```
dotnet restore  
dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc MvcMovieContext --  
relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries
```

如果收到错误 No executable found matching command "dotnet-aspnet-codegenerator"，verify：

- 你现在位于项目目录中。项目目录中包含 Program.cs、Startup.cs 和 .csproj 文件。
- 你的 dotnet 版本是 1.1 或更高版本。运行 dotnet 获取版本。
- 你已将 <DotNetCliToolReference> 元素添加到 [MvcMovie.csproj](#) 文件。

基架引擎创建以下组件：

- 电影控制器 (Controllers/MoviesController.cs)
- “创建”、“删除”、“详细信息”、“编辑”和“索引”页面的 Razor 视图文件 (Views/Movies/*.cshtml)

自动创建 [CRUD](#)(创建、读取、更新和删除)操作方法和视图的过程称为“搭建基架”。你很快将具有功能完整的 Web 应用程序，可使用此应用程序管理电影数据库。

将文件添加到 Visual Studio

- 将 MovieController.cs 文件添加到 Visual Studio 项目：
 - 右键单击“控制器”文件夹，然后选择“添加”>“添加文件”。
 - 选择 MovieController.cs 文件。
- 添加“电影”文件夹和视图：
 - 右键单击“视图”文件夹，然后选择“添加”>“添加现有文件夹”。
 - 导航到“视图”文件夹，选择“视图\电影”，然后选择“打开”。
 - 在“从‘电影’选择要添加的文件”对话框中，选择“包括全部”，然后选择“确定”。

添加初始迁移

从命令行运行以下 .NET Core CLI 命令：

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

dotnet ef migrations add InitialCreate 命令生成用于创建初始数据库架构的代码。此架构以 (Models/MvcMovieContext.cs 文件中的) DbContext 中指定的模型为基础。Initial 参数用于为迁移命名。可以使用任意名称，但是按照惯例应选择描述迁移的名称。有关详细信息，请参阅[迁移简介](#)。

dotnet ef database update 命令在用于创建数据库的 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 Up 方法。

测试应用

- 运行应用并点击“Mvc Movie”链接。

- 点击“新建”链接，创建电影。

Title

Release Date

Genre

Price

- 可能无法在 `Price` 字段中输入小数点或逗号。若要使 `jQuery` 验证支持使用逗号（“,”）表示小数点及使用非美国英语日期格式的非英语区域设置，必须执行使应用全球化的步骤。请参阅 <https://github.com/aspnet/Docs/issues/4076> 和其他资源了解详细信息。目前，只能输入整数，例如 10。
- 在一些区域设置中，需要指定日期格式。请参阅下方突出显示的代码。

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

我们将在本教程的后续部分中探讨 `DataAnnotations`。

点击“创建”后，窗体会发布到服务器，其中电影信息会保存在数据库中。应用重定向到 /Movies URL，其中会显示新创建的电影信息。

Index

Create New

Title	Release Date	Genre	Price	
Conan	3/3/2017	Comedy	\$1.99	Edit Details Delete

© 2017 - MvcMovie

再创建几个其他的电影条目。试用“编辑”、“详细信息”和“删除”链接，它们均可正常工作。

现在你已拥有用于显示、编辑、更新和删除数据的数据库和页面。在下一个教程中，我们将使用此数据库。

其他资源

- [标记帮助程序](#)
- [全球化和本地化](#)

[上一篇：添加视图](#)

[下一篇：使用](#)

SQL

在 ASP.NET Core MVC 项目中使用 SQLite

2018/5/8 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

`MvcMovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlite("Data Source=MvcMovie.db"));

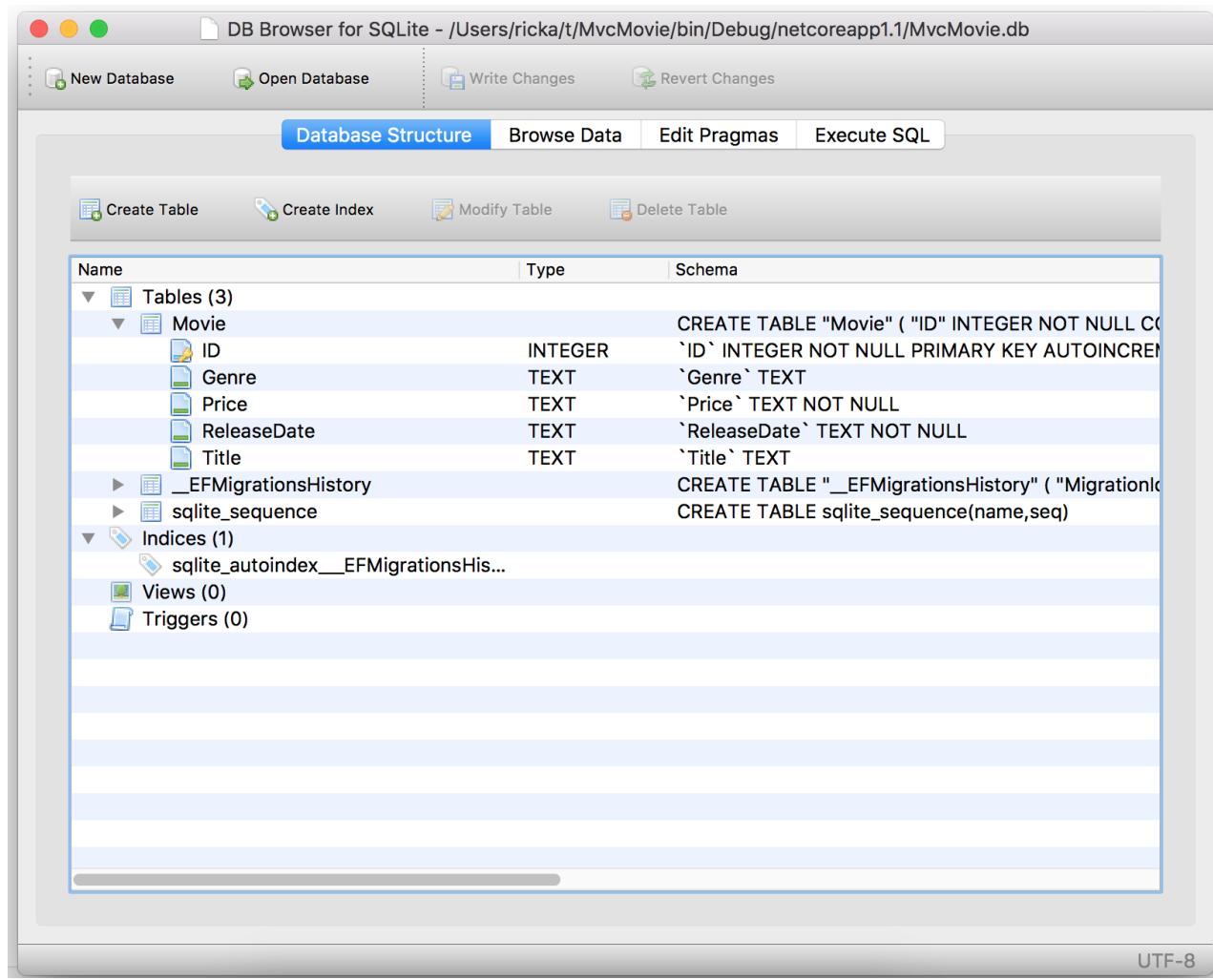
}
```

SQLite

[SQLite](#) 网站上表示:

SQLite 是一个自包含、高可靠性、嵌入式、功能完整、公共域的 SQL 数据库引擎。SQLite 是世界上使用最多的数据引擎。

可以下载许多第三方工具来管理并查看 SQLite 数据库。下面的图片来自 [DB Browser for SQLite](#)。如果你有最喜欢的 SQLite 工具, 请发表评论以分享你喜欢的方面。



设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-1-11"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

如果 DB 中没有任何电影，则会返回种子初始值设定项。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    // Requires using MvcMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

删除 DB 中的所有记录(使种子方法运行)。停止并启动应用以设定数据库种子。

应用将显示设定为种子的数据。

A screenshot of a Microsoft Edge browser window. The title bar shows "Ricka - Movie App". The address bar displays "localhost:5000/Movies". The main content area is titled "MvcMovie" with a "Create New" link. Below is a table listing four movies:

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

At the bottom left, it says "© 2017 - MvcMovie".

[上一篇 - 添加模型](#)

型

[下一篇 - 控制器方法和视图](#)

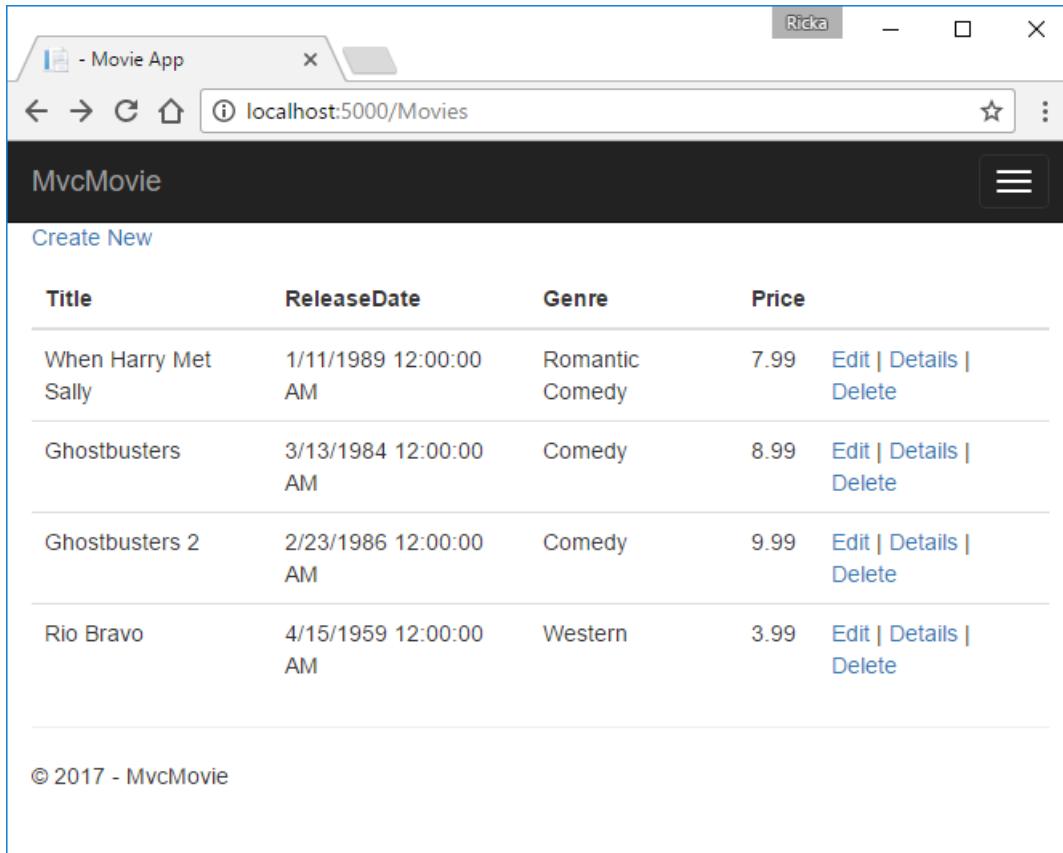
图

ASP.NET Core MVC 应用中的控制器方法和视图

2018/5/8 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(下图中的 12:00:00 AM), 并且“ReleaseDate”应为两个词。



Title	ReleaseDate	Genre	Price
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99

© 2017 - MvcMovie

打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

生成并运行应用。

我们将在下一教程中介绍 [DataAnnotations](#)。[Display](#) 特性指定要显示的字段名称的内容(本例中应为“Release

Date", 而不是"ReleaseDate")。 **DataType** 属性指定数据的类型(日期), 使字段中存储的时间信息不会显示。

浏览到 **Movies** 控制器, 并将鼠标指针悬停在"编辑"链接上以查看目标 URL。

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

© 2016 - MvcMovie

http://localhost:1234/Movies/Edit/5

"编辑"、"详细信息"和"删除"链接是在 Views/Movies/Index.cshtml 文件中由 Core MVC 定位标记帮助程序生成的。

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
<a asp-action="Details" asp-route-id="@item.ID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
```

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在上面的代码中, **AnchorTagHelper** 从控制器操作方法和路由 ID 动态生成 HTML **href** 特性值。在最喜欢的浏览器中使用"查看源", 或使用开发人员工具来检查生成的标记。生成的 HTML 的一部分如下所示:

```
<td>
<a href="/Movies/Edit/4"> Edit </a> |
<a href="/Movies/Details/4"> Details </a> |
<a href="/Movies/Delete/4"> Delete </a>
</td>
```

重新调用在 Startup.cs 文件中设置的**路由**的格式:

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

ASP.NET Core 将 `http://localhost:1234/Movies/Edit/4` 转换为对 **Movies** 控制器的 **Edit** 操作方法的请求, 参数 **Id** 为 4。(控制器方法也称为**操作方法**。)

标记帮助程序是 ASP.NET Core 中最受欢迎的新功能之一。有关详细信息，请参阅[其他资源](#)。

打开 `Movies` 控制器并检查两个 `Edit` 操作方法。以下代码显示了 `HTTP GET Edit` 方法，此方法将提取电影并填充由 `Edit.cshtml` Razor 文件生成的编辑表单。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

以下代码显示 `HTTP POST Edit` 方法，它会处理已发布的电影值：

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`[Bind]` 特性是防止过度发布的一种方法。只应在 `[Bind]` 特性中包含想要更改的属性。有关详细信息，请参阅 [Protect your controller from over-posting](#)(防止控制器过度发布)。[ViewModels](#) 提供了一种替代方法以防止过度发布。

请注意第二个 `Edit` 操作方法的前面是 `[HttpPost]` 特性。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

`HttpPost` 特性指定只能为 `POST` 请求调用此 `Edit` 方法。可将 `[HttpGet]` 属性应用于第一个编辑方法，但不是必需，因为 `[HttpGet]` 是默认设置。

`ValidateAntiForgeryToken` 特性用于[防止请求伪造](#)，并与编辑视图文件 (`Views/Movies/Edit.cshtml`) 中生成的防伪标记相配对。编辑视图文件使用[表单标记帮助程序](#)生成防伪标记。

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

[表单标记帮助程序](#)会生成隐藏的防伪标记，此标记必须与电影控制器的 `Edit` 方法中 `[ValidateAntiForgeryToken]` 生成的防伪标记相匹配。有关详细信息，请参阅[反请求伪造](#)。

`HttpGet` `Edit` 方法采用电影 `ID` 参数，使用 Entity Framework `SingleOrDefaultAsync` 方法查找电影，并将所选电影返回到“编辑”视图。如果无法找到电影，则返回 `NotFound` (HTTP 404)。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

当基架系统创建“编辑”视图时，它会检查 `Movie` 类并创建代码为类的每个属性呈现 `<label>` 和 `<input>` 元素。以下示例显示由 Visual Studio 基架系统生成的“编辑”视图：

```

@model MvcMovie.Models.Movie

 @{
     ViewData["Title"] = "Edit";
 }

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Price" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

请注意视图模板在文件顶端有一个 `@model MvcMovie.Models.Movie` 语句。`@model MvcMovie.Models.Movie` 指定视图期望的视图模板的模型为 `Movie` 类型。

基架的代码使用几个标记帮助程序方法来简化 HTML 标记。[标签标记帮助程序](#) 显示字段的名称 ("Title"、"ReleaseDate"、"Genre" 或 "Price")。[输入标记帮助程序](#) 呈现 HTML `<input>` 元素。[验证标记帮助程序](#) 显示与该属性相关联的任何验证消息。

运行应用程序并导航到 `/Movies` URL。点击“编辑”链接。在浏览器中查看页面的源。为 `<form>` 元素生成的 HTML 如下所示。

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxp63fRFqUePGvuI5jGzsloJu1L7X9le1gy7NCI1SduCRx9jDQC1rv9p0TTmqUyXnJBXhmrjcUVDJyDUMm7-MF_9rK8aAZdRd10ri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOsTEg7RU" />
</form>
```

`<input>` 元素位于 `HTML <form>` 元素中，后者的 `action` 特性设置为发布到 `/Movies/Edit/id` URL。当单击 `Save` 按钮时，表单数据将发布到服务器。关闭 `</form>` 元素之前的最后一行显示 [表单标记帮助程序](#) 生成的隐藏的 `XSRF` 标记。

处理 POST 请求

以下列表显示了 `Edit` 操作方法的 `[HttpPost]` 版本。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

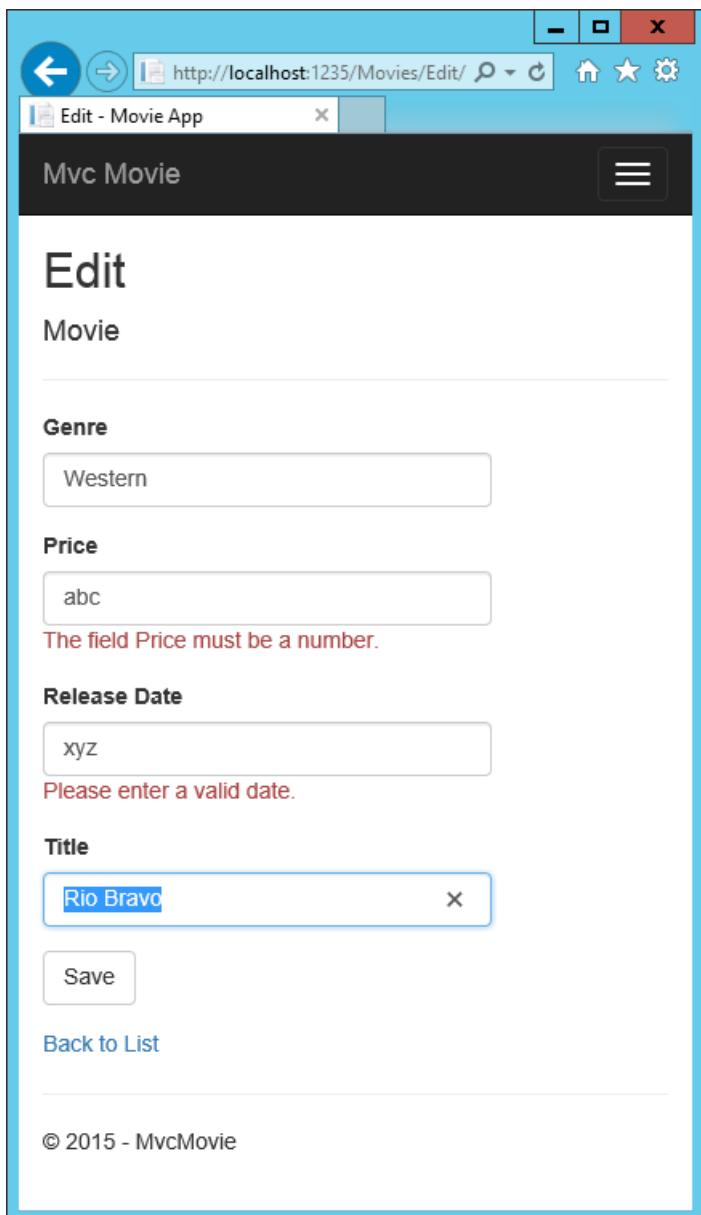
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

[ValidateAntiForgeryToken] 特性验证表单标记帮助程序中的防伪标记生成器生成的隐藏的 [XSRF](#) 标记

[模型绑定](#) 系统采用发布的表单值，并创建一个作为 `movie` 参数传递的 `Movie` 对象。`ModelState.IsValid` 方法验证表单中提交的数据是否可以用于修改(编辑或更新) `Movie` 对象。如果数据有效，将保存此数据。通过调用数据库上下文的 `SaveChangesAsync` 方法，将更新(编辑)的电影数据保存到数据库。保存数据后，代码将用户重定向到 `MoviesController` 类的 `Index` 操作方法，此方法显示电影集合，包括刚才所做的更改。

在表单发布到服务器之前，客户端验证会检查字段上的任何验证规则。如果有任何验证错误，则将显示错误消息，并且不会发布表单。如果禁用 JavaScript，则不会进行客户端验证，但服务器将检测无效的发布值，并且表单值将与错误消息一起重新显示。稍后在本教程中，我们将更详细地研究[模型验证](#)。`Views/Movies/Edit.cshtml` 视图模板中的[验证标记帮助程序](#)负责显示相应的错误消息。



电影控制器中的所有 `HttpGet` 方法都遵循类似的模式。它们获取电影对象(对于 `Index` 获取的是对象列表)并将对象(模型)传递给视图。`Create` 方法将空的电影对象传递给 `Create` 视图。在方法的 `[HttpPost]` 重载中, 创建、编辑、删除或其他方式修改数据的所有方法都执行此操作。以 `HTTP GET` 方式修改数据是一种安全隐患。以 `HTTP GET` 方法修改数据也违反了 HTTP 最佳做法和架构 REST 模式, 后者指定 GET 请求不应更改应用程序的状态。换句话说, 执行 GET 操作应是没有任何隐患的安全操作, 也不会修改持久数据。

其他资源

- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)
- [防请求伪造](#)
- [防止控制器过度发布](#)
- [ViewModels](#)
- [表单标记帮助程序](#)
- [输入标记帮助程序](#)
- [标签标记帮助程序](#)
- [选择标记帮助程序](#)
- [验证标记帮助程序](#)

[上一篇 - 使用
SQLITE](#)

[下一篇 - 添加搜
索](#)

将搜索添加到 ASP.NET Core MVC 应用

2018/5/8 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Index` 操作方法添加搜索功能, 以实现按“类型”或“名称”搜索电影。

使用以下代码更新 `Index` 方法:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

`Index` 操作方法的第一行创建了 `LINQ` 查询用于选择电影:

```
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义, 它还不会针对数据库运行。

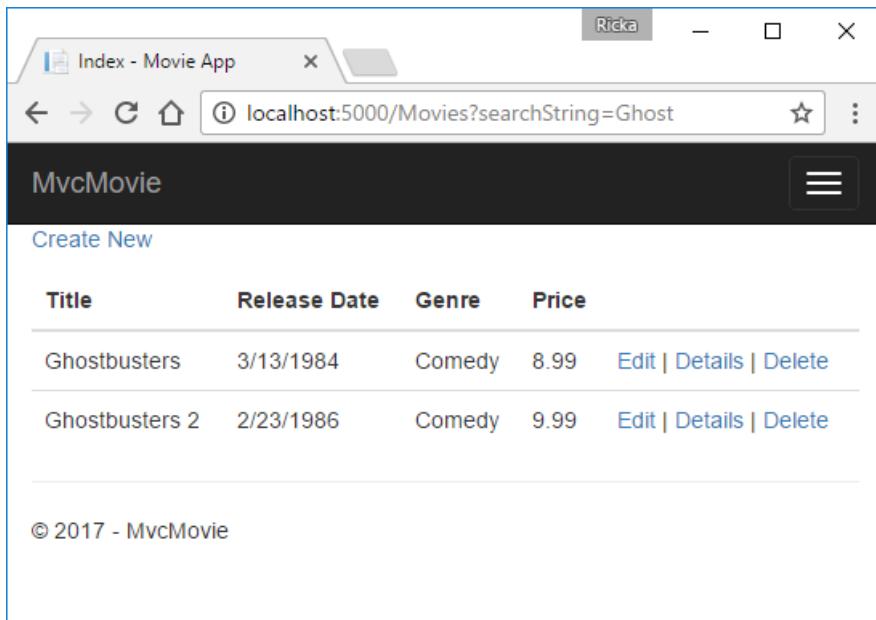
如果 `searchString` 参数包含一个字符串, 电影查询则会被修改为根据搜索字符串的值进行筛选:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

上面的 `s => s.Title.Contains()` 代码是 [Lambda 表达式](#)。Lambda 在基于方法的 `LINQ` 查询中用作标准查询运算符方法的参数, 如 `Where` 方法或 `Contains` (上述的代码中所使用的)。在对 `LINQ` 查询进行定义或通过调用方法 (如 `Where`、`Contains` 或 `OrderBy`) 进行修改后, 此查询不会被执行。相反, 会延迟执行查询。这意味着表达式的计算会延迟, 直到真正循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关延迟执行查询的详细信息, 请参阅[Query Execution](#)(查询执行)。

注意:`Contains` 方法在数据库上运行, 而不是在上面显示的 c# 代码中运行。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上, `Contains` 映射到 `SQL LIKE`, 这是不区分大小写的。在 SQLite 中, 由于使用了默认排序规则, 因此需要区分大小写。

导航到 `/Movies/Index`。将查询字符串(如 `?searchString=Ghost`)追加到 URL。筛选的电影将显示出来。



如果将 `Index` 方法的签名更改为具有名称为 `id` 的参数，则 `id` 参数将匹配 `Startup.cs` 中设置的默认路由的可选 `{id}` 占位符。

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

请注意：SQLite 区分大小写，因此需搜索“Ghost”而非“ghost”。

之前的 `Index` 方法：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

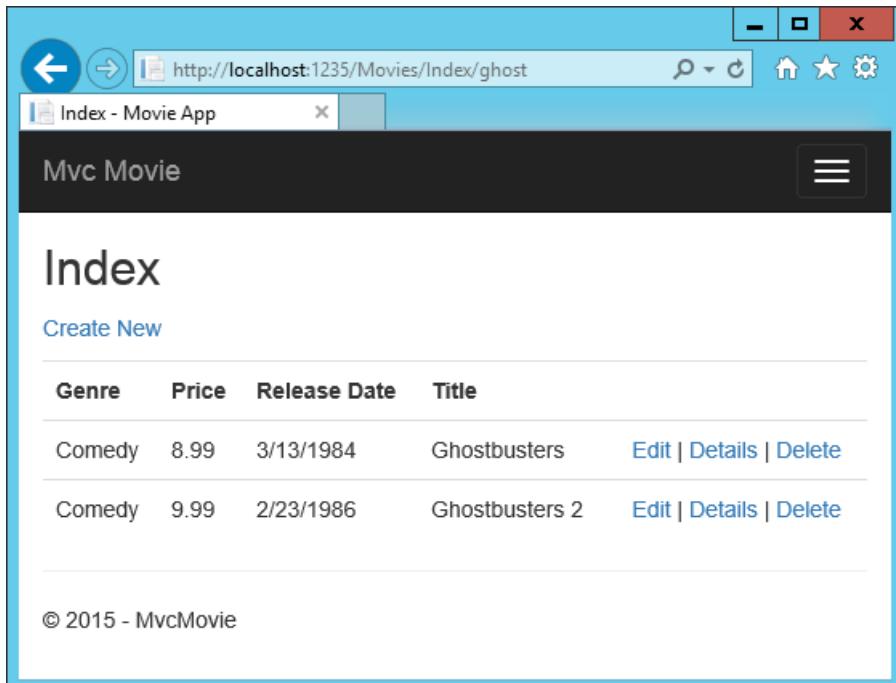
更新后带 `id` 参数的 `Index` 方法：

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

现可将搜索标题作为路由数据(URL 段)而非查询字符串值进行传递。



但是，不能指望用户在每次要搜索电影时都修改 URL。因此需要添加 UI 元素来帮助他们筛选电影。若已更改 `Index` 方法的签名，以测试如何传递绑定路由的 `ID` 参数，请改回原样，使其采用名为 `searchString` 的参数：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

打开“Views/Movies/Index.cshtml”文件，并添加以下突出显示的 `<form>` 标记：

```
ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

此 HTML `<form>` 标记使用表单标记帮助程序，因此提交表单时，筛选器字符串会发布到电影控制器的 `Index` 操作。

作。保存更改，然后测试筛选器。

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

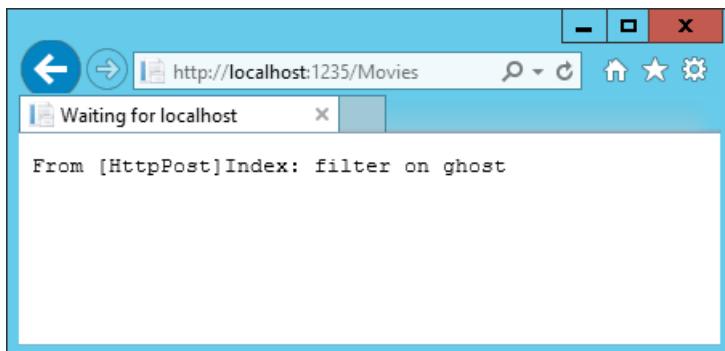
如你所料，不存在 `Index` 方法的 `[HttpPost]` 重载。无需重载，因为该方法不更改应用的状态，仅筛选数据。

可添加以下 `[HttpPost] Index` 方法。

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

`notUsed` 参数用于创建 `Index` 方法的重载。本教程稍后将对此进行探讨。

如果添加此方法，则操作调用程序将与 `[HttpPost] Index` 方法匹配，且将运行 `[HttpPost] Index` 方法，如下图所示。



但是，即使添加 `Index` 方法的 `[HttpPost]` 版本，其实现方式也受到限制。假设你想要将特定搜索加入书签，或向朋友发送一个链接，让他们单击链接即可查看筛选出的相同电影列表。请注意，HTTP POST 请求的 URL 与 GET 请求的 URL 相同 (`localhost:xxxxx/Movies/Index`)，其中不包含搜索信息。搜索字符串信息作为表单域值发送给服务器。可使用浏览器开发人员工具或出色的 `Fiddler` 工具对其进行验证。下图展示了 Chrome 浏览器开发人员工具：

The screenshot shows the Chrome DevTools Network tab with a red box highlighting the 'Movies' entry in the list. Another red box highlights the 'General' section of the request details, showing the URL, method, status code, and headers. A third red box highlights the 'Form Data' section, showing the search string and a long RequestVerificationToken.

在请求正文 中，可看到搜索参数和 **XSRF** 标记。请注意，正如之前教程所述，[表单标记帮助程序](#) 会生成一个 XSRF 防伪标记。不会修改数据，因此无需验证控制器方法中的标记。

搜索参数位于请求正文而非 URL 中，因此无法捕获该搜索信息进行书签设定或与他人共享。将通过指定请求为 **HTTP GET** 进行修复。

在“Views\movie\Index.cshtml”Razor 视图中更改 `<form>` 标记以指定 `method="get"`：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

现在提交搜索后，URL 将包含搜索查询字符串。即使具备 `HttpPost Index` 方法，搜索也将转到 `HttpGet Index` 操

作方法。

Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete

以下标记显示对 `form` 标记的更改：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

添加“按流派搜索”

将以下 `MovieGenreViewModel` 类添加到“模型”文件夹：

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

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}
```

“电影流派”视图模型将包含：

- 电影列表。
- 包含流派列表的 `SelectList`。用户可通过它从列表中选择一种流派。
- 包含所选流派的 `movieGenre`。

将 `MoviesController.cs` 中的 `Index` 方法替换为以下代码：

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();

    return View(movieGenreVM);
}

```

以下代码是一种 `LINQ` 查询，可从数据库中检索所有流派。

```

// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

```

通过投影不同的流派创建 `SelectList`（我们不希望选择列表中的流派重复）。

```
movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())
```

向索引视图添加“按流派搜索”

按如下更新 `Index.cshtml`：

```

@model MvcMovie.Models.MovieGenreViewModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="movieGenre" asp-items="Model.genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.movies[0].Title)
```

在上述代码中，`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。由于只检查但未计算 Lambda 表达式，因此当 `model`、`model.movies[0]` 或 `model.movies` 为 `null` 或空时，你不会收到访问冲突。对 Lambda 表达式求值时（例如，`@Html.DisplayFor(modelItem => item.Title)`），将求得该模型的属性值。

通过按流派或/和电影标题搜索来测试应用。

[上一篇 - 控制器方法和视图](#)

[下一篇 - 添加字段](#)

添加新字段

2018/5/8 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程将向 `Movies` 表添加一个新字段。当更改架构(添加一个新的字段)时, 我们将丢弃数据库并创建一个新的数据库。此工作流适用于在没有要保留的任何生产数据的早期开发阶段。

在部署了应用并且具有要保留的数据后, 在需要更改架构时则不能丢弃数据库。Entity Framework [Code First 迁移](#)使你能够更新架构并迁移数据库, 而不会导致数据丢失。迁移是使用 SQL Server 时的一个常用的功能, 但 SQLite 不支持许多迁移架构操作, 因此只可能进行非常简单的迁移。有关详细信息, 请参阅 [SQLite 限制](#)。

向电影模型添加分级属性

打开 Models/Movie.cs 文件, 并添加 `Rating` 属性:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

因为已经添加新字段到 `Movie` 类, 所以还需要更新绑定允许名单, 将此新属性纳入其中。在 MoviesController.cs 中, 更新 `Create` 和 `Edit` 操作方法的 `[Bind]` 属性, 以包括 `Rating` 属性:

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

还需要更新视图模板以在浏览器视图中显示、创建和编辑新的 `Rating` 属性。

编辑 /Views/Movies/Index.cshtml 文件并添加 `Rating` 字段:

```


| @Html.DisplayNameFor(model => model.movies[0].Title) | @Html.DisplayNameFor(model => model.movies[0].ReleaseDate) | @Html.DisplayNameFor(model => model.movies[0].Genre) | @Html.DisplayNameFor(model => model.movies[0].Price) | @Html.DisplayNameFor(model => model.movies[0].Rating) |
|------------------------------------------------------|------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------|-------------------------------------------------------|
| @Html.DisplayFor(modelItem => item.Title)            | @Html.DisplayFor(modelItem => item.ReleaseDate)            | @Html.DisplayFor(modelItem => item.Genre)            | @Html.DisplayFor(modelItem => item.Price)            | @Html.DisplayFor(modelItem => item.Rating)            |


```

使用 `Rating` 字段更新 /Views/Movies/Create.cshtml。

在将 DB 更新为包括新字段之前，应用不会正常工作。如果现在运行，会出现以下 `SqliteException`：

```
SqliteException: SQLite Error 1: 'no such column: m.Rating'.
```

看到此错误是因为更新的 Movie 模型类与现有数据库的 Movie 表架构不同。（数据库表中没有 `Rating` 列。）

可通过几种方法解决此错误：

1. **丢弃数据库**，让 Entity Framework 基于新的模型类架构自动重新创建数据库。如果使用此方法，则会丢失数据库中的现有数据，因此不能对生产数据库使用此方法！使用初始值设定项，以使用测试数据自动设定数据库种子，这通常是开发应用的有效方式。
2. 手动修改现有数据库的架构，使它与模型类匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。
3. 使用 **Code First** 迁移更新数据库架构。

在本教程中，我们将在架构更改时丢弃并重新创建数据库。从终端运行以下命令丢弃 db：

```
dotnet ef database drop
```

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 做出此更改。

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

将 `Rating` 字段添加到 `Edit`、`Details` 和 `Delete` 视图。

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。模板。

上一篇 - 添加搜索

下一篇 - 添加验证

添加验证

2018/5/8 • 11 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Movie` 模型添加验证逻辑, 并确保每当用户创建或编辑电影时, 都会强制执行验证规则。

坚持 DRY 原则

MVC 的设计原则之一是 `DRY` (“不要自我重复”)。ASP.NET MVC 支持你仅指定一次功能或行为, 然后使它应用到整个应用中。这可以减少所需编写的代码量, 并使编写的代码更少出错, 更易于测试和维护。

MVC 和 Entity Framework Core Code First 提供的验证支持是 DRY 原则在实际操作中的极佳示例。可以在一个位置(模型类中)以声明方式指定验证规则, 并且在应用中的所有位置强制执行。

将验证规则添加到电影模型

打开 `Movie.cs` 文件。DataAnnotations 提供一组内置验证特性, 可通过声明方式应用于任何类或属性。(它还包含 `DataType` 等格式特性, 这些特性可帮助进行格式设置, 但不提供任何验证。)

更新 `Movie` 类以使用内置的 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'''\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性指定要对应用这些特性的模型属性强制执行的行为。`Required` 和 `MinimumLength` 特性表示属性必须有值; 但用户可输入空格来满足此验证。`RegularExpression` 特性用于限制可输入的字符。在上述代码中, `Genre` 和 `Rating` 仅可使用字母(禁用空格、数字和特殊字符)。`Range` 特性将值限制在指定范围内。`StringLength` 特性使你能够设置字符串属性的最大长度, 以及可选的最小长度。从本质上来说, 需要值类型(如 `decimal`、`int`、`float`、`DateTime`), 但不需要 `[Required]` 特性。

让 ASP.NET 强制自动执行验证规则有助于提升应用的可靠性。同时它能确保你无法忘记验证某些内容, 并防止你

无意中将错误数据导入数据库。

MVC 中的验证错误 UI

运行应用并导航到电影控制器。

点击“新建”连接添加新电影的链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

Create

Movie

Title

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

Release Date

mm/dd/yyyy

Genre

The Genre field is required.

Price

z

The field Price must be a number.

Rating

The field Rating must match the regular expression '^([A-Z]+[a-zA-Z-\\s]*\$)'.

Create

Back to List

© 2017 - MvcMovie

注意

可能无法在 `Price` 字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 [GitHub 问题 4076](#)。

请注意表单如何自动呈现每个包含无效值的字段中相应的验证错误消息。客户端（使用 JavaScript 和 jQuery）和服务器端（若用户禁用 JavaScript）都必定会遇到这些错误。

明显的好处在于不需要在 `MoviesController` 类或 `Create.cshtml` 视图中更改单个代码行来启用此验证 UI。在本教程前面创建的控制器和视图会自动选取验证规则，这些规则是通过在 `Movie` 模型类的属性上使用验证特性所指定的。使用 `Edit` 操作方法测试验证后，即已应用相同的验证。

存在客户端验证错误时，不会将表单数据发送到服务器。可通过使用 [Fiddler 工具](#)或[F12 开发人员工具](#)在 `HTTP Post` 方法中设置断点来对此进行验证。

验证工作原理

你可能想知道在不对控制器或视图中的代码进行任何更新的情况下，验证 UI 是如何生成的。下列代码显示两种 `Create` 方法。

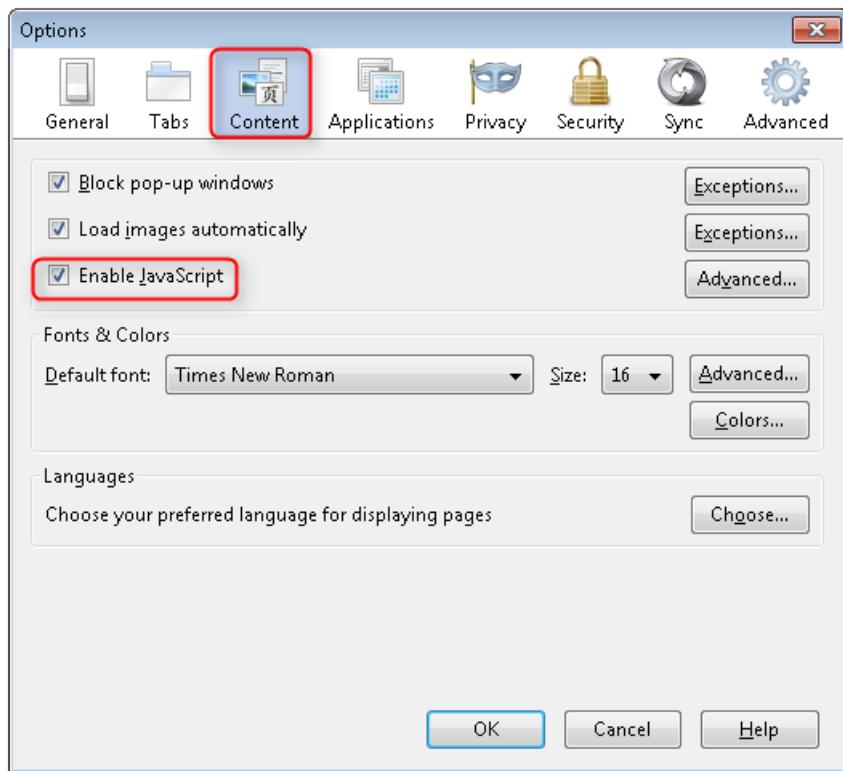
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

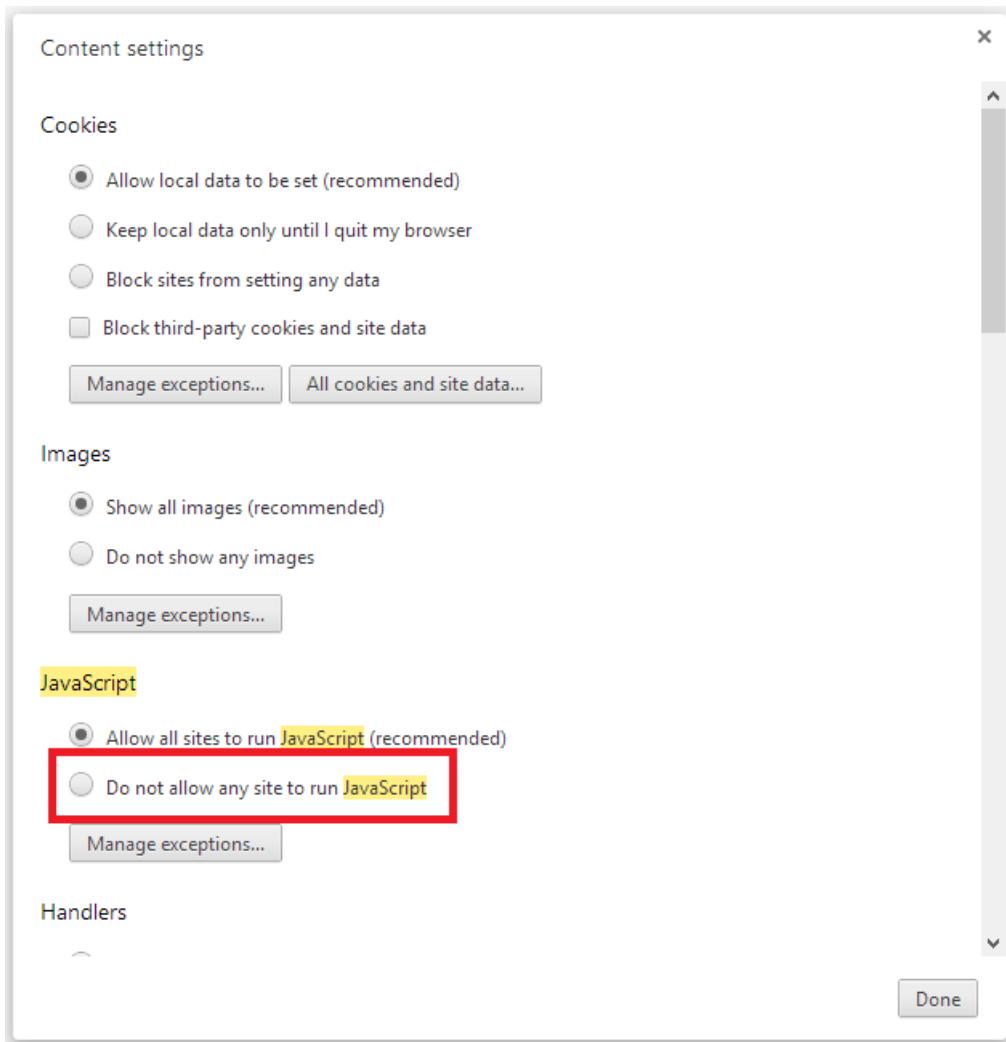
第一个 (HTTP GET) `Create` 操作方法显示初始的“创建”表单。第二个 (`[HttpPost]`) 版本处理表单发布。第二个 `Create` 方法 (`[HttpPost]` 版本) 调用 `ModelState.IsValid` 以检查电影是否有任何验证错误。调用此方法将评估已应用于对象的任何验证特性。如果对象有验证错误，则 `Create` 方法会重新显示此表单。如果没有错误，此方法则将新电影保存在数据库中。在我们的电影示例中，在检测到客户端上存在验证错误时，表单不会发布到服务器。当存在客户端验证错误时，第二个 `Create` 方法永远不会被调用。如果在浏览器中禁用 JavaScript，客户端验证将被禁用，而你可以测试 HTTP POST `Create` 方法 `ModelState.IsValid` 检测任何验证错误。

可以在 `[HttpPost] Create` 方法中设置断点，并验证方法从未被调用，客户端验证在检测到存在验证错误时不会提交表单数据。如果在浏览器中禁用 JavaScript，然后提交错误的表单，将触发断点。在没有 JavaScript 的情况下仍然可以进行完整的验证。

以下图片显示如何在 FireFox 浏览器中禁用 JavaScript。



以下图片显示如何在 Chrome 浏览器中禁用 JavaScript。



禁用 JavaScript 后，发布无效数据并单步执行调试程序。

```
74     // POST: Movies/Create
75     // To protect from overposting attacks, please enable this
76     // more details see http://go.microsoft.com/fwlink/?LinkID=142072
77     [HttpPost]
78     [ValidateAntiForgeryToken]
79     0 references
80     public async Task<IActionResult> Create([Bind("ID,Title")]
81     {
82         if (ModelState.IsValid)
83         {
84             _context.Add(movie);
85             await _context.SaveChangesAsync();
86             return RedirectToAction("Index");
87         }
88         return View(movie);
89     }

```

以下是之前在本教程中已搭建基架的 Create.cshtml 视图模板的一部分。以上所示的操作方法使用它来显示初始表单，并在发生错误时重新显示此表单。

```
<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>

        @*Markup removed for brevity.*@
    </div>
</form>
```

输入标记帮助程序使用 [DataAnnotations](#) 特性，并在客户端上生成 jQuery 验证所需的 HTML 特性。[验证标记帮助程序](#) 用于显示验证错误。有关详细信息，请参阅[验证](#)。

此方法真正好的一点是：无论是控制器还是 `Create` 视图模板都不知道强制实施的实际验证规则或显示的特定错误消息。仅可在 `Movie` 类中指定验证规则和错误字符串。这些相同的验证规则自动应用于 `Edit` 视图和可能创建用于编辑模型的任何其他视图模板。

需要更改验证逻辑时，可以通过将验证特性添加到模型在同一个位置实现此操作。（在此示例中为 `Movie` 类）。无需担心对应用程序的不同部分所强制执行规则的方式不一致 - 所有验证逻辑都将定义在一个位置并用于整个应用程序。这使代码非常简洁，并且更易于维护和改进。这意味着对 DRY 原则的完全遵守。

使用 `DataType` 特性

打开 `Movie.cs` 文件并检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。我们已经在发布日期和价格字段中应用了 `DataType` 枚举值。以下代码显示具有适当 `DataType` 特性的 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 属性仅提供相关提示来帮助视图引擎设置数据格式（并提供元素/属性，例如向 URL 提供 `<a>` 和向电子邮件提供 ``）。可以使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型，它们不是验证属性。在此示例中，我们只想跟踪日期，而不是时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`（读作 data dash）特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。（你可能不想为某些字段执行此操作—例如对于货币值，你可能不希望文本框中的货币符号可编辑。）

可以单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能（例如显示日历控件、区域设置适用的货币符号、电子邮件链接等）
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- `DataType` 特性使 MVC 能够选择正确的字段模板来呈现数据（如果 `DisplayFormat` 由自身使用，则使用的是字符串模板）。

注意

jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

需要禁用 jQuery 日期验证才能使用具有 `DateTime` 的 `Range` 特性。通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^([A-Z]+[a-zA-Z''"\s-]*)$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^([A-Z]+[a-zA-Z'''\s-]*)$"), StringLength(5)]
    public string Rating { get; set; }
}
```

在本系列的下一部分中，我们将回顾应用程序，并对自动生成的 `Details` 和 `Delete` 方法进行一些改进。

其他资源

- [使用表单](#)
- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)

[上一篇 - 添加字段](#)

[下一篇 - 检查详细信息和删除方法](#)

检查 ASP.NET Core 应用的 Details 和 Delete 方法

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

打开电影控制器，并检查 `Details` 方法：

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

创建此操作方法的 MVC 基架引擎添加显示调用方法的 HTTP 请求的注释。在此情况下，它是包含三个 URL 段的 GET 请求，这三个段为 `Movies` 控制器、`Details` 方法和 `id` 值。回顾这些在 `Startup.cs` 中定义的段。

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

EF 可以使用 `SingleOrDefaultAsync` 方法轻松搜索数据。该方法中内置的一个重要安全功能是，代码会先验证搜索方法已经找到电影，然后再执行操作。例如，一个黑客可能通过将链接创建的 URL 从 `http://localhost:xxxx/Movies/Details/1` 更改为类似 `http://localhost:xxxx/Movies/Details/12345` 的值（或者不代表任何实际电影的其他值）将错误引入站点。如果未检查是否有空电影，则应用可能引发异常。

检查 `Delete` 和 `DeleteConfirmed` 方法。

```

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

请注意，`HTTP GET Delete` 方法不删除指定的电影，而是返回可在其中提交 (HttpPost) 删除的电影视图。执行删除操作以响应 GET 请求(或者说，执行编辑操作、创建操作或更改数据的任何其他操作)会打开安全漏洞。

删除数据的 `[HttpPost]` 方法命名为 `DeleteConfirmed`，以便为 HTTP POST 方法提供一个唯一的签名或名称。下面显示了两个方法签名：

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

公共语言运行时 (CLR) 需要重载方法拥有唯一的参数签名(相同的方法名称但不同的参数列表)。但是，这里需要两个 `Delete` 方法 -- 一个用于 GET，另一个用于 POST -- 这两个方法拥有相同的参数签名。(它们都需要接受单个整数作为参数。)

可通过两种方法解决此问题，一种是为方法提供不同的名称。这正是前面的示例中的基架机制进行的操作。但是，这会造成一个小问题：ASP.NET 按名称将 URL 段映射到操作方法，如果重命名方法，则路由通常无法找到该方法。该示例中也提供了解决方案，即向 `DeleteConfirmed` 方法添加 `ActionName("Delete")` 属性。该属性对路由系统执行映射，以便包括 POST 请求的 /Delete/ 的 URL 可找到 `DeleteConfirmed` 方法。

对于名称和签名相同的方法，另一个常用解决方法是手动更改 POST 方法的签名以包括额外(未使用)的参数。这正是前面的文章中添加 `notUsed` 参数时进行的操作。这里为了 `[HttpPost] Delete` 方法可以执行同样的操作：

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

发布到 Azure

有关如何使用 Visual Studio 将该应用发布到 Azure 的说明, 请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。此外, 还可以从[命令行](#)发布应用。

感谢读完这篇 ASP.NET Core MVC 简介。我们期待你的意见。[MVC 和 EF Core 入门](#)是本教程的优选后续教程。

[上一篇](#)

使用 Visual Studio Code 创建 ASP.NET Core MVC 应用

2018/4/10 • 1 min to read • [Edit Online](#)

本系列教程介绍了使用 Visual Studio Code 生成 ASP.NET Core MVC Web 应用所涉及的基础知识。

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#)，再使用 MVC 版本。Razor 页面教程：

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面](#)版本选择本教程，请在[此 GitHub 问题](#)中说明原因。

1. [入门](#)
2. [添加控制器](#)
3. [添加视图](#)
4. [添加模型](#)
5. [使用 SQLite](#)
6. [控制器方法和视图](#)
7. [添加搜索](#)
8. [添加新字段](#)
9. [添加验证](#)
10. [检查 Details 和 Delete 方法](#)

macOS、Linux 或 Windows 上的 ASP.NET Core MVC 简介

2018/5/14 • 2 min to read • [Edit Online](#)

作者: Rick Anderson

本教程将介绍基础知识，指导你使用 [Visual Studio Code \(VS Code\)](#) 构建 ASP.NET Core MVC Web 应用。本教程假定用户熟悉 VS Code。有关详细信息，请参阅 [VS Code 入门](#) 和 [Visual Studio Code 帮助](#)。

本教程介绍具有控制器和视图的 ASP.NET Core MVC Web 开发。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议先尝试 [Razor 页面教程](#)，再使用 MVC 版本。[Razor 页面教程](#):

- 是开发新应用程序的首选方法。
- 易于关注。
- 涵盖更多功能。

如果通过 [Razor 页面](#) 版本选择本教程，请在[此 GitHub 问题](#)中说明原因。

本教程提供 3 个版本：

- macOS: [使用 Visual Studio for Mac 创建 ASP.NET Core MVC 应用](#)
- Windows: [使用 Visual Studio 创建 ASP.NET Core MVC 应用](#)
- macOS、Linux 和 Windows: [使用 Visual Studio Code 创建 ASP.NET Core MVC 应用](#)

系统必备

Install the following:

- [.NET Core SDK 2.0 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)
- [.NET Core SDK 2.1 RC1 or later](#)
- [Visual Studio Code](#)
- [C# for Visual Studio Code](#)

通过 dotnet 创建 Web 应用

从终端运行以下命令：

```
mkdir MvcMovie  
cd MvcMovie  
dotnet new mvc
```

在 Visual Studio Code (VS Code) 中打开“MvcMovie”文件夹并选择“Startup.cs”文件。

- 对于警告消息 -““MvcMovie”中缺少进行生成和调试所需的资产。是否添加它们？”，请选择“是”
- 对于信息性消息 -“存在未解析的依赖项”，请选择“还原”。

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Builder;
6  using Microsoft.AspNetCore.Hosting;
7  using Microsoft.AspNetCore.HttpsPolicy;
8  using Microsoft.AspNetCore.Mvc;
9  using Microsoft.Extensions.Configuration;
10 using Microsoft.Extensions.DependencyInjection;
11 using Microsoft.Extensions.Logging;
12 using Microsoft.Extensions.Options;
13
14 namespace TodoApi
15 {
16     public class Startup
17     {
18         ...
19     }
20 }
```

按“调试”(F5) 生成并运行程序。

Home Page - MvcMovie

localhost:5000

MvcMovie

Microsoft Azure |

Application uses

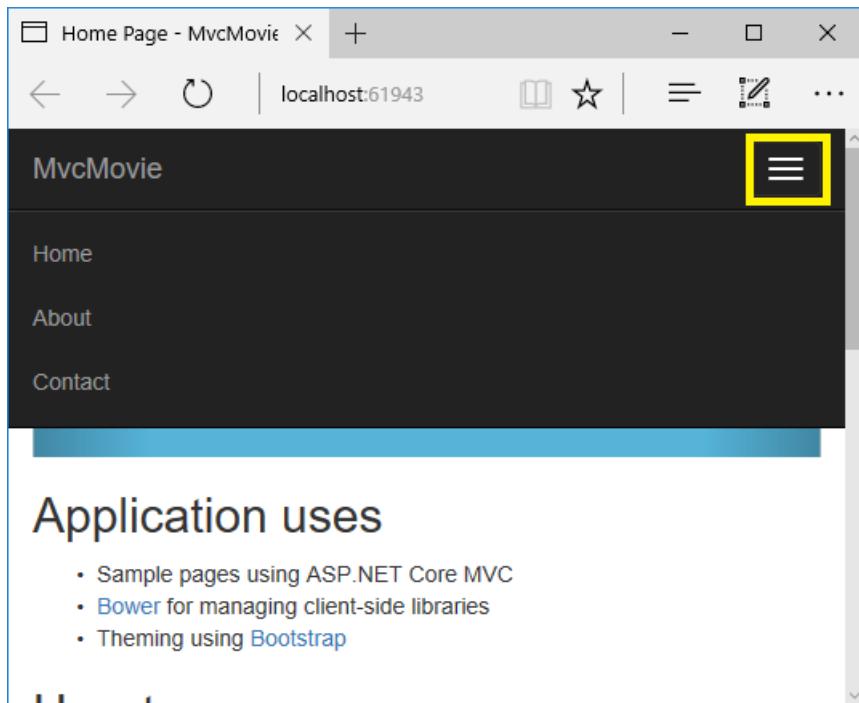
- Sample pages using ASP.NET Core MVC
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

- Add a Controller and View
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet

VS Code 启动 Kestrel Web 服务器并运行应用。请注意，地址栏显示 `localhost:5000` 而非 `example.com` 等内容。这是因为 `localhost` 是本地计算机的标准主机名。

默认模板提供可用的“主页”、“关于”和“联系”链接。上面的浏览器图像未显示这些链接。根据浏览器的大小，可能需要单击导航图标才能显示这些链接。



在本教程的下一部分中，我们将了解 MVC 并开始编写一些代码。

Visual Studio Code 帮助

- [入门](#)
- [调试](#)
- [集成终端](#)
- [键盘快捷键
 - \[macOS 键盘快捷方式\]\(#\)
 - \[Linux 键盘快捷键\]\(#\)
 - \[Windows 键盘快捷键\]\(#\)](#)

[下一篇 - 添加控制
器](#)

将控制器添加到 ASP.NET Core 应用

2018/5/14 • 7 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

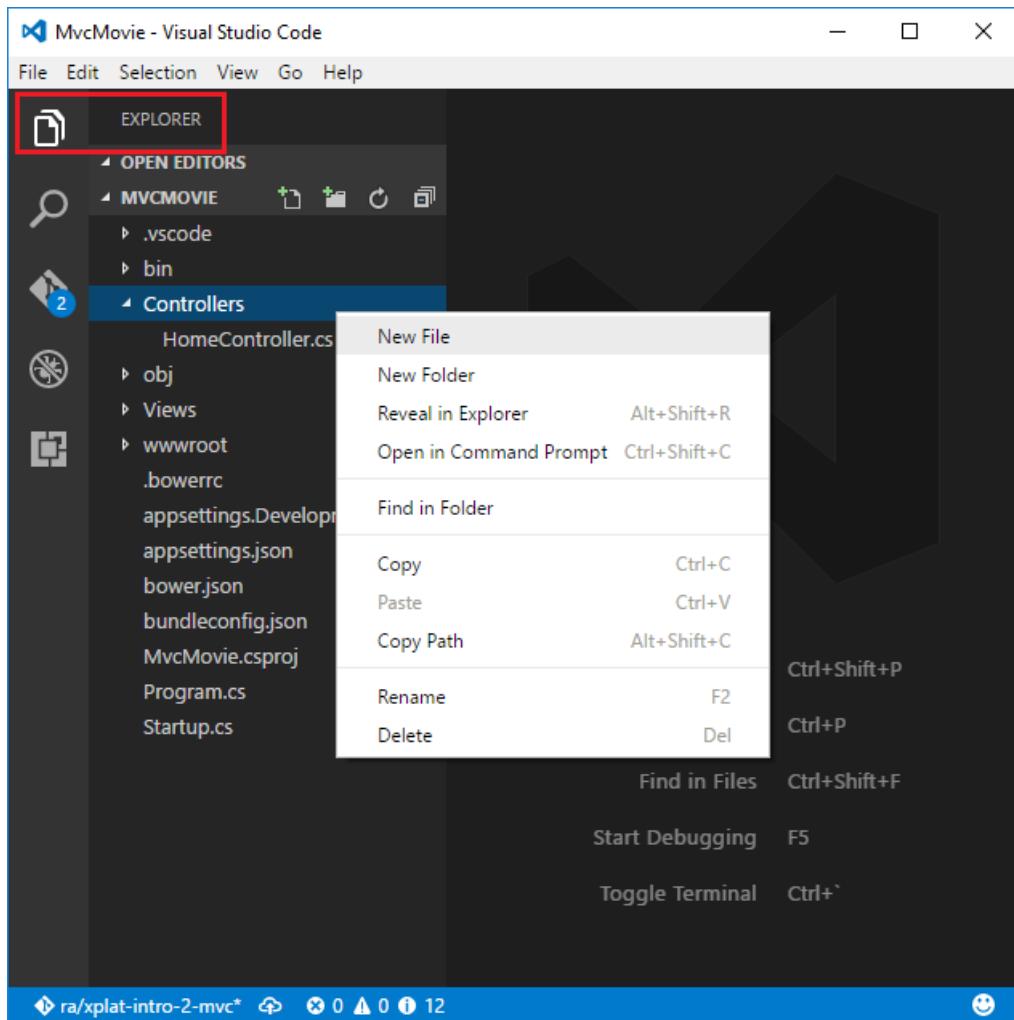
模型-视图-控制器 (MVC) 体系结构模式将应用分成 3 个主要组件: 模型 (M)、视图 (V) 和控制器 (C)。MVC 模式有助于创建比传统单片应用更易于测试和更新的应用。基于 MVC 的应用包含:

- **模式:** 表示应用数据的类。模型类使用验证逻辑来对该数据强制实施业务规则。通常, 模型对象检索模型状态并将其存储在数据库中。本教程中, `Movie` 模型将从数据库中检索电影数据, 并将其提供给视图或对其进行更新。更新后的数据将写入到数据库。
- **视图:** 视图是显示应用用户界面 (UI) 的组件。此 UI 通常会显示模型数据。
- **控制器:** 处理浏览器请求的类。它们检索模型数据并调用返回响应的视图模板。在 MVC 应用中, 视图仅显示信息; 控制器处理并响应用户输入和交互。例如, 控制器处理路由数据和查询字符串值, 并将这些值传递给模型。该模型可使用这些值查询数据库。例如, `http://localhost:1234/Home/About` 具有 `Home` (控制器) 的路由数据和 `About` (在 `Home` 控制器上调用的操作方法)。`http://localhost:1234/Movies/Edit/5` 是一个请求, 用于通过电影控制器编辑 ID 为 5 的电影。本教程稍后将探讨路由数据。

MVC 模式可帮助创建分隔不同应用特性(输入逻辑、业务逻辑和 UI 逻辑)的应用, 同时让这些元素之间实现松散耦合。该模式可指定应用中每种逻辑的位置。UI 逻辑位于视图中。输入逻辑位于控制器中。业务逻辑位于模型中。这种隔离有助于控制构建应用时的复杂程度, 因为它可用于一次处理一个实现特性, 而不影响其他特性的代码。例如, 处理视图代码时不必依赖业务逻辑代码。

本教程系列中介绍了这些概念, 并展示了如何使用它们构建电影应用。MVC 项目包含“控制器”和“视图”文件夹。

- 在“VS Code”中, 选择“EXPLORER”图标, 然后在按住 Control 的同时单击(右键单击)“控制器”, 选择“新建文件”, 然后将新文件命名为 `HelloWorldController.cs`。



将“Controllers/HelloWorldController.cs”的内容替换为以下内容：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

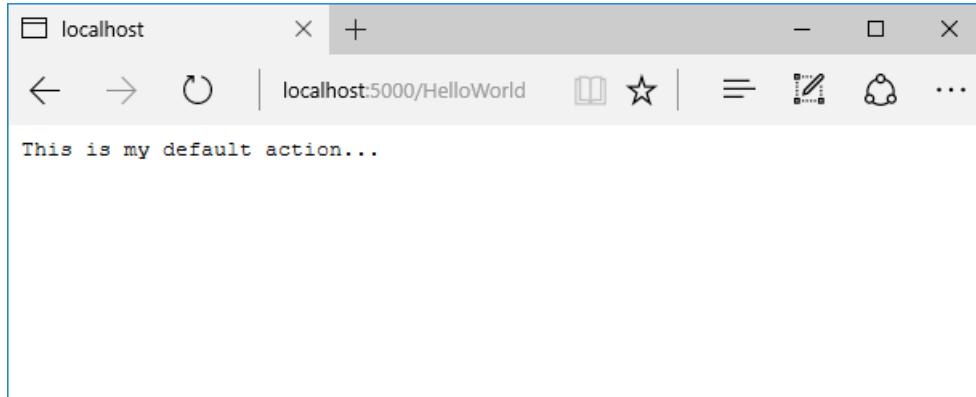
控制器中的每个 `public` 方法均可作为 HTTP 终结点调用。上述示例中，两种方法均返回一个字符串。请注意每个方法前面的注释。

HTTP 终结点是 Web 应用程序中可定向的 URL(例如 `http://localhost:1234/HelloWorld`)，其中结合了所用的协议

`HTTP`、`TCP` 端口等 Web 服务器的网络位置 `localhost:1234`，以及目标 URI `HelloWorld`。

第一条注释指出这是一个 `HTTP GET` 方法，它通过向基 URL 追加“/HelloWorld/”进行调用。第二条注释指定一个 `HTTP GET` 方法，它通过向 URL 追加“/HelloWorld/Welcome/”进行调用。本教程稍后将使用基架引擎生成 `HTTP POST` 方法。

在非调试模式下运行应用，并将“HelloWorld”追加到地址栏中的路径。`Index` 方法返回一个字符串。



MVC 根据入站 URL 调用控制器类(及其中的操作方法)。MVC 所用的默认 URL 路由逻辑使用如下格式来确定调用的代码：

`/[Controller]/[ActionName]/[Parameters]`

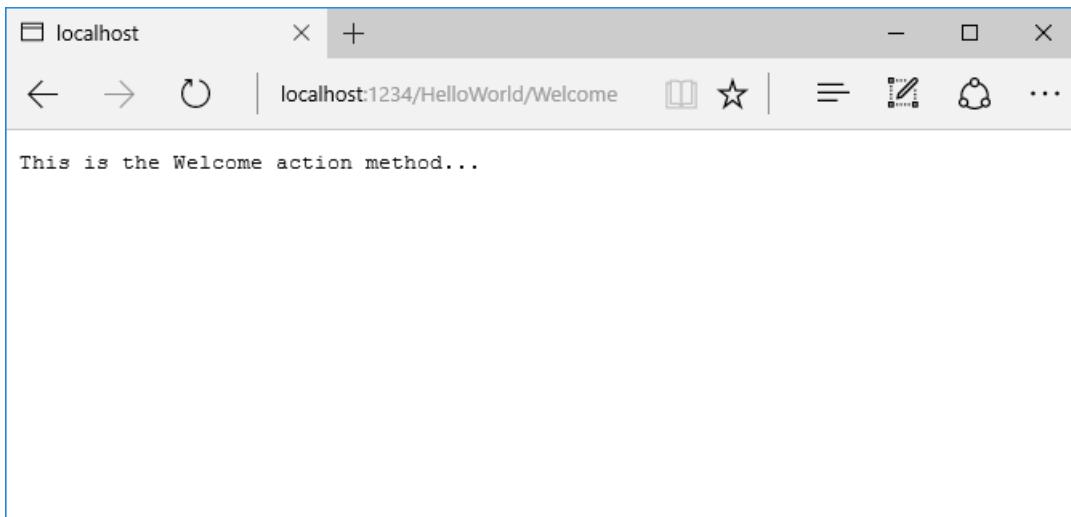
在 `Startup.cs` 文件的 `Configure` 方法中设置路由的格式。

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

如果运行应用且不提供任何 URL 段，它将默认为上面突出显示的模板行中指定的“Home”控制器和“Index”方法。

第一个 URL 段决定要运行的控制器类。因此 `localhost:xxxx/HelloWorld` 映射到 `HelloWorldController` 类。该 URL 段的第二部分决定类上的操作方法。因此 `localhost:xxxx/HelloWorld/Index` 将触发 `HelloWorldController` 类的 `Index` 运行。请注意，只需浏览到 `localhost:xxxx/HelloWorld`，而 `Index` 方法默认调用。原因是 `Index` 是默认方法，如果未显式指定方法名称，则将在控制器上调用它。URL 段的第三部分 (`id`) 针对的是路由数据。稍后可在本教程中看到路由数据。

浏览到 `http://localhost:xxxx/HelloWorld/Welcome`。`Welcome` 方法运行并返回字符串“这是 Welcome 操作方法...”。对于此 URL，采用 `HelloWorld` 控制器和 `Welcome` 操作方法。目前尚未使用 URL 的 `[Parameters]` 部分。



修改代码，将一些参数信息从 URL 传递到控制器。例如 `/HelloWorld/Welcome?name=Rick&numtimes=4`。更改 `Welcome` 方法以包括以下代码中显示的两个参数：

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

前面的代码：

- 使用 C# 可选参数功能指示，未为 `numTimes` 参数传递值时该参数默认为 1。
- 使用 `HtmlEncoder.Default.Encode` 防止恶意输入（即 JavaScript）损害应用。
- 使用内插字符串。

运行应用并浏览到：

```
http://localhost:xxxx>HelloWorld/Welcome?name=Rick&numtimes=4
```

（将 xxxx 替换为端口号。）可在 URL 中对 `name` 和 `numtimes` 使用其他值。MVC 模型绑定系统可将命名参数从地址栏中的查询字符串自动映射到方法中的参数。有关详细信息，请参阅[模型绑定](#)。

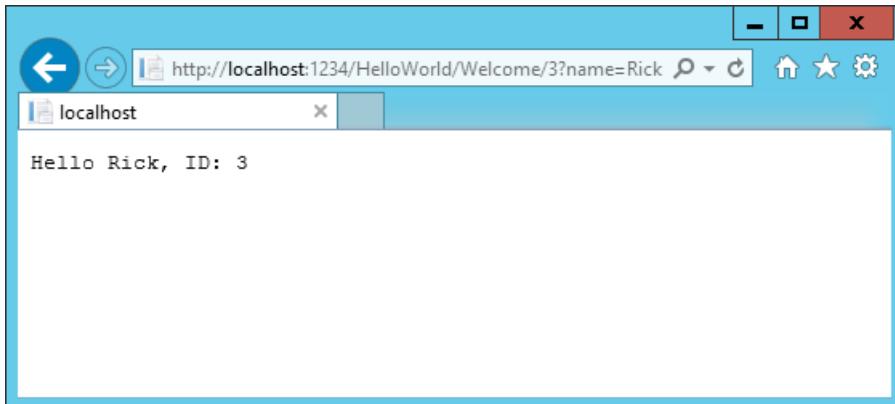


在上图中，未使用 URL 段 (`Parameters`)，且 `name` 和 `numTimes` 参数作为[查询字符串](#)进行传递。上述 URL 中的 `?`（问号）为分隔符，后接查询字符串。`&` 字符用于分隔查询字符串。

将 `Welcome` 方法替换为以下代码：

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

运行应用并输入以下 URL: <http://localhost:xxx/HelloWorld/Welcome/3?name=Rick>



此时, 第三个 URL 段与路由参数 `id` 相匹配。`Welcome` 方法包含 `MapRoute` 方法中匹配 URL 模板的参数 `id`。后面的 `?(id?)` 中表示 `id` 参数可选。

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

上述示例中, 控制器始终执行 MVC 的“VC”部分, 即视图和控制器工作。控制器将直接返回 HTML。通常不希望控制器直接返回 HTML, 因为编码和维护非常繁琐。通常, 需使用单独的 Razor 视图模板文件来帮助生成 HTML 响应。可在下一教程中执行该操作。

[上一篇 - 添加控制器](#) [下一篇 - 添加视图](#)

将视图添加到 ASP.NET Core MVC 应用

2018/5/8 • 9 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将修改 `HelloWorldController` 类, 进而使用 Razor 视图模板文件来顺利封装为客户端生成 HTML 响应的过程。

使用 Razor 创建视图模板文件。基于 Razor 的模板具有".cshtml"文件扩展名。它们提供了一种巧妙的方法来使用 C# 创建 HTML 输出。

当前, `Index` 方法返回带有在控制器类中硬编码的消息的字符串。在 `HelloWorldController` 类中, 将 `Index` 方法替换为以下代码:

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

上述代码返回 `View` 对象。它使用视图模板对浏览器生成 HTML 响应。类似上述 `Index` 方法的控制器方法(也称为操作方法)通常返回 `IActionResult`(或派生自 `ActionResult` 的类), 而非类似字符串的类型。

为 `HelloWorldController` 添加 `Index` 视图。

- 添加一个名为“Views/HelloWorld”的新文件夹。
- 向 `Views/HelloWorld` 文件夹添加名为“`Index.cshtml`”的新文件。

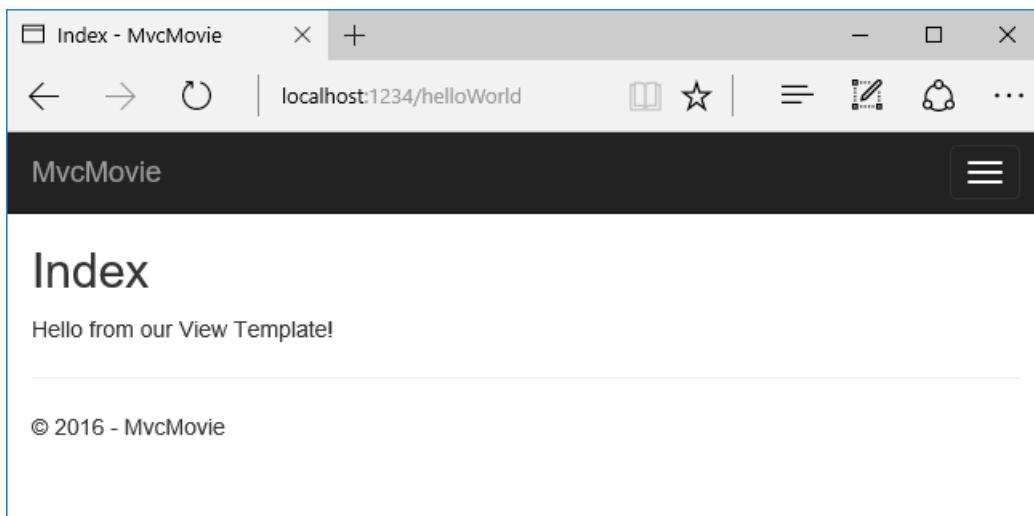
使用以下内容替换 Razor 视图文件 `Views/HelloWorld/Index.cshtml` 的内容:

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

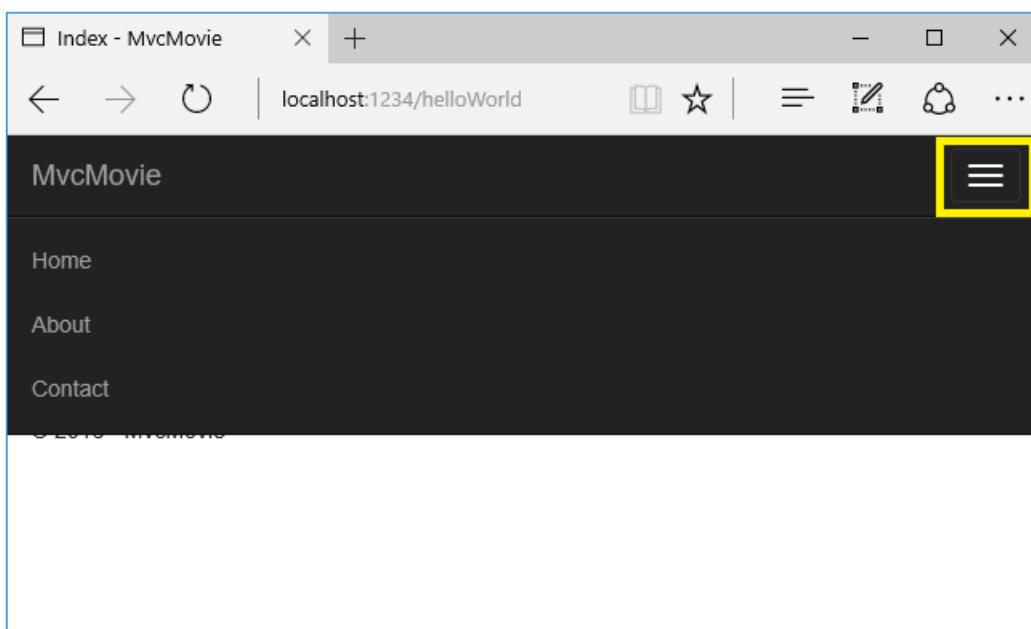
<h2>Index</h2>

<p>Hello from our View Template!</p>
```

导航到 `http://localhost:xxxx/HelloWorld`。`HelloWorldController` 中的 `Index` 方法作用不大; 它运行 `return View();` 语句, 指定此方法应使用视图模板文件来呈现对浏览器的响应。因为没有显式指定视图模板文件的名称, 所以 MVC 默认使用 `/Views/HelloWorld` 文件夹中的 `Index.cshtml` 视图文件。下面图片显示了视图中硬编写的字符串“Hello from our View Template!”



如果浏览器窗口较小(例如在移动设备上), 则可能需要在右上角切换(点击)Bootstrap 导航按钮以查看“首页”、“关于”和“联系人”链接。



更改视图和布局页面

点击菜单链接(“MvcMovie”、“首页”、“关于”)。每页显示相同的菜单布局。菜单布局是在 Views/Shared/_Layout.cshtml 文件中实现的。打开 Views/Shared/_Layout.cshtml 文件。

布局模板使你能够在一个位置指定网站的 HTML 容器布局, 然后将它应用到网站中的多个页面。查找 `@RenderBody()` 行。`RenderBody` 是显示创建的所有特定于视图的页面的占位符, 已包装在布局页面中。例如, 如果选择“关于”链接, Views/Home/About.cshtml 视图将在 `RenderBody` 方法中呈现。

更改布局文件中的标题和菜单链接

在标题元素中, 将 `MvcMovie` 更改为 `Movie App`。将布局模板中的定位文本从 `MvcMovie` 更改为 `Movie App`, 并将控制器从 `Home` 更改为 `Movies`, 如下所示:

注意:ASP.NET Core 2.0 版本略有不同。它不包含 `@inject ApplicationInsights` 和 `@Html.Raw(JavaScriptSnippet.FullScript)`。

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
```

```

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
              value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
    @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Movies" asp-action="Index" class="navbar-brand">Movie App</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2017 - MvcMovie</p>
        </footer>
    </div>

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment names="Staging,Production">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
               asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
               asp-fallback-test="window.jQuery"
               crossorigin="anonymous"
               integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHII9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
               asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
               asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
               crossorigin="anonymous"
               integrity="sha384-Tc5IQib027qvjSMfHj0MaLkfWVxZxUPnCJA712mCWNIPG9mGCD8wGNICPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

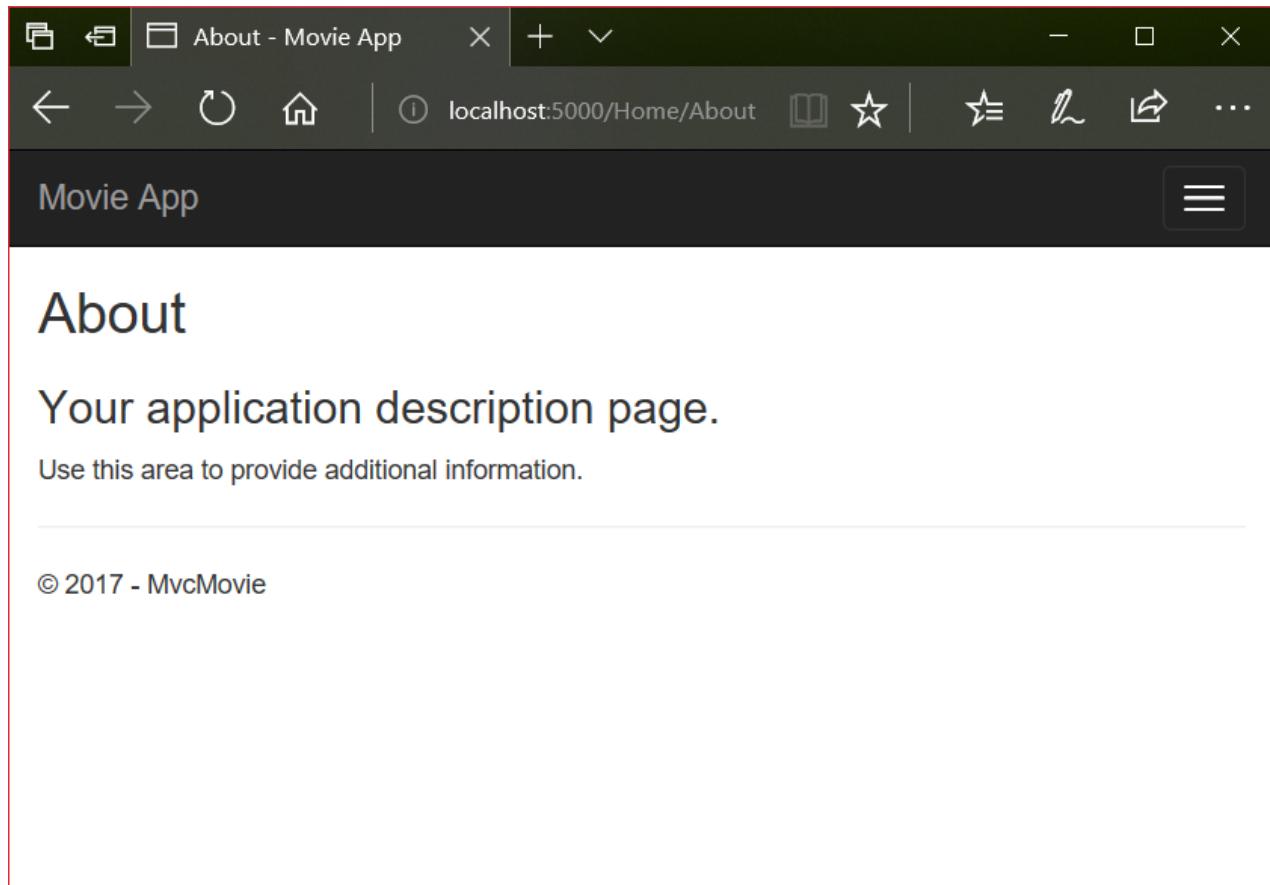
```

```
@RenderSection("Scripts", required: false)
</body>
</html>
```

警告

我们尚未实现 `Movies` 控制器，所以如果单击此链接，将收到 404(未找到)错误。

保存更改并点击“关于”链接。请注意浏览器选项卡上的标题现在显示的是“关于 - 电影应用”，而不是“关于 - Mvc 电影”：



点击“联系人”链接，请注意，标题和定位文本还会显示“电影应用”。我们能够在布局模板中进行一次更改，让网站上的所有页面都反映新的链接文本和新标题。

检查 `Views/_ViewStart.cshtml` 文件：

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

`Views/_ViewStart.cshtml` 文件将 `Views/Shared/_Layout.cshtml` 文件引入到每个视图中。可以使用 `Layout` 属性设置不同的布局视图，或将它设置为 `null`，这样将不会使用任何布局文件。

更改 `Index` 视图的标题。

打开 `Views/HelloWorld/Index.cshtml`。有两处需要更改的地方：

- 浏览器标题中显示的文字。
- 辅助标题(`<h2>` 元素)。

稍稍对它们进行一些更改，这样可以看出哪一段代码更改了应用的哪部分。

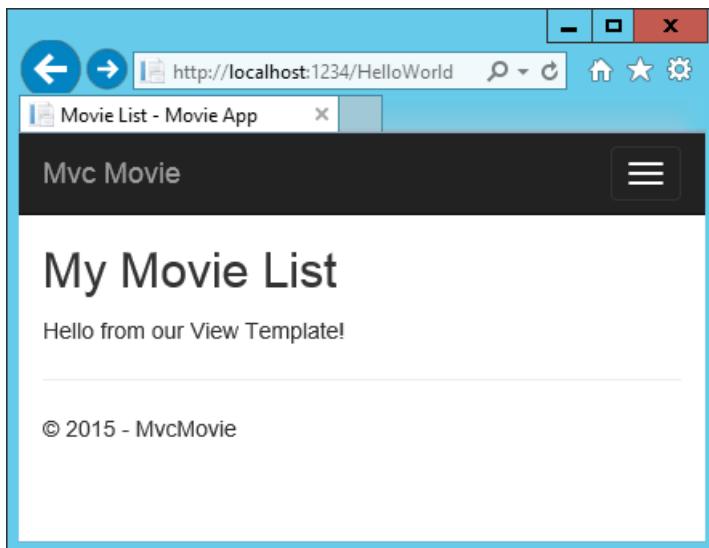
```
@{  
    ViewData["Title"] = "Movie List";  
}  
  
<h2>My Movie List</h2>  
  
<p>Hello from our View Template!</p>
```

上述代码中的 `ViewData["Title"] = "Movie List";` 将 `ViewData` 字典的 `Title` 属性设置为“Movie List”。`Title` 属性在布局页面中的 `<title>` HTML 元素中使用：

```
<title>@ViewData["Title"] - Movie App</title>
```

保存更改并导航到 `http://localhost:xxxx/HelloWorld`。请注意，浏览器标题、主标题和辅助标题已更改。（如果没有在浏览器中看到更改，则可能正在查看缓存的内容。在浏览器中按 `Ctrl + F5` 强制加载来自服务器的响应。）浏览器标题是使用我们在 `Index.cshtml` 视图模板中设置的 `ViewData["Title"]` 以及在布局文件中添加的额外“- Movie App”创建的。

还要注意，`Index.cshtml` 视图模板中的内容是如何与 `Views/Shared/_Layout.cshtml` 视图模板合并的，并且注意单个 HTML 响应被发送到了浏览器。凭借布局模板可以很容易地对应用程序中所有页面进行更改。若要了解更多信息，请参阅[布局](#)。



但我们这一点点“数据”（在此示例中为“Hello from our View Template!”消息）是硬编码的。MVC 应用程序有一个“V”（视图），而你已有一个“C”（控制器），但还没有“M”（模型）。

将数据从控制器传递给视图

控制器操作会被调用以响应传入的 URL 请求。控制器类是编写处理传入浏览器请求的代码的地方。控制器从数据源检索数据，并决定将哪些类型的响应发送回浏览器。可以从控制器使用视图模板来生成并格式化对浏览器的 HTML 响应。

控制器负责提供所需的数据，使视图模板能够呈现响应。最佳做法：视图模板不应该直接执行业务逻辑或与数据库进行交互。相反，视图模板应仅使用由控制器提供给它的数据。保持此“关注点分离”有助于保持代码干净、可测试性和可维护性。

目前，`HelloWorldController` 类中的 `Welcome` 方法采用 `name` 和 `ID` 参数，然后将值直接输出到浏览器。应将控制器更改为使用视图模板，而不是使控制器将此响应呈现为字符串。视图模板会生成动态响应，这意味着必须将适当的数据位从控制器传递给视图以生成响应。为此，可以让控制器将视图模板所需的动态数据（参数）放置在视图模板稍后可以访问的 `ViewData` 字典中。

返回到 HelloWorldController.cs 文件，并更改 `Welcome` 方法以将 `Message` 和 `NumTimes` 值添加到 `ViewData` 字典。`ViewData` 字典是一个动态对象，这意味着你可以将任何所需的内容放在其中；只有将内容放在其中后 `ViewData` 对象才具有定义的属性。MVC 模型绑定系统自动将命名参数（`name` 和 `numTimes`）从地址栏中的查询字符串映射到方法中的参数。完整的 HelloWorldController.cs 文件如下所示：

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

`ViewData` 字典对象包含将传递给视图的数据。

创建一个名为 Views/HelloWorld/Welcome.cshtml 的 Welcome 视图模板。

在 Welcome.cshtml 视图模板中创建一个循环，显示“Hello” `NumTimes`。使用以下内容替换 Views/HelloWorld/Welcome.cshtml 的内容：

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

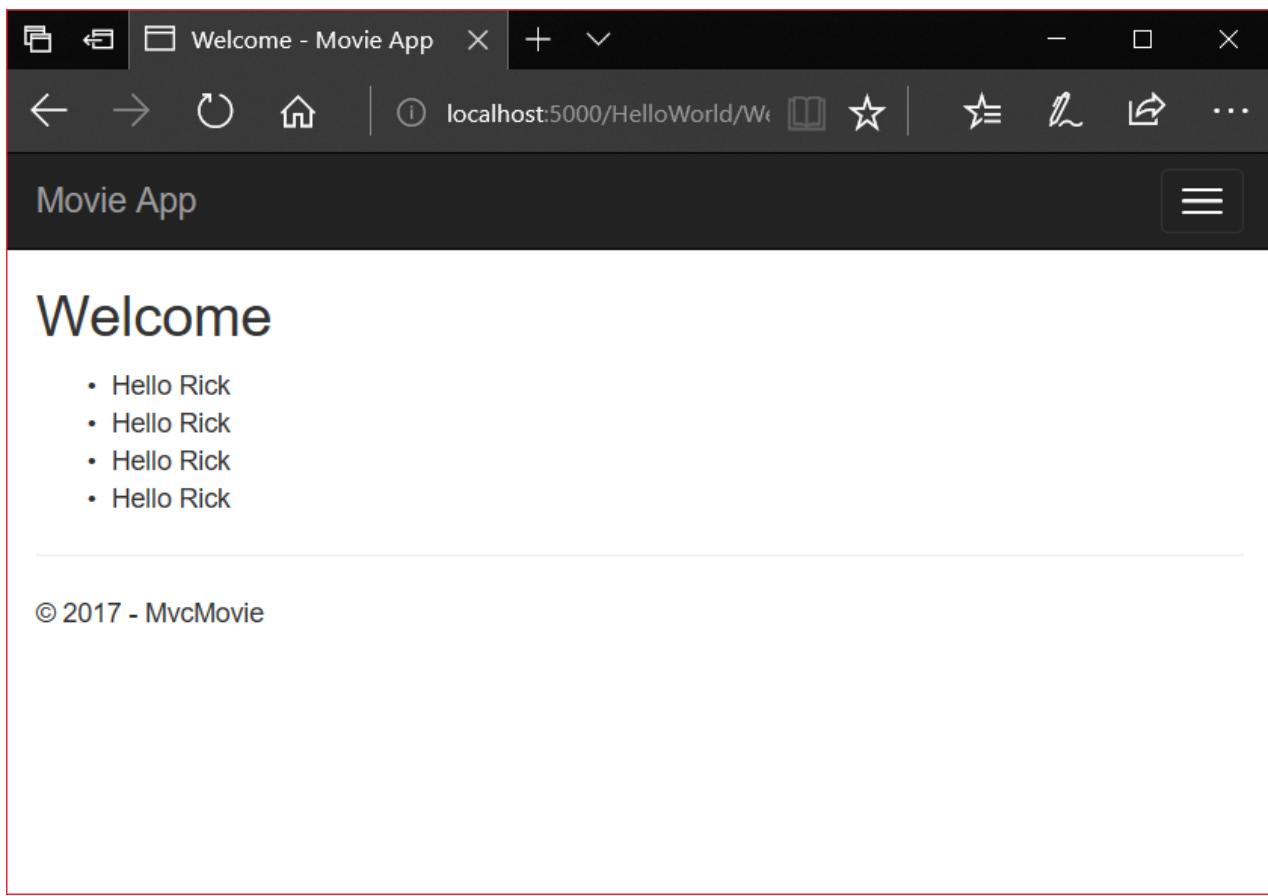
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

保存更改并浏览到以下 URL：

```
http://localhost:xxxx>HelloWorld/Welcome?name=Rick&numtimes=4
```

数据取自 URL，并传递给使用 MVC 模型绑定器的控制器。控制器将数据打包到 `ViewData` 字典中，并将该对象传递给视图。然后，视图将数据作为 HTML 呈现给浏览器。



在上面的示例中，我们使用 `ViewData` 字典将数据从控制器传递给视图。稍后在本教程中，我们将使用视图模型将数据从控制器传递给视图。传递数据的视图模型方法通常比 `ViewData` 字典方法更为优先。有关详细信息，请参阅 [ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC](#) (MVC 中 ViewModel、ViewData、ViewBag、TempData 和 Session 之间的比较)。

当然，这是模型的一种“M”类型，而不是数据库类。让我们用学到的内容来创建一个电影数据库。

[上一篇 - 添加控制器](#) [下一篇 - 添加模型](#)

将模型添加到 ASP.NET Core MVC 应用

2018/5/8 • 5 min to read • [Edit Online](#)

将模型添加到 ASP.NET Core MVC 应用

作者: [Rick Anderson](#) 和 [Tom Dykstra](#)

在本部分中，将添加用于管理数据库中电影的一些类。这些类将是 MVC 应用的“Model”部分。

可以结合 [Entity Framework Core](#) (EF Core) 使用这些类来处理数据库。EF Core 是对象关系映射 (ORM) 框架，可以简化需要编写的数据访问代码。[EF Core 支持许多数据库引擎](#)。

要创建的模型类称为 POCO 类(源自“普通旧 CLR 对象”)，因为它们与 EF Core 没有任何依赖关系。它们只定义将存储在数据库中的数据的属性。

在本教程中，首先将编写模型类，然后 EF Core 将创建数据库。有一种备选方法(此处未介绍)是从已存在的数据库生成模型类。有关此方法的信息，请参阅 [ASP.NET Core - Existing Database](#)(ASP.NET Core - 现有数据库)。

添加数据模型类

- 将类添加到名为“Movie.cs”的“Models”文件夹。
- 将以下代码添加到“Models/Movie.cs”文件：

```
using System;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

数据库需要 `ID` 字段以获取主键。

生成应用以确认没有任何错误，最后将 Model 添加到你的 MVC 应用。

准备项目以搭建基架

- 将以下突出显示的 NuGet 包添加到 MvcMovie.csproj 文件：

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>

```

- 保存该文件，并对信息性消息“存在未解析的依赖项”选择“还原”。
- 创建 Models/MvcMovieContext.cs 文件并添加以下 `MvcMovieContext` 类：

```

using Microsoft.EntityFrameworkCore;

namespace MvcMovie.Models
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
    }
}

```

- 打开 Startup.cs 文件并添加两个 using：

```

using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie
{
    public class Startup
    {

```

- 将数据库上下文添加到 Startup.cs 文件：

```

        public void ConfigureServices(IServiceCollection services)
        {
            // Add framework services.
            services.AddMvc();

            services.AddDbContext<MvcMovieContext>(options =>
                options.UseSqlite("Data Source=MvcMovie.db"));
        }

```

这会告诉实体框架，数据模型中包含哪些模型类。现在定义一个 Movie 对象实体集，此实体集会表示为数据库中的一个 Movie 表。

- 生成项目并确定没有任何错误。

为 MovieController 搭建基架

在项目文件夹中打开终端窗口，然后运行以下命令：

```
dotnet restore  
dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc MvcMovieContext --  
relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries
```

基架引擎创建以下组件：

- 电影控制器 (Controllers/MoviesController.cs)
- “创建”、“删除”、“详细信息”、“编辑”和“索引”页面的 Razor 视图文件 (Views/Movies/*.cshtml)

自动创建 **CRUD** (创建、读取、更新和删除) 操作方法和视图的过程称为“搭建基架”。你很快就会拥有一个功能完备的 Web 应用程序，并且你可以使用它管理电影数据库。

添加初始迁移

从命令行运行以下 .NET Core CLI 命令：

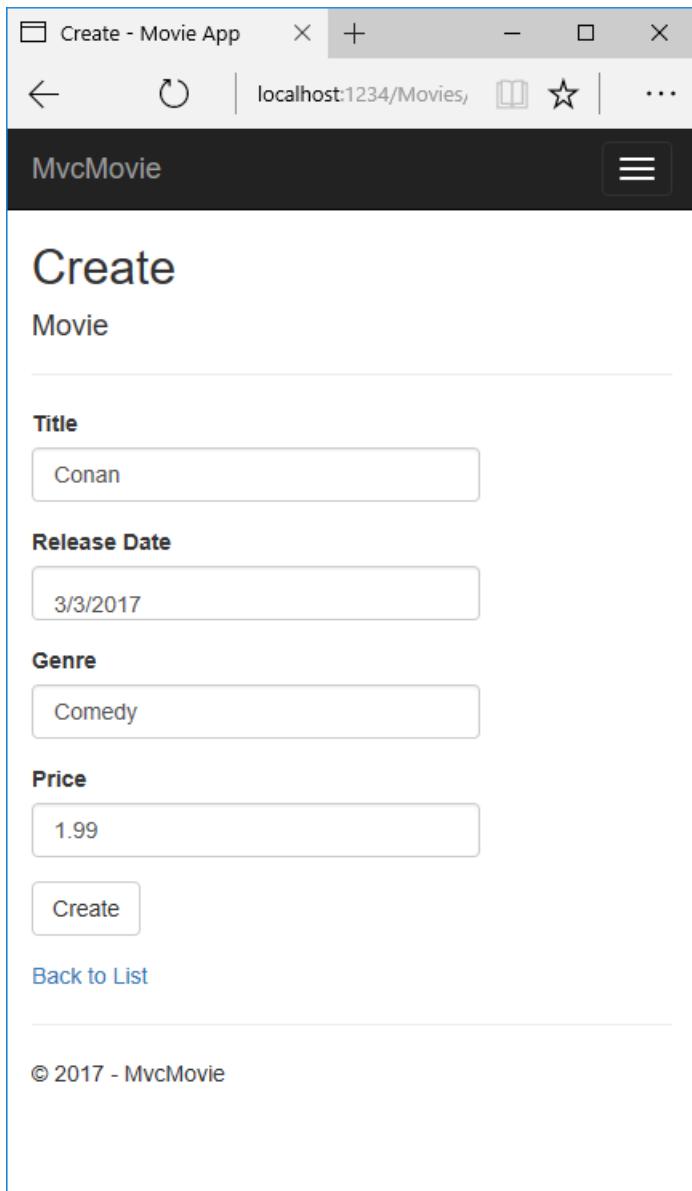
```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

`dotnet ef migrations add InitialCreate` 命令生成用于创建初始数据库架构的代码。此架构以 (Models/MvcMovieContext.cs 文件中的) `DbContext` 中指定的模型为基础。`Initial` 参数用于为迁移命名。可以使用任意名称，但是按照惯例应选择描述迁移的名称。有关详细信息，请参阅[迁移简介](#)。

`dotnet ef database update` 命令在用于创建数据库的 Migrations/<time-stamp>_InitialCreate.cs 文件中运行 `Up` 方法。

测试应用

- 运行应用并点击“Mvc Movie”链接。
- 点击“新建”链接，创建电影。



- 可能无法在 `Price` 字段中输入小数点或逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点及使用非美国英语日期格式的非英语区域设置，必须执行使应用全球化的步骤。请参阅<https://github.com/aspnet/Docs/issues/4076> 和[其他资源](#)了解详细信息。目前，只能输入整数，例如 10。
- 在一些区域设置中，需要指定日期格式。请参阅下方突出显示的代码。

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

我们将在本教程的后续部分中探讨 `DataAnnotations`。

点击“创建”后，窗体会发布到服务器，其中电影信息会保存在数据库中。应用重定向到 /Movies URL，其中会显示

新创建的电影信息。

Index

Create New

Title	Release Date	Genre	Price	
Conan	3/3/2017	Comedy	\$1.99	Edit Details Delete

© 2017 - MvcMovie

再创建几个其他的电影条目。试用“编辑”、“详细信息”和“删除”链接，它们均可正常工作。

现在你已拥有用于显示、编辑、更新和删除数据的数据库和页面。在下一个教程中，我们将使用此数据库。

其他资源

- [标记帮助程序](#)
- [全球化和本地化](#)

[上一篇 - 添加视图](#) [下一篇 - 使用SQLITE](#)

在 ASP.NET Core MVC 项目中使用 SQLite

2018/5/8 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

`MvcMovieContext` 对象处理连接到数据库并将 `Movie` 对象映射到数据库记录的任务。在 `Startup.cs` 文件的 `ConfigureServices` 方法中向 [依赖关系注入](#) 容器注册数据库上下文:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

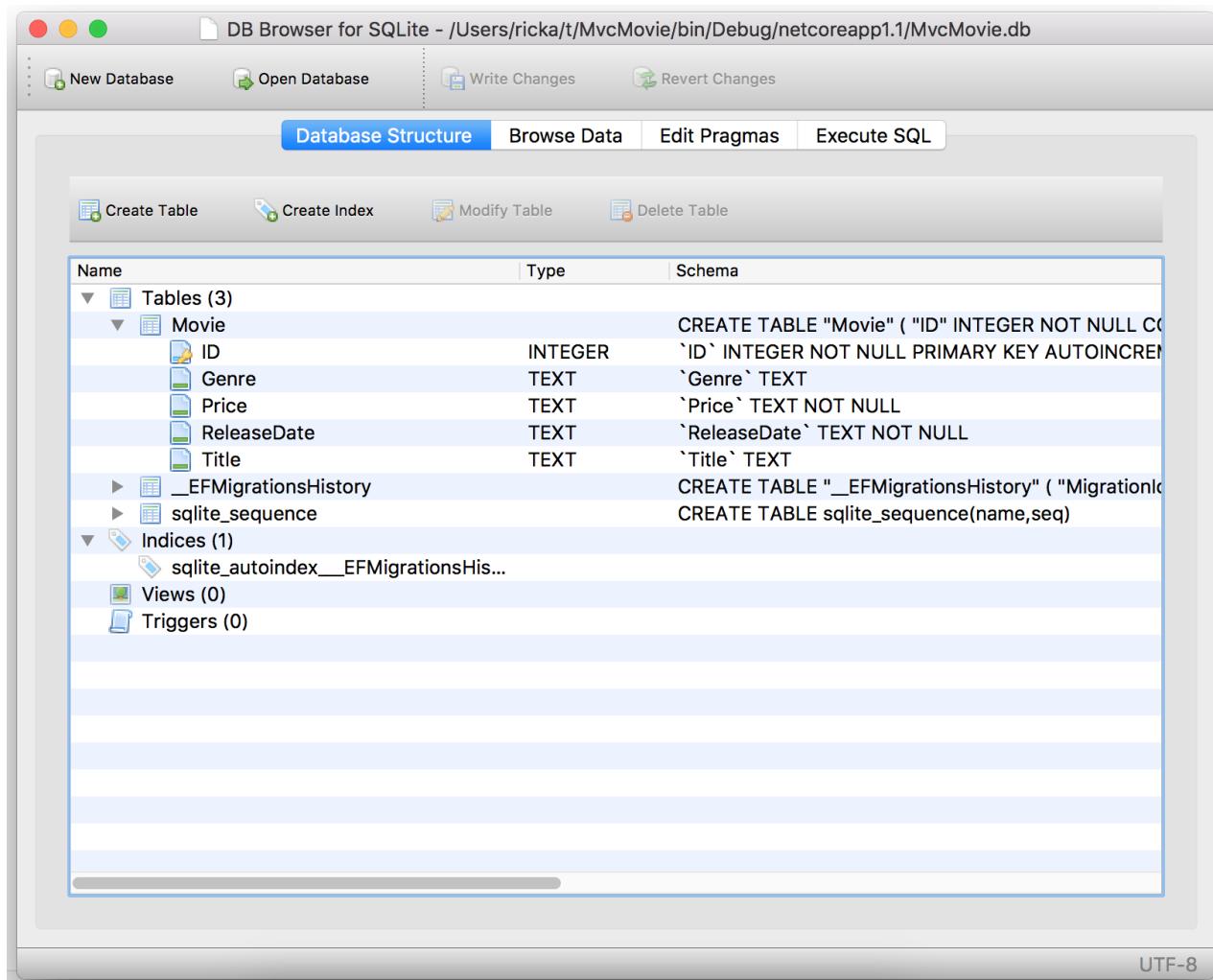
    services.AddDbContext<MvcMovieContext>(options =>
        options.UseSqlite("Data Source=MvcMovie.db"));
}
```

SQLite

[SQLite](#) 网站上表示:

SQLite 是一个自包含、高可靠性、嵌入式、功能完整、公共域的 SQL 数据库引擎。SQLite 是世界上使用最多的数据引擎。

可以下载许多第三方工具来管理并查看 SQLite 数据库。下面的图片来自 [DB Browser for SQLite](#)。如果你有最喜欢的 SQLite 工具, 请发表评论以分享你喜欢的方面。



设定数据库种子

在 Models 文件夹中创建一个名为 `SeedData` 的新类。将生成的代码替换为以下代码：

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return; // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-1-11"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },
                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },
                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}

```

如果 DB 中没有任何电影，则会返回种子初始值设定项。

```
if (context.Movie.Any())
{
    return; // DB has been seeded.
}
```

添加种子初始值设定项

将种子初始值设定项添加 Program.cs 文件中的 `Main` 方法：

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    // Requires using MvcMovie.Models;
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

测试应用

删除 DB 中的所有记录(使种子方法运行)。停止并启动应用以设定数据库种子。

应用将显示设定为种子的数据。

Ricka - Movie App

localhost:5000/Movies

MvcMovie

Create New

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

© 2017 - MvcMovie

[上一篇 - 添加模型](#)

[下一篇 - 控制器方法和视图](#)

ASP.NET Core 中的控制器方法和视图

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

我们的电影应用有个不错的开始, 但是展示效果还不够理想。我们不希望看到时间(如下图所示的 12:00:00 AM), 并且“ReleaseDate”应为两个词。

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	Edit Details Delete
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	Edit Details Delete
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	Edit Details Delete
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	Edit Details Delete

© 2017 - MvcMovie

打开 Models/Movie.cs 文件, 并添加以下代码中突出显示的行:

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

生成并运行应用。

我们将在下一教程中介绍 [DataAnnotations](#)。[Display](#) 特性指定要显示的字段名称的内容(本例中应为“Release

Date", 而不是"ReleaseDate")。 **DataType** 属性指定数据的类型(日期), 使字段中存储的时间信息不会显示。

浏览到 **Movies** 控制器, 并将鼠标指针悬停在"编辑"链接上以查看目标 URL。

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

© 2016 - MvcMovie

http://localhost:1234/Movies/Edit/5

"编辑"、"详细信息"和"删除"链接是在 Views/Movies/Index.cshtml 文件中由 Core MVC 定位标记帮助程序生成的。

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
<a asp-action="Details" asp-route-id="@item.ID">Details</a> |
<a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
</td>
</tr>
```

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。在上面的代码中, **AnchorTagHelper** 从控制器操作方法和路由 ID 动态生成 HTML **href** 特性值。在最喜欢的浏览器中使用"查看源", 或使用开发人员工具来检查生成的标记。生成的 HTML 的一部分如下所示:

```
<td>
<a href="/Movies/Edit/4"> Edit </a> |
<a href="/Movies/Details/4"> Details </a> |
<a href="/Movies/Delete/4"> Delete </a>
</td>
```

重新调用在 Startup.cs 文件中设置的**路由**的格式:

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

ASP.NET Core 将 `http://localhost:1234/Movies/Edit/4` 转换为对 **Movies** 控制器的 **Edit** 操作方法的请求, 参数 **Id** 为 4。(控制器方法也称为**操作方法**。)

标记帮助程序是 ASP.NET Core 中最受欢迎的新功能之一。有关详细信息，请参阅[其他资源](#)。

打开 `Movies` 控制器并检查两个 `Edit` 操作方法。以下代码显示了 `HTTP GET Edit` 方法，此方法将提取电影并填充由 `Edit.cshtml` Razor 文件生成的编辑表单。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

以下代码显示 `HTTP POST Edit` 方法，它会处理已发布的电影值：

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

`[Bind]` 特性是防止过度发布的一种方法。只应在 `[Bind]` 特性中包含想要更改的属性。有关详细信息，请参阅 [Protect your controller from over-posting](#)(防止控制器过度发布)。[ViewModels](#) 提供了一种替代方法以防止过度发布。

请注意第二个 `Edit` 操作方法的前面是 `[HttpPost]` 特性。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

`HttpPost` 特性指定只能为 `POST` 请求调用此 `Edit` 方法。可将 `[HttpGet]` 属性应用于第一个编辑方法，但不是必需，因为 `[HttpGet]` 是默认设置。

`ValidateAntiForgeryToken` 特性用于[防止请求伪造](#)，并与编辑视图文件 (`Views/Movies/Edit.cshtml`) 中生成的防伪标记相配对。编辑视图文件使用[表单标记帮助程序](#)生成防伪标记。

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

[表单标记帮助程序](#)会生成隐藏的防伪标记，此标记必须与电影控制器的 `Edit` 方法中 `[ValidateAntiForgeryToken]` 生成的防伪标记相匹配。有关详细信息，请参阅[反请求伪造](#)。

`HttpGet` `Edit` 方法采用电影 `ID` 参数，使用 Entity Framework `SingleOrDefaultAsync` 方法查找电影，并将所选电影返回到“编辑”视图。如果无法找到电影，则返回 `NotFound` (HTTP 404)。

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

当基架系统创建“编辑”视图时，它会检查 `Movie` 类并创建代码为类的每个属性呈现 `<label>` 和 `<input>` 元素。以下示例显示由 Visual Studio 基架系统生成的“编辑”视图：

```

@model MvcMovie.Models.Movie

 @{
     ViewData["Title"] = "Edit";
 }

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Genre" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Price" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

请注意视图模板在文件顶端有一个 `@model MvcMovie.Models.Movie` 语句。`@model MvcMovie.Models.Movie` 指定视图期望的视图模板的模型为 `Movie` 类型。

基架的代码使用几个标记帮助程序方法来简化 HTML 标记。[标签标记帮助程序](#) 显示字段的名称 ("Title"、"ReleaseDate"、"Genre" 或 "Price")。[输入标记帮助程序](#) 呈现 HTML `<input>` 元素。[验证标记帮助程序](#) 显示与该属性相关联的任何验证消息。

运行应用程序并导航到 `/Movies` URL。点击“编辑”链接。在浏览器中查看页面的源。为 `<form>` 元素生成的 HTML 如下所示。

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxp63fRFqUePGvuI5jGzsloJu1L7X9le1gy7NCI1SduCRx9jDQC1rv9p0TTmqUyXnJBXhmrjcUVDJyDUMm7-MF_9rK8aAZdRd10ri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOsTEg7RU" />
</form>
```

`<input>` 元素位于 `HTML <form>` 元素中，后者的 `action` 特性设置为发布到 `/Movies/Edit/id` URL。当单击 `Save` 按钮时，表单数据将发布到服务器。关闭 `</form>` 元素之前的最后一行显示 [表单标记帮助程序](#) 生成的隐藏的 `XSRF` 标记。

处理 POST 请求

以下列表显示了 `Edit` 操作方法的 `[HttpPost]` 版本。

```

// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

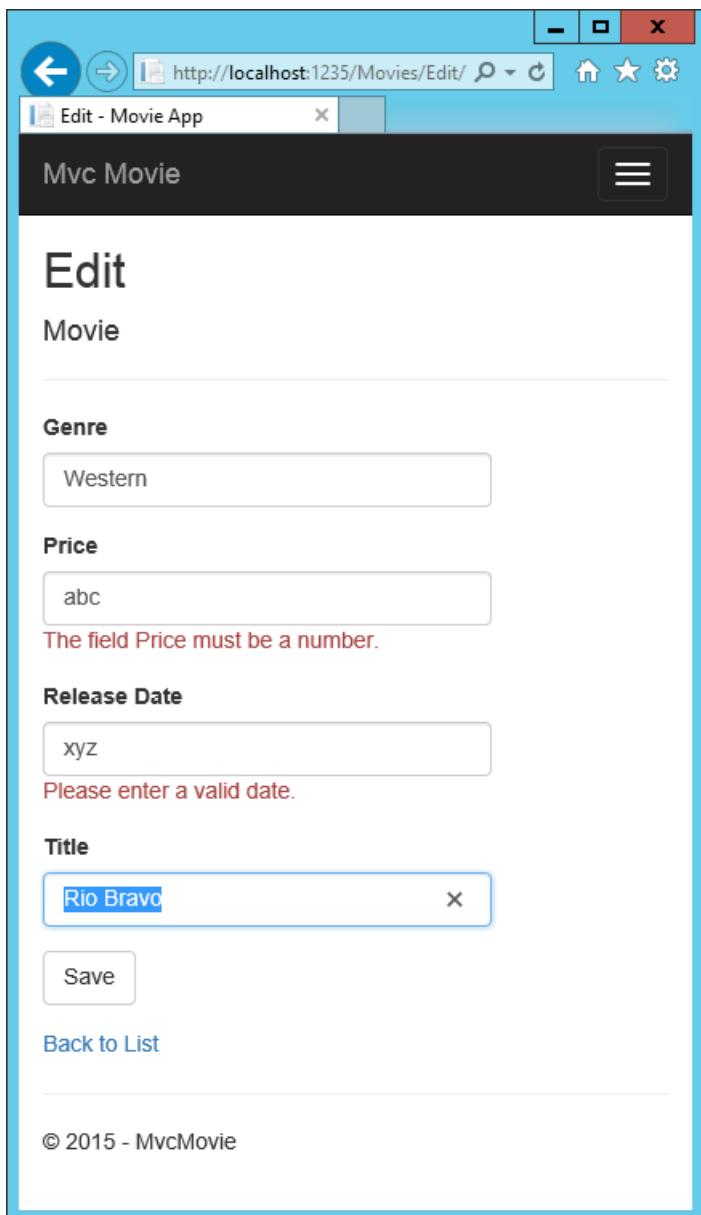
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

[ValidateAntiForgeryToken] 特性验证表单标记帮助程序中的防伪标记生成器生成的隐藏的 [XSRF](#) 标记

[模型绑定](#) 系统采用发布的表单值，并创建一个作为 `movie` 参数传递的 `Movie` 对象。`ModelState.IsValid` 方法验证表单中提交的数据是否可以用于修改(编辑或更新) `Movie` 对象。如果数据有效，将保存此数据。通过调用数据库上下文的 `SaveChangesAsync` 方法，将更新(编辑)的电影数据保存到数据库。保存数据后，代码将用户重定向到 `MoviesController` 类的 `Index` 操作方法，此方法显示电影集合，包括刚才所做的更改。

在表单发布到服务器之前，客户端验证会检查字段上的任何验证规则。如果有任何验证错误，则将显示错误消息，并且不会发布表单。如果禁用 JavaScript，则不会进行客户端验证，但服务器将检测无效的发布值，并且表单值将与错误消息一起重新显示。稍后在本教程中，我们将更详细地研究[模型验证](#)。`Views/Movies/Edit.cshtml` 视图模板中的[验证标记帮助程序](#)负责显示相应的错误消息。



电影控制器中的所有 `HttpGet` 方法都遵循类似的模式。它们获取电影对象(对于 `Index` 获取的是对象列表)并将对象(模型)传递给视图。`Create` 方法将空的电影对象传递给 `Create` 视图。在方法的 `[HttpPost]` 重载中, 创建、编辑、删除或其他方式修改数据的所有方法都执行此操作。以 `HTTP GET` 方式修改数据是一种安全隐患。以 `HTTP GET` 方法修改数据也违反了 HTTP 最佳做法和架构 REST 模式, 后者指定 GET 请求不应更改应用程序的状态。换句话说, 执行 GET 操作应是没有任何隐患的安全操作, 也不会修改持久数据。

其他资源

- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)
- [防请求伪造](#)
- [防止控制器过度发布](#)
- [ViewModels](#)
- [表单标记帮助程序](#)
- [输入标记帮助程序](#)
- [标签标记帮助程序](#)
- [选择标记帮助程序](#)
- [验证标记帮助程序](#)

[上一篇 - 使用
SQLITE](#)

[下一篇 - 添加搜
索](#)

将搜索添加到 ASP.NET Core MVC 应用

2018/5/8 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Index` 操作方法添加搜索功能, 以实现按“类型”或“名称”搜索电影。

使用以下代码更新 `Index` 方法:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

`Index` 操作方法的第一行创建了 `LINQ` 查询用于选择电影:

```
var movies = from m in _context.Movie
             select m;
```

此时仅对查询进行了定义, 它还不会针对数据库运行。

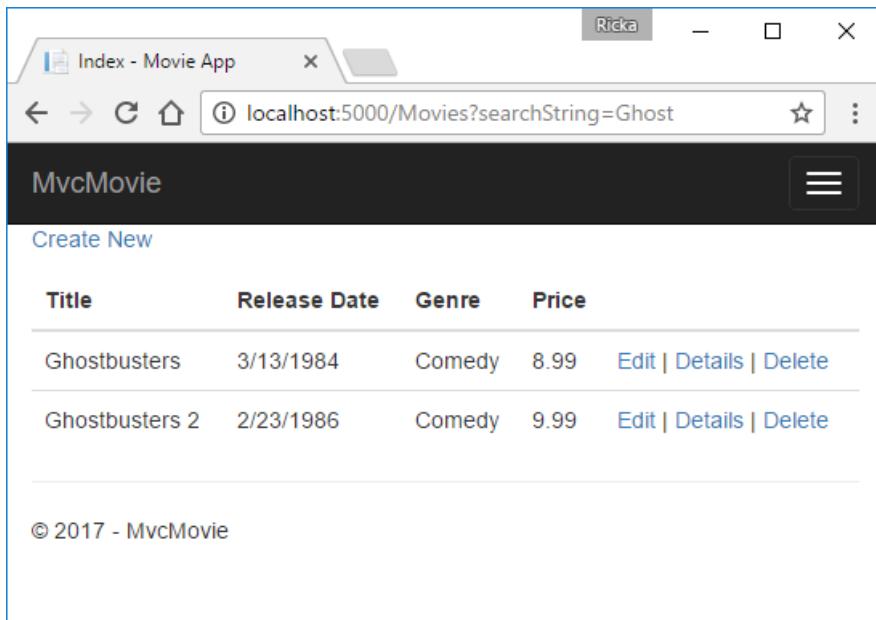
如果 `searchString` 参数包含一个字符串, 电影查询则会被修改为根据搜索字符串的值进行筛选:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

上面的 `s => s.Title.Contains()` 代码是 [Lambda 表达式](#)。Lambda 在基于方法的 `LINQ` 查询中用作标准查询运算符方法的参数, 如 `Where` 方法或 `Contains` (上述的代码中所使用的)。在对 `LINQ` 查询进行定义或通过调用方法 (如 `Where`、`Contains` 或 `OrderBy`) 进行修改后, 此查询不会被执行。相反, 会延迟执行查询。这意味着表达式的计算会延迟, 直到真正循环访问其实现的值或者调用 `ToListAsync` 方法为止。有关延迟执行查询的详细信息, 请参阅[Query Execution](#)(查询执行)。

注意:`Contains` 方法在数据库上运行, 而不是在上面显示的 c# 代码中运行。查询是否区分大小写取决于数据库和排序规则。在 SQL Server 上, `Contains` 映射到 `SQL LIKE`, 这是不区分大小写的。在 SQLite 中, 由于使用了默认排序规则, 因此需要区分大小写。

导航到 `/Movies/Index`。将查询字符串(如 `?searchString=Ghost`)追加到 URL。筛选的电影将显示出来。



如果将 `Index` 方法的签名更改为具有名称为 `id` 的参数，则 `id` 参数将匹配 `Startup.cs` 中设置的默认路由的可选 `{id}` 占位符。

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

请注意：SQLite 区分大小写，因此需搜索“Ghost”而非“ghost”。

之前的 `Index` 方法：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

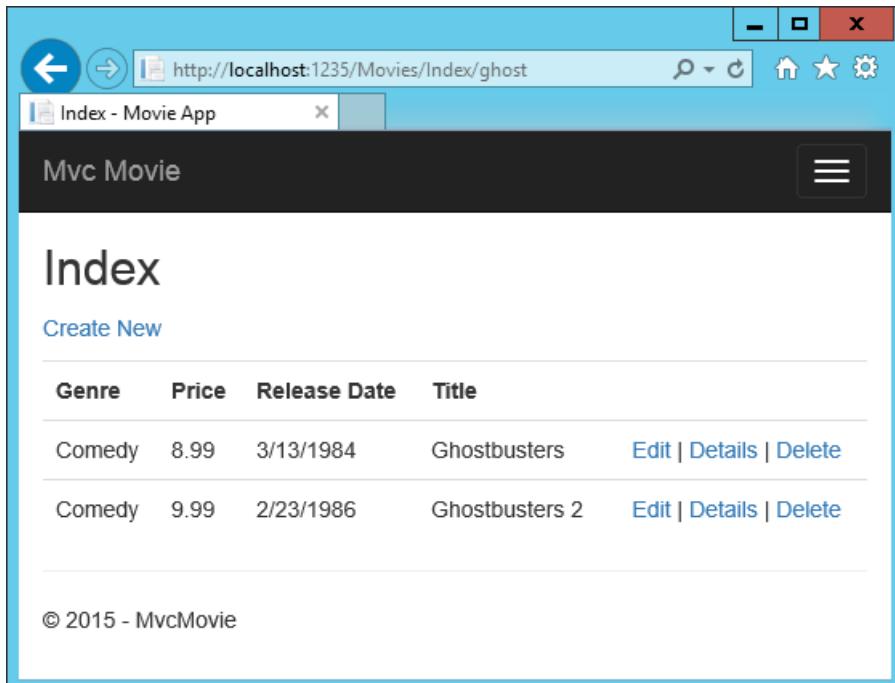
更新后带 `id` 参数的 `Index` 方法：

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

现可将搜索标题作为路由数据(URL 段)而非查询字符串值进行传递。



Genre	Price	Release Date	Title	
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete

但是，不能指望用户在每次要搜索电影时都修改 URL。因此需要添加 UI 元素来帮助他们筛选电影。若已更改 `Index` 方法的签名，以测试如何传递绑定路由的 `ID` 参数，请改回原样，使其采用名为 `searchString` 的参数：

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

打开“Views/Movies/Index.cshtml”文件，并添加以下突出显示的 `<form>` 标记：

```
ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

此 HTML `<form>` 标记使用表单标记帮助程序，因此提交表单时，筛选器字符串会发布到电影控制器的 `Index` 操作。

作。保存更改，然后测试筛选器。

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	Edit Details Delete
Comedy	8.99	3/13/1984	Ghostbusters	Edit Details Delete
Comedy	9.99	2/23/1986	Ghostbusters 2	Edit Details Delete
Western	3.99	4/15/1959	Rio Bravo	Edit Details Delete

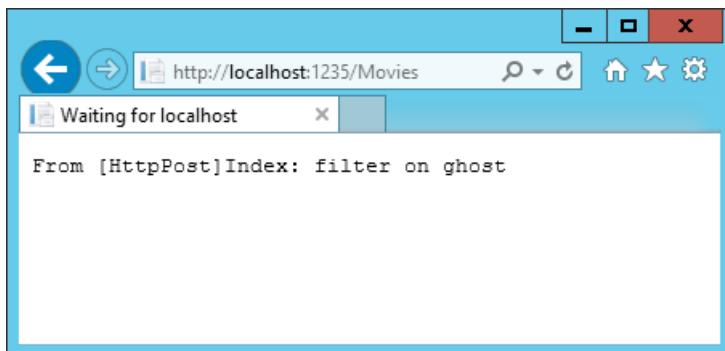
如你所料，不存在 `Index` 方法的 `[HttpPost]` 重载。无需重载，因为该方法不更改应用的状态，仅筛选数据。

可添加以下 `[HttpPost] Index` 方法。

```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

`notUsed` 参数用于创建 `Index` 方法的重载。本教程稍后将对此进行探讨。

如果添加此方法，则操作调用程序将与 `[HttpPost] Index` 方法匹配，且将运行 `[HttpPost] Index` 方法，如下图所示。



但是，即使添加 `Index` 方法的 `[HttpPost]` 版本，其实现方式也受到限制。假设你想要将特定搜索加入书签，或向朋友发送一个链接，让他们单击链接即可查看筛选出的相同电影列表。请注意，HTTP POST 请求的 URL 与 GET 请求的 URL 相同 (`localhost:xxxxx/Movies/Index`)，其中不包含搜索信息。搜索字符串信息作为表单域值发送给服务器。可使用浏览器开发人员工具或出色的 `Fiddler` 工具对其进行验证。下图展示了 Chrome 浏览器开发人员工具：

The screenshot shows the Chrome DevTools Network tab with a red box highlighting the 'Movies' entry in the list. Another red box highlights the 'General' section of the request details, showing the URL, method, status code, and headers. A third red box highlights the 'Form Data' section, showing the search string and a long RequestVerificationToken.

在请求正文 中，可看到搜索参数和 **XSRF** 标记。请注意，正如之前教程所述，[表单标记帮助程序](#) 会生成一个 XSRF 防伪标记。不会修改数据，因此无需验证控制器方法中的标记。

搜索参数位于请求正文而非 URL 中，因此无法捕获该搜索信息进行书签设定或与他人共享。将通过指定请求为 **HTTP GET** 进行修复。

在“Views\movie\Index.cshtml”Razor 视图中更改 `<form>` 标记以指定 `method="get"`：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

现在提交搜索后，URL 将包含搜索查询字符串。即使具备 `HttpPost Index` 方法，搜索也将转到 `HttpGet Index` 操

作方法。

以下标记显示对 `form` 标记的更改：

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

添加“按流派搜索”

将以下 `MovieGenreViewModel` 类添加到“模型”文件夹：

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

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> movies;
        public SelectList genres;
        public string movieGenre { get; set; }
    }
}
```

“电影流派”视图模型将包含：

- 电影列表。
- 包含流派列表的 `SelectList`。用户可通过它从列表中选择一种流派。
- 包含所选流派的 `movieGenre`。

将 `MoviesController.cs` 中的 `Index` 方法替换为以下代码：

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!String.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel();
    movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    movieGenreVM.movies = await movies.ToListAsync();

    return View(movieGenreVM);
}

```

以下代码是一种 `LINQ` 查询，可从数据库中检索所有流派。

```

// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                         orderby m.Genre
                                         select m.Genre;

```

通过投影不同的流派创建 `SelectList`（我们不希望选择列表中的流派重复）。

```
movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())
```

向索引视图添加“按流派搜索”

按如下更新 `Index.cshtml`：

```

@model MvcMovie.Models.MovieGenreViewModel

 @{
     ViewData["Title"] = "Index";
 }

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <select asp-for="movieGenre" asp-items="Model.genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

检查以下 HTML 帮助程序中使用的 Lambda 表达式：

```
@Html.DisplayNameFor(model => model.movies[0].Title)
```

在上述代码中，`DisplayNameFor` HTML 帮助程序检查 Lambda 表达式中引用的 `Title` 属性来确定显示名称。由于只检查但未计算 Lambda 表达式，因此当 `model`、`model.movies[0]` 或 `model.movies` 为 `null` 或空时，你不会收到访问冲突。对 Lambda 表达式求值时（例如，`@Html.DisplayFor(modelItem => item.Title)`），将求得该模型的属性值。

通过按流派或/和电影标题搜索来测试应用。

[上一篇 - 控制器方法和视图](#)

[下一篇 - 添加字段](#)

添加新字段

2018/5/8 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本教程将向 `Movies` 表添加一个新字段。当更改架构(添加一个新的字段)时, 我们将丢弃数据库并创建一个新的数据库。此工作流适用于在没有要保留的任何生产数据的早期开发阶段。

在部署了应用并且具有要保留的数据后, 在需要更改架构时则不能丢弃数据库。Entity Framework [Code First 迁移](#)使你能够更新架构并迁移数据库, 而不会导致数据丢失。迁移是使用 SQL Server 时的一个常用的功能, 但 SQLite 不支持许多迁移架构操作, 因此只可能进行非常简单的迁移。有关详细信息, 请参阅 [SQLite 限制](#)。

向电影模型添加分级属性

打开 Models/Movie.cs 文件, 并添加 `Rating` 属性:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

因为已经添加新字段到 `Movie` 类, 所以还需要更新绑定允许名单, 将此新属性纳入其中。在 MoviesController.cs 中, 更新 `Create` 和 `Edit` 操作方法的 `[Bind]` 属性, 以包括 `Rating` 属性:

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

还需要更新视图模板以在浏览器视图中显示、创建和编辑新的 `Rating` 属性。

编辑 /Views/Movies/Index.cshtml 文件并添加 `Rating` 字段:

```


| @Html.DisplayNameFor(model => model.movies[0].Title) | @Html.DisplayNameFor(model => model.movies[0].ReleaseDate) | @Html.DisplayNameFor(model => model.movies[0].Genre) | @Html.DisplayNameFor(model => model.movies[0].Price) | @Html.DisplayNameFor(model => model.movies[0].Rating) |
|------------------------------------------------------|------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------|-------------------------------------------------------|
| @Html.DisplayFor(modelItem => item.Title)            | @Html.DisplayFor(modelItem => item.ReleaseDate)            | @Html.DisplayFor(modelItem => item.Genre)            | @Html.DisplayFor(modelItem => item.Price)            | @Html.DisplayFor(modelItem => item.Rating)            |


```

使用 `Rating` 字段更新 /Views/Movies/Create.cshtml。

在将 DB 更新为包括新字段之前，应用不会正常工作。如果现在运行，会出现以下 `SqliteException`：

```
SqliteException: SQLite Error 1: 'no such column: m.Rating'.
```

看到此错误是因为更新的 Movie 模型类与现有数据库的 Movie 表架构不同。（数据库表中没有 `Rating` 列。）

可通过几种方法解决此错误：

1. **丢弃数据库**，让 Entity Framework 基于新的模型类架构自动重新创建数据库。如果使用此方法，则会丢失数据库中的现有数据，因此不能对生产数据库使用此方法！使用初始值设定项，以使用测试数据自动设定数据库种子，这通常是开发应用的有效方式。
2. 手动修改现有数据库的架构，使它与模型类匹配。此方法的优点是可以保留数据。可以手动或通过创建数据库更改脚本进行此更改。
3. 使用 **Code First** 迁移更新数据库架构。

在本教程中，我们将在架构更改时丢弃并重新创建数据库。从终端运行以下命令丢弃 db：

```
dotnet ef database drop
```

更新 `SeedData` 类，使它提供新列的值。示例更改如下所示，但可能需要对每个 `new Movie` 做出此更改。

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

将 `Rating` 字段添加到 `Edit`、`Details` 和 `Delete` 视图。

运行应用，并验证是否可以创建/编辑/显示具有 `Rating` 字段的电影。模板。

上一篇 - 添加搜索

下一篇 - 添加验证

添加验证

2018/5/8 • 11 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

在本部分中, 将向 `Movie` 模型添加验证逻辑, 并确保每当用户创建或编辑电影时, 都会强制执行验证规则。

坚持 DRY 原则

MVC 的设计原则之一是 `DRY` (“不要自我重复”)。ASP.NET MVC 支持你仅指定一次功能或行为, 然后使它应用到整个应用中。这可以减少所需编写的代码量, 并使编写的代码更少出错, 更易于测试和维护。

MVC 和 Entity Framework Core Code First 提供的验证支持是 DRY 原则在实际操作中的极佳示例。可以在一个位置(模型类中)以声明方式指定验证规则, 并且在应用中的所有位置强制执行。

将验证规则添加到电影模型

打开 `Movie.cs` 文件。DataAnnotations 提供一组内置验证特性, 可通过声明方式应用于任何类或属性。(它还包含 `DataType` 等格式特性, 这些特性可帮助进行格式设置, 但不提供任何验证。)

更新 `Movie` 类以使用内置的 `Required`、`StringLength`、`RegularExpression` 和 `Range` 验证特性。

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z''"\s-]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'''\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

验证特性指定要对应用这些特性的模型属性强制执行的行为。`Required` 和 `MinimumLength` 特性表示属性必须有值; 但用户可输入空格来满足此验证。`RegularExpression` 特性用于限制可输入的字符。在上述代码中, `Genre` 和 `Rating` 仅可使用字母(禁用空格、数字和特殊字符)。`Range` 特性将值限制在指定范围内。`StringLength` 特性使你能够设置字符串属性的最大长度, 以及可选的最小长度。从本质上来说, 需要值类型(如 `decimal`、`int`、`float`、`DateTime`), 但不需要 `[Required]` 特性。

让 ASP.NET 强制自动执行验证规则有助于提升应用的可靠性。同时它能确保你无法忘记验证某些内容, 并防止你

无意中将错误数据导入数据库。

MVC 中的验证错误 UI

运行应用并导航到电影控制器。

点击“新建”连接添加新电影的链接。使用无效值填写表单。当 jQuery 客户端验证检测到错误时，会显示一条错误消息。

Create

Movie

Title

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

Release Date

mm/dd/yyyy

Genre

The Genre field is required.

Price

z

The field Price must be a number.

Rating

The field Rating must match the regular expression '^([A-Z]+[a-zA-Z-\\s]*\$)'.

Create

Back to List

© 2017 - MvcMovie

注意

可能无法在 `Price` 字段中输入十进制逗号。若要使 jQuery 验证支持使用逗号（“,”）表示小数点的非英语区域设置，以及支持非美国英语日期格式，必须执行使应用全球化的步骤。有关添加十进制逗号的说明，请参阅 [GitHub 问题 4076](#)。

请注意表单如何自动呈现每个包含无效值的字段中相应的验证错误消息。客户端（使用 JavaScript 和 jQuery）和服务器端（若用户禁用 JavaScript）都必定会遇到这些错误。

明显的好处在于不需要在 `MoviesController` 类或 `Create.cshtml` 视图中更改单个代码行来启用此验证 UI。在本教程前面创建的控制器和视图会自动选取验证规则，这些规则是通过在 `Movie` 模型类的属性上使用验证特性所指定的。使用 `Edit` 操作方法测试验证后，即已应用相同的验证。

存在客户端验证错误时，不会将表单数据发送到服务器。可通过使用 [Fiddler 工具](#)或[F12 开发人员工具](#)在 `HTTP Post` 方法中设置断点来对此进行验证。

验证工作原理

你可能想知道在不对控制器或视图中的代码进行任何更新的情况下，验证 UI 是如何生成的。下列代码显示两种 `Create` 方法。

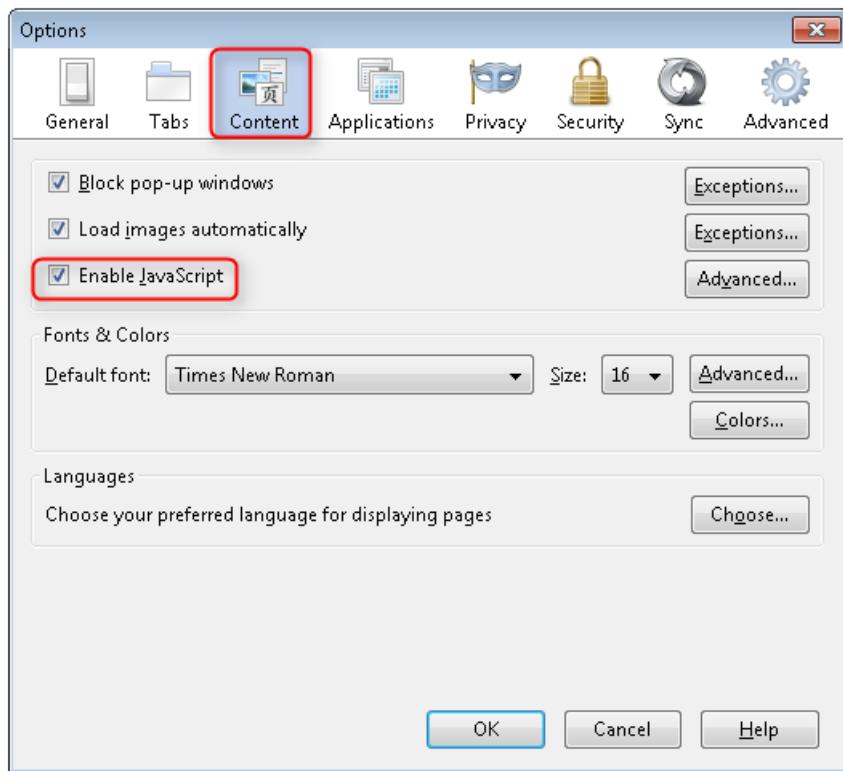
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

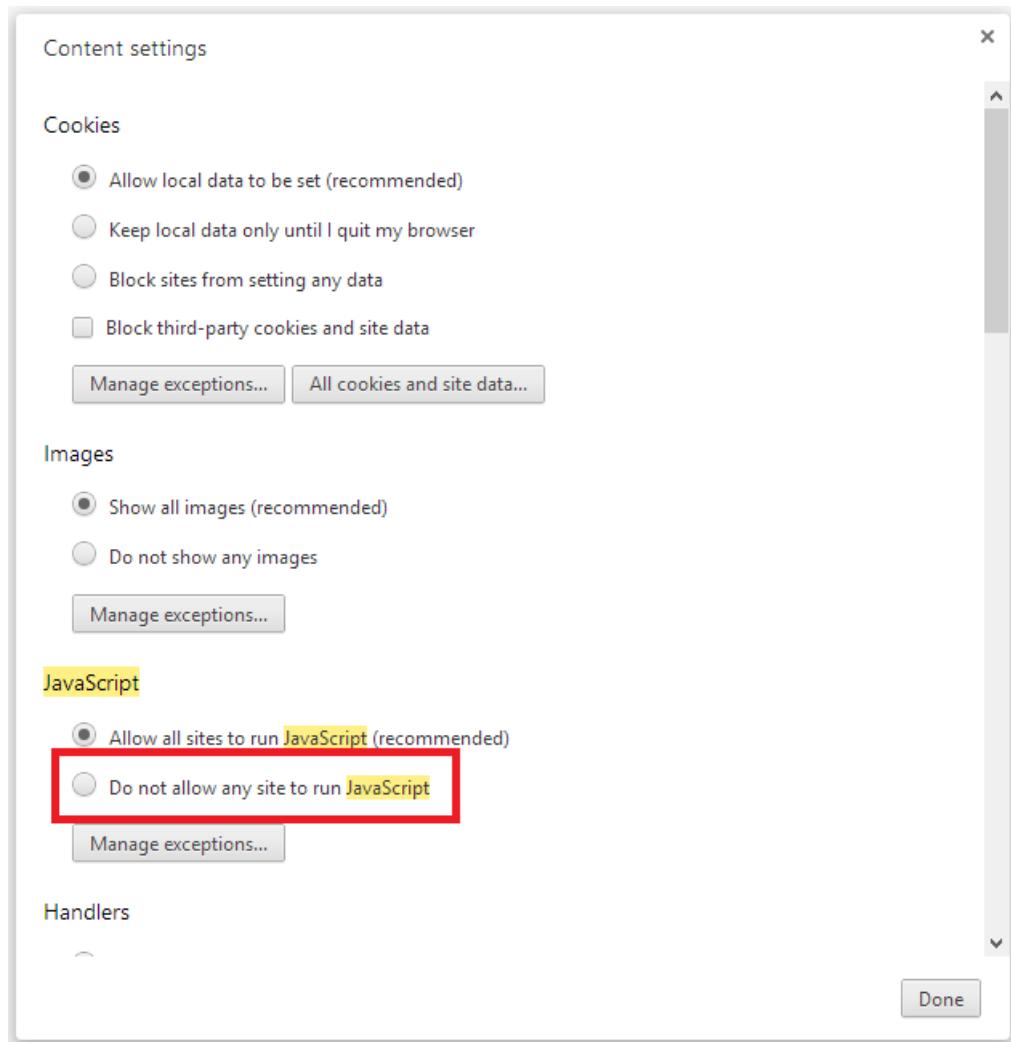
第一个 (HTTP GET) `Create` 操作方法显示初始的“创建”表单。第二个 (`[HttpPost]`) 版本处理表单发布。第二个 `Create` 方法 (`[HttpPost]` 版本) 调用 `ModelState.IsValid` 以检查电影是否有任何验证错误。调用此方法将评估已应用于对象的任何验证特性。如果对象有验证错误，则 `Create` 方法会重新显示此表单。如果没有错误，此方法则将新电影保存在数据库中。在我们的电影示例中，在检测到客户端上存在验证错误时，表单不会发布到服务器。当存在客户端验证错误时，第二个 `Create` 方法永远不会被调用。如果在浏览器中禁用 JavaScript，客户端验证将被禁用，而你可以测试 HTTP POST `Create` 方法 `ModelState.IsValid` 检测任何验证错误。

可以在 `[HttpPost] Create` 方法中设置断点，并验证方法从未被调用，客户端验证在检测到存在验证错误时不会提交表单数据。如果在浏览器中禁用 JavaScript，然后提交错误的表单，将触发断点。在没有 JavaScript 的情况下仍然可以进行完整的验证。

以下图片显示如何在 FireFox 浏览器中禁用 JavaScript。



以下图片显示如何在 Chrome 浏览器中禁用 JavaScript。



禁用 JavaScript 后，发布无效数据并单步执行调试程序。

```
74     // POST: Movies/Create
75     // To protect from overposting attacks, please enable this
76     // more details see http://go.microsoft.com/fwlink/?LinkID=142072
77     [HttpPost]
78     [ValidateAntiForgeryToken]
79     0 references
80     public async Task<IActionResult> Create([Bind("ID,Title")]
81     {
82         if (ModelState.IsValid)
83         {
84             _context.Add(movie);
85             await _context.SaveChangesAsync();
86             return RedirectToAction("Index");
87         }
88         return View(movie);
89     }

```

以下是之前在本教程中已搭建基架的 Create.cshtml 视图模板的一部分。以上所示的操作方法使用它来显示初始表单，并在发生错误时重新显示此表单。

```
<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>

        @*Markup removed for brevity.*@
    </div>
</form>
```

输入标记帮助程序使用 [DataAnnotations](#) 特性，并在客户端上生成 jQuery 验证所需的 HTML 特性。[验证标记帮助程序](#) 用于显示验证错误。有关详细信息，请参阅[验证](#)。

此方法真正好的一点是：无论是控制器还是 `Create` 视图模板都不知道强制实施的实际验证规则或显示的特定错误消息。仅可在 `Movie` 类中指定验证规则和错误字符串。这些相同的验证规则自动应用于 `Edit` 视图和可能创建用于编辑模型的任何其他视图模板。

需要更改验证逻辑时，可以通过将验证特性添加到模型在同一个位置实现此操作。（在此示例中为 `Movie` 类）。无需担心对应用程序的不同部分所强制执行规则的方式不一致 - 所有验证逻辑都将定义在一个位置并用于整个应用程序。这使代码非常简洁，并且更易于维护和改进。这意味着对 DRY 原则的完全遵守。

使用 `DataType` 特性

打开 `Movie.cs` 文件并检查 `Movie` 类。除了一组内置的验证特性，`System.ComponentModel.DataAnnotations` 命名空间还提供格式特性。我们已经在发布日期和价格字段中应用了 `DataType` 枚举值。以下代码显示具有适当 `DataType` 特性的 `ReleaseDate` 和 `Price` 属性。

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

`DataType` 属性仅提供相关提示来帮助视图引擎设置数据格式（并提供元素/属性，例如向 URL 提供 `<a>` 和向电子邮件提供 ``）。可以使用 `RegularExpression` 特性验证数据的格式。`DataType` 属性用于指定比数据库内部类型更具体的数据类型，它们不是验证属性。在此示例中，我们只想跟踪日期，而不是时间。`DataType` 枚举提供了多种数据类型，例如日期、时间、电话号码、货币、电子邮件地址等。应用程序还可通过 `DataType` 特性自动提供类型特定的功能。例如，可以为 `DataType.EmailAddress` 创建 `mailto:` 链接，并且可以在支持 HTML5 的浏览器中为 `DataType.Date` 提供日期选择器。`DataType` 特性发出 HTML 5 `data-`（读作 data dash）特性供 HTML 5 浏览器理解。`DataType` 特性不提供任何验证。

`DataType.Date` 不指定显示日期的格式。默认情况下，数据字段根据基于服务器的 `CultureInfo` 的默认格式进行显示。

`DisplayFormat` 特性用于显式指定日期格式：

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

`ApplyFormatInEditMode` 设置指定在文本框中显示值以进行编辑时也应用格式。（你可能不想为某些字段执行此操作—例如对于货币值，你可能不希望文本框中的货币符号可编辑。）

可以单独使用 `DisplayFormat` 特性，但通常建议使用 `DataType` 特性。`DataType` 特性传达数据的语义而不是传达如何在屏幕上呈现数据，并提供 `DisplayFormat` 不具备的以下优势：

- 浏览器可启用 HTML5 功能（例如显示日历控件、区域设置适用的货币符号、电子邮件链接等）
- 默认情况下，浏览器将根据区域设置采用正确的格式呈现数据。
- `DataType` 特性使 MVC 能够选择正确的字段模板来呈现数据（如果 `DisplayFormat` 由自身使用，则使用的是字符串模板）。

注意

jQuery 验证不适用于 `Range` 属性和 `DateTime`。例如，以下代码将始终显示客户端验证错误，即便日期在指定的范围内：

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

需要禁用 jQuery 日期验证才能使用具有 `DateTime` 的 `Range` 特性。通常，在模型中编译固定日期是不恰当的，因此不推荐使用 `Range` 特性和 `DateTime`。

以下代码显示组合在一行上的特性：

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^([A-Z]+[a-zA-Z''"\s-]*)$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    public decimal Price { get; set; }

    [RegularExpression(@"^([A-Z]+[a-zA-Z'''\s-]*)$"), StringLength(5)]
    public string Rating { get; set; }
}
```

在本系列的下一部分中，我们将回顾应用程序，并对自动生成的 `Details` 和 `Delete` 方法进行一些改进。

其他资源

- [使用表单](#)
- [全球化和本地化](#)
- [标记帮助程序简介](#)
- [创作标记帮助程序](#)

[上一篇 - 添加字段](#)

[下一篇 - 检查详细信息和删除方法](#)

检查 ASP.NET Core 应用的 Details 和 Delete 方法

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

打开电影控制器，并检查 `Details` 方法：

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

创建此操作方法的 MVC 基架引擎添加显示调用方法的 HTTP 请求的注释。在此情况下，它是包含三个 URL 段的 GET 请求，这三个段为 `Movies` 控制器、`Details` 方法和 `id` 值。回顾这些在 `Startup.cs` 中定义的段。

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

EF 可以使用 `SingleOrDefaultAsync` 方法轻松搜索数据。该方法中内置的一个重要安全功能是，代码会先验证搜索方法已经找到电影，然后再执行操作。例如，一个黑客可能通过将链接创建的 URL 从 `http://localhost:xxxx/Movies/Details/1` 更改为类似 `http://localhost:xxxx/Movies/Details/12345` 的值（或者不代表任何实际电影的其他值）将错误引入站点。如果未检查是否有空电影，则应用可能引发异常。

检查 `Delete` 和 `DeleteConfirmed` 方法。

```

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

    var movie = await _context.Movie
        .SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}

```

请注意，`HTTP GET Delete` 方法不删除指定的电影，而是返回可在其中提交 (HttpPost) 删除的电影视图。执行删除操作以响应 GET 请求(或者说，执行编辑操作、创建操作或更改数据的任何其他操作)会打开安全漏洞。

删除数据的 `[HttpPost]` 方法命名为 `DeleteConfirmed`，以便为 HTTP POST 方法提供一个唯一的签名或名称。下面显示了两个方法签名：

```

// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

```

```

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{

```

公共语言运行时 (CLR) 需要重载方法拥有唯一的参数签名(相同的方法名称但不同的参数列表)。但是，这里需要两个 `Delete` 方法 -- 一个用于 GET，另一个用于 POST -- 这两个方法拥有相同的参数签名。(它们都需要接受单个整数作为参数。)

可通过两种方法解决此问题，一种是为方法提供不同的名称。这正是前面的示例中的基架机制进行的操作。但是，这会造成一个小问题：ASP.NET 按名称将 URL 段映射到操作方法，如果重命名方法，则路由通常无法找到该方法。该示例中也提供了解决方案，即向 `DeleteConfirmed` 方法添加 `ActionName("Delete")` 属性。该属性对路由系统执行映射，以便包括 POST 请求的 /Delete/ 的 URL 可找到 `DeleteConfirmed` 方法。

对于名称和签名相同的方法，另一个常用解决方法是手动更改 POST 方法的签名以包括额外(未使用)的参数。这正是前面的文章中添加 `notUsed` 参数时进行的操作。这里为了 `[HttpPost] Delete` 方法可以执行同样的操作：

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

发布到 Azure

有关如何使用 Visual Studio 将该应用发布到 Azure 的说明, 请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。此外, 还可以从[命令行](#)发布应用。

感谢读完这篇 ASP.NET Core MVC 简介。我们期待你的意见。[MVC 和 EF Core 入门](#)是本教程的优选后续教程。

[上一篇](#)

使用 ASP.NET Core 和 Visual Studio for Mac 创建 Web API

2018/5/14 • 17 min to read • [Edit Online](#)

作者: Rick Anderson 和 Mike Wasson

注意

ASP.NET Core 2.1 is in preview and not recommended for production use.

本教程将生成一个用于管理“待办事项”列表的 Web API。不构造 UI。

本教程提供 3 个版本：

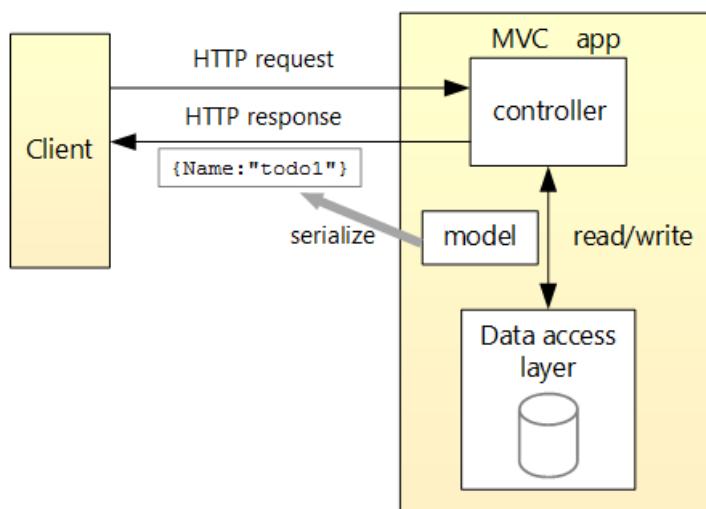
- macOS: 使用 Visual Studio for Mac 创建 Web API (本教程)
- Windows: [使用 Visual Studio for Windows 创建 Web API](#)
- macOS、Linux、Windows: [使用 Visual Studio Code 创建 Web API](#)

概述

本教程将创建以下 API：

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中，唯一的模型是待办事项。模型表示为 C# 类，也称为 Plain Old C# Object (POCO)。
- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

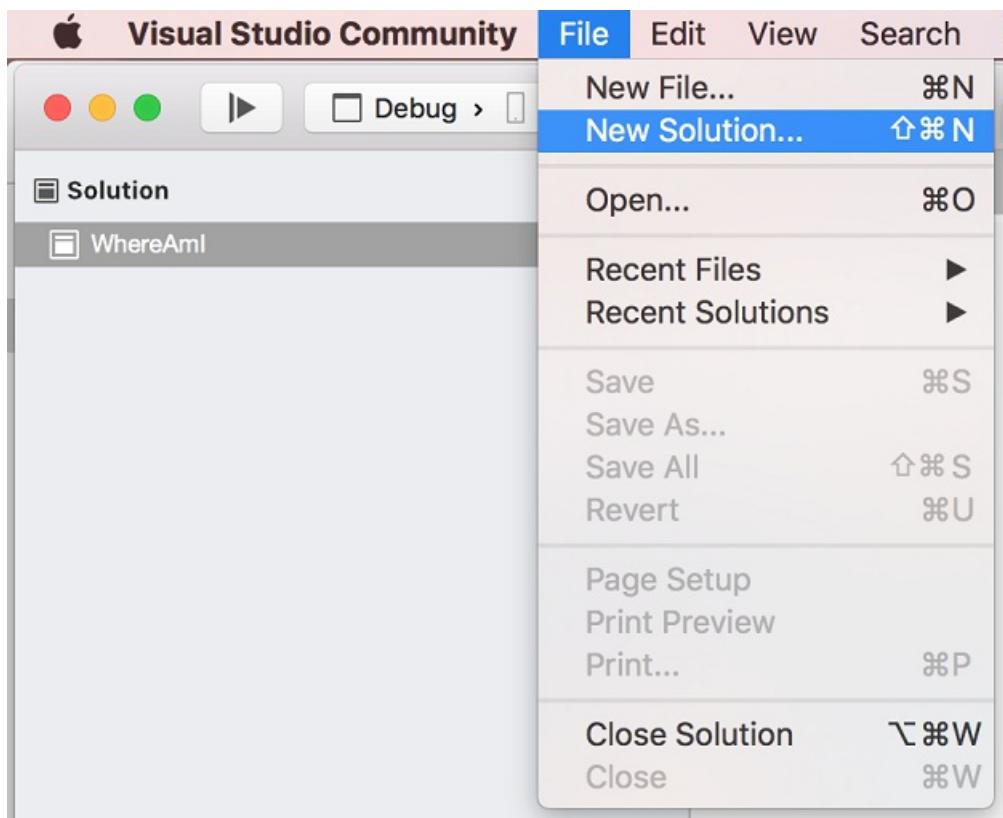
请参阅 [macOS 或 Linux 上的 ASP.NET Core MVC 介绍](#)，获取使用永久数据库的示例。

系统必备

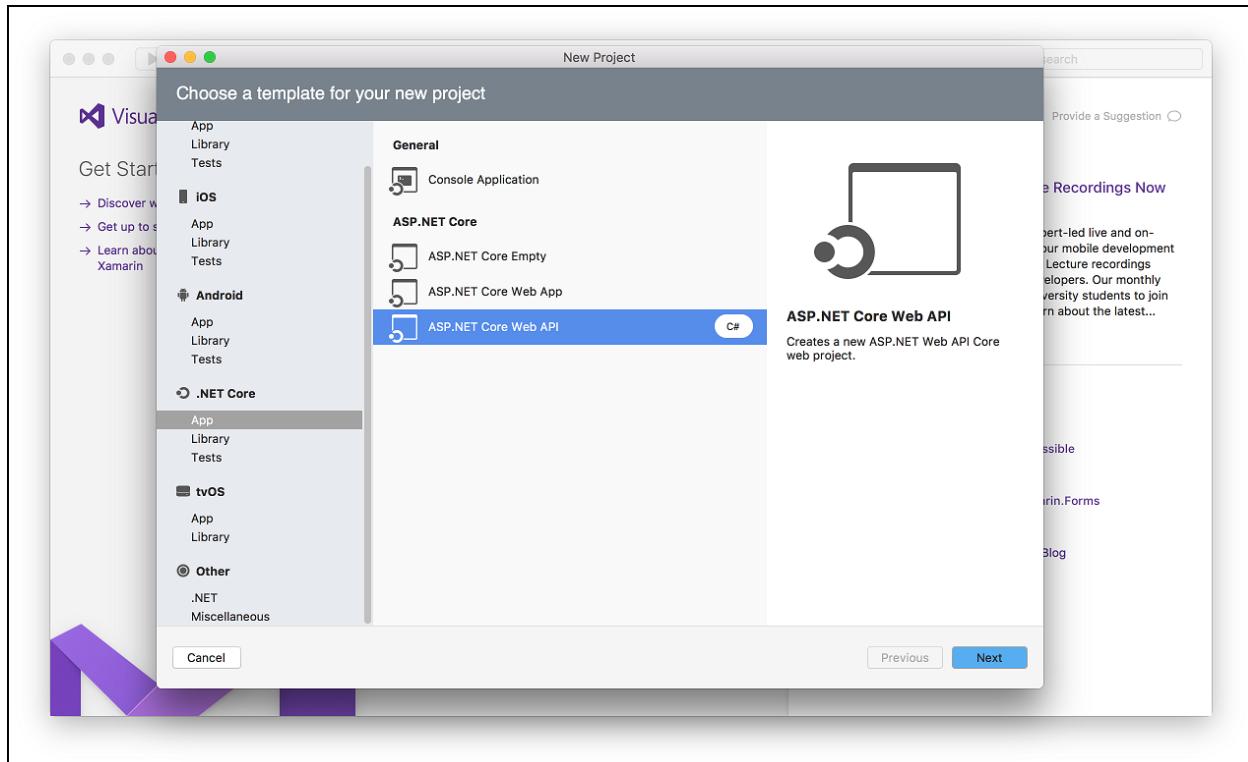
[Visual Studio for Mac](#)

创建项目

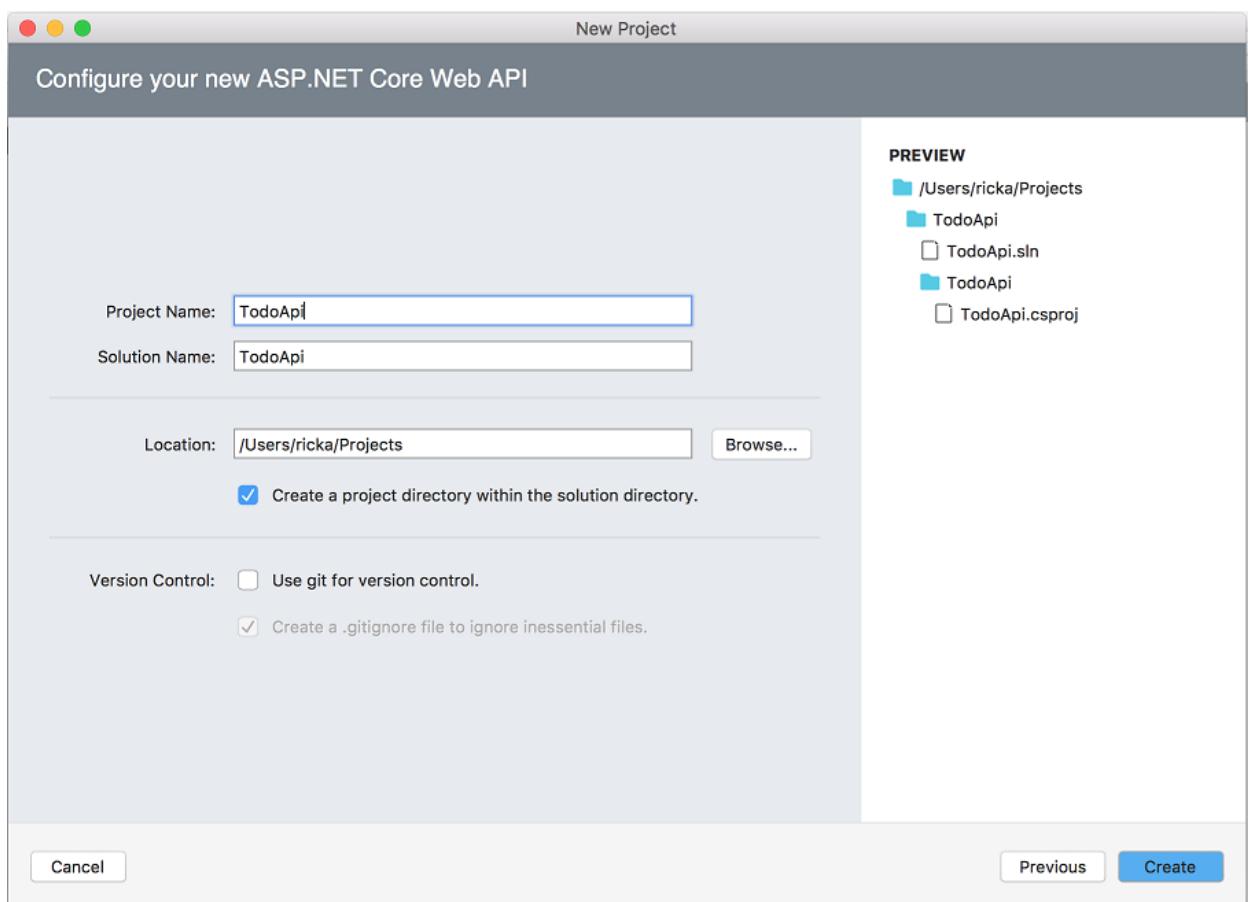
在 Visual Studio 中，选择“文件”>“新建解决方案”。



选择“.NET Core App”>“ASP.NET Core Web API”>“下一步”。



输入“TodoApi”作为“项目名称”，然后选择“创建”。



启动应用

在 Visual Studio 中，选择“运行” > “开始执行(调试)”启动应用。Visual Studio 启动浏览器并导航到

`http://localhost:5000`。将收到 HTTP 404(未找到)错误。将 URL 更改为 `http://localhost:<port>/api/values`。

`ValuesController` 数据已显示：

```
["value1","value2"]
```

添加对 Entity Framework Core 的支持

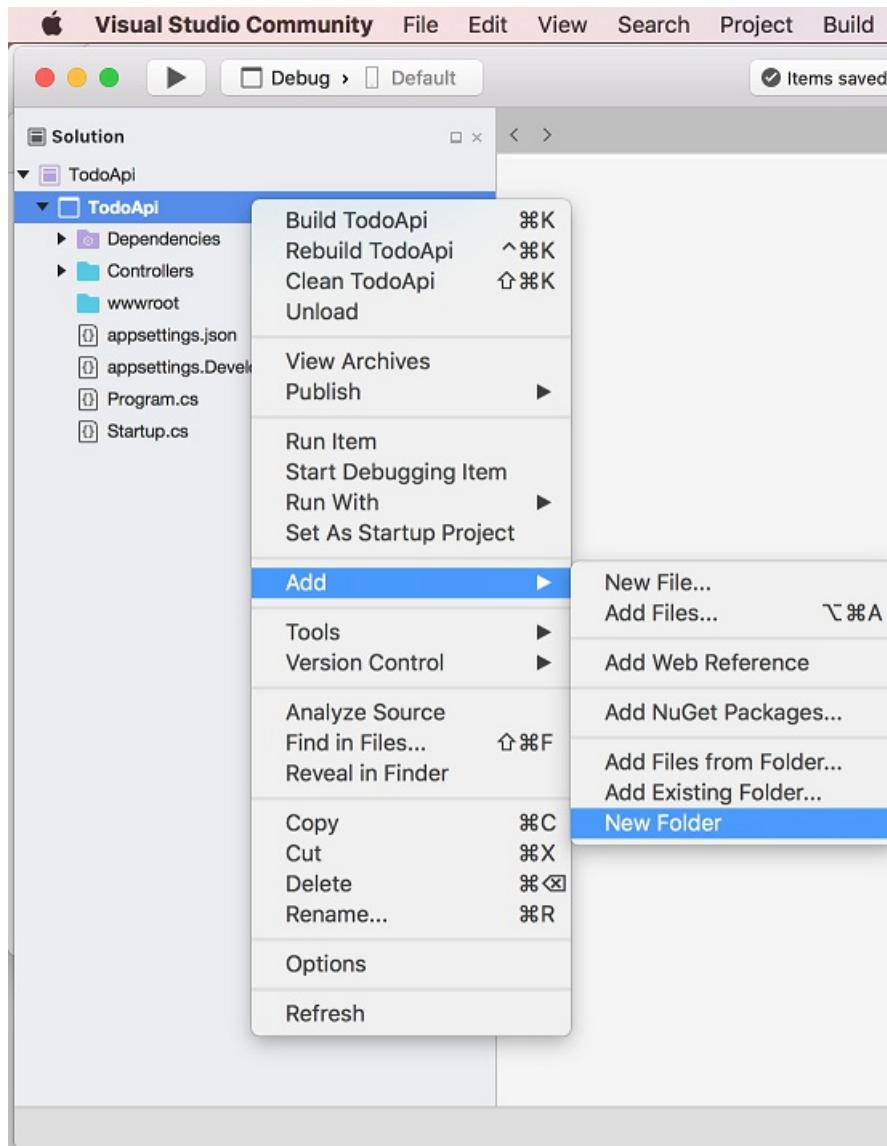
安装 [Entity Framework Core InMemory](#) 数据库提供程序。此数据库提供程序允许将 Entity Framework Core 和内存数据库一起使用。

- 从“项目”菜单中，选择“添加 NuGet 包”。
 - 或者，可以右键单击“依赖项”，然后选择“添加包”。
- 在搜索框中输入 `EntityFrameworkCore.InMemory`。
- 选择 `Microsoft.EntityFrameworkCore.InMemory`，然后选择“添加包”。

添加模型类

模型是表示应用中的数据的对象。在此示例中，唯一的模型是待办事项。

在解决方案资源管理器中，右键单击项目。选择“添加”>“新建文件夹”。将文件夹命名为“Models”。



注意

可将模型类置于项目的任意位置，但按照惯例会使用“模型”文件夹。

右键单击“Models”文件夹，然后选择“添加”>“新建文件”>“常规”>“空类”。将类命名为“TodoItem”，然后单击“新建”。

将生成的代码替换为：

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时，数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调 Entity Framework 功能的主类。可以通过继承 `Microsoft.EntityFrameworkCore.DbContext` 类的方式创建此类。

将 `TodoContext` 类添加到“Models”文件夹。

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 `Startup.cs` 文件的内容替换为以下代码：

[!code-csharp]

[!code-csharp]

前面的代码：

- 删 除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在解决方案资源管理器的“控制器”文件夹中，添加 `TodoController` 类。

将生成的代码替换为以下代码：

[!code-csharp]

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

[!code-csharp]

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以后用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项(如果不存在)添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

```
[!code-csharp]
```

```
[!code-csharp]
```

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
 {  
   "id": 1,  
   "name": "Item1",  
   "isComplete": false  
 }  
]
```

稍后将在本教程中演示如何使用 [Postman](#) 或 [curl](#) 查看 HTTP 响应。

路由和 URL 路径

`[HttpGet]` 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板(如 `[HttpGet("/products")]`)，则将它追加到路径。此示例不使用模板。
有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

```
[!code-csharp]
```

```
[!code-csharp]
```

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。
- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 JSON，并将 JSON 写入响应消息的正文中。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 `IActionResult` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 `ActionResult<T>` 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 Visual Studio 中，选择“运行”>“开始执行(调试)”启动应用。Visual Studio 启动浏览器并导航到 `http://localhost:<port>`，其中 `<port>` 是随机选择的端口号。将收到 HTTP 404(未找到)错误。将 URL 更改为 `http://localhost:<port>/api/values`。`ValuesController` 数据已显示：

```
["value1","value2"]
```

导航到位子 `http://localhost:<port>/api/todo` 的 `Todo` 控制器：

```
[{"key":1,"name":"Item1","isComplete":false}]
```

实现其他的 CRUD 操作

我们将向控制器添加 `Create`、`Update` 和 `Delete` 方法。这些方法是主题的变体，因此在这里只提供代码并突出显示主要的差异。添加或更改代码后生成项目。

创建

```
[!code-csharp]
```

上述方法响应至 HTTP POST，由 `[HttpPost]` 属性指示。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

```
[!code-csharp]
```

上述方法响应至 HTTP POST，由 `[HttpPost]` 属性指示。MVC 从 HTTP 请求正文获取待办事项的值。

`CreatedAtRoute` 方法返回 201 响应。201 响应是在服务器上创建新资源的 HTTP POST 方法的标准响应。

`CreatedAtRoute` 还会向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 [10.2.2 201 已创建](#)。

使用 Postman 发送创建请求

- 启动应用(“运行”>“开始执行(调试)”)。
- 打开 Postman。

The screenshot shows the Postman application interface. At the top, there's a navigation bar with 'File', 'Edit', 'View', 'Help', and workspace-related items like 'New', 'Import', 'Runner', 'My Workspace' (which is currently selected), and sync status. Below the navigation is a search bar with the URL 'https://localhost:44375'. A red box highlights the 'POST' dropdown menu. To its right is the target URL 'https://localhost:44375/api/todo'. Further right are 'Params' and a 'Send' button. Below the URL input, there are tabs for 'Authorization', 'Headers (3)', 'Body' (which is selected and highlighted with a red box), 'Pre-request Script', and 'Tests'. Under 'Body', there are four radio buttons: 'form-data', 'x-www-form-urlencoded', 'raw' (which is selected and highlighted with a red box), and 'binary'. A dropdown menu next to 'raw' shows 'JSON (application/json)' which is also highlighted with a red box. The main content area shows a JSON object with four lines of code: 1. {, 2. "name": "walk dog",, 3. "isComplete": true, 4. }. A red box highlights this entire JSON block. At the bottom of the interface, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Body' tab is active. On the right side, it shows the status 'Status: 201 Created' and time 'Time: 148 ms'. Below this, there are buttons for 'Pretty', 'Raw', 'Preview', and 'JSON' (which is selected and highlighted with a red box). The JSON preview area shows the same JSON object with an additional line 1. { at the top. At the bottom of this area are icons for copy, search, and refresh.

- 更新 localhost URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 输入包含待办事项的请求正文，类似以下 JSON：

```
{  
  "name": "walk dog",  
  "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应，则禁用“SSL 证书验证”选项。在“文件”>“设置”下可以找到该选项。在禁用该设置后，再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡，然后复制位置标头值：

POST https://localhost:44375/api/todo

Headers (3)

```
Content-Type: application/json; charset=utf-8
Date: Fri, 27 Apr 2018 18:32:32 GMT
Location: https://localhost/api/Todo/6
```

Status: 201 Created Time: 148 ms

可以使用此位置标头 URI 访问创建的资源。`Create` 方法返回 `CreatedAtRoute`。传递至 `CreatedAtRoute` 的第一个参数代表用于生成 URL 的命名路由。请回想一下， `GetById` 方法创建的 `"GetTodo"` 命令路由：

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

更新

```
[!code-csharp]
```

```
[!code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 204(无内容)。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

```
{  
    "key": 1,  
    "name": "walk dog",  
    "isComplete": true  
}
```

The screenshot shows the Postman application interface. The top navigation bar includes File, Edit, View, Help, New, Import, Runner, and My Workspace. The main workspace shows two tabs: https://localhost:44375 and https://localhost:44375. A red box highlights the 'PUT' method dropdown. The URL https://localhost:44375/api/todo/1 is entered in the address bar. To the right are Params and Send buttons. Below the address bar, the Body tab is selected, indicated by a blue dot. The raw tab is also highlighted with a red box. The content type is set to JSON (application/json). The body content is a JSON object:

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```

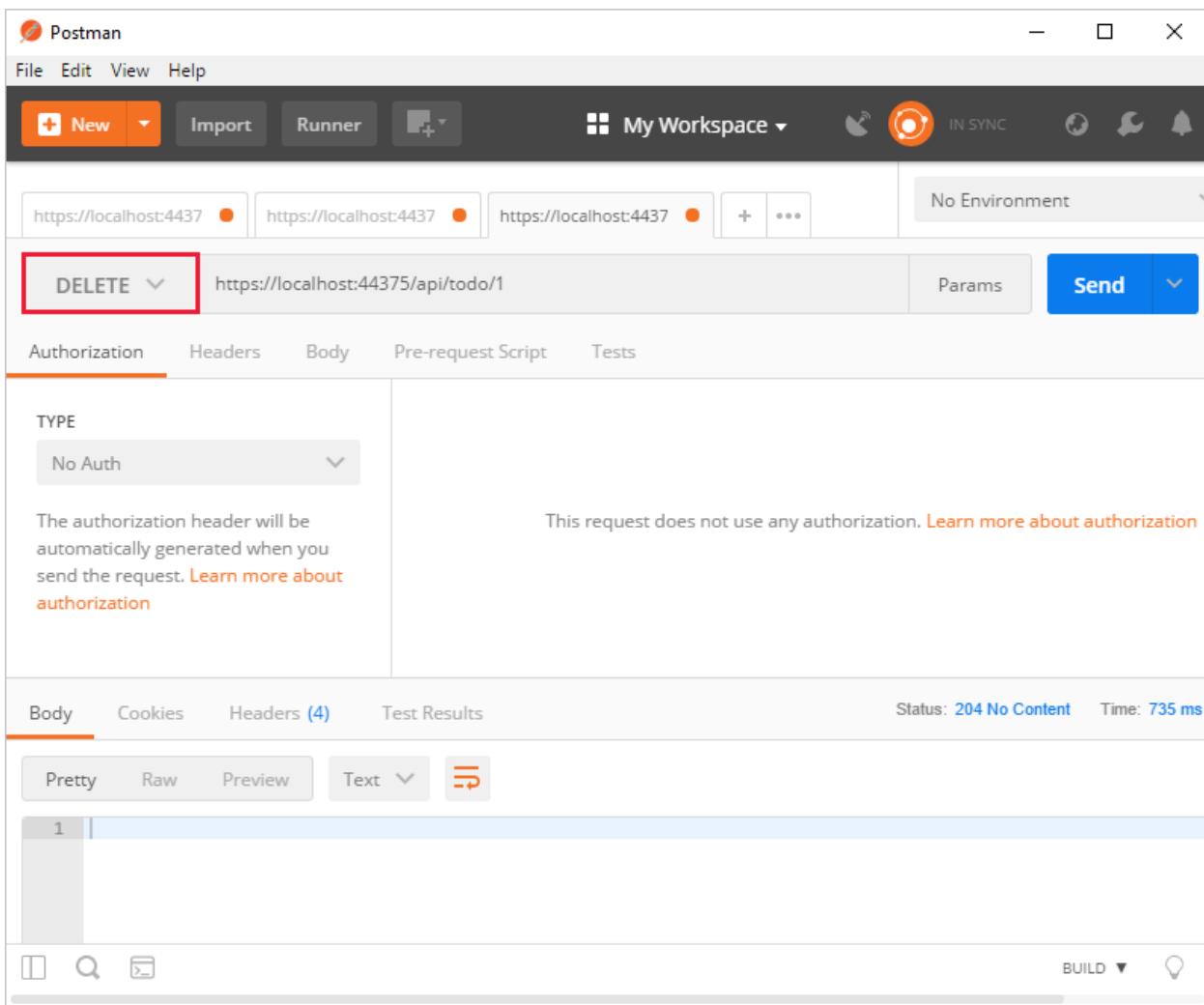
Below the body editor, the status bar shows Status: 204 No Content and Time: 70 ms. The Body section has Pretty, Raw, Preview, and Text tabs, with Text currently selected. The response pane is empty.

删除

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

响应是 204(无内容)。



使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 `Startup.Configure` 中调用 `UseStaticFiles` 和 `UseDefaultFiles` 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 `index.html` 的 HTML 文件添加至项目的 `wwwroot` 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }
    </style>

```

```

        }

    table {
        font-family: Arial, sans-serif;
        border: 1px solid;
        border-collapse: collapse;
    }

    th {
        background-color: #0066CC;
        color: white;
    }

    td {
        border: 1px solid;
        padding: 5px;
    }

```

</style>

</head>

<body>

<h1>To-do CRUD</h1>

<h3>Add</h3>

<form action="javascript:void(0);" method="POST" onsubmit="addItem()">

<input type="text" id="add-name" placeholder="New to-do">

<input type="submit" value="Add">

</form>

<div id="spoiler">

<h3>Edit</h3>

<form class="my-form">

<input type="hidden" id="edit-id">

<input type="checkbox" id="edit-isComplete">

<input type="text" id="edit-name">

<input type="submit" value="Edit">

✖

</form>

</div>

<p id="counter"></p>

<table>

<tr>

<th>Is Complete</th>

<th>Name</th>

<th></th>

<th></th>

</tr>

<tbody id="todos"></tbody>

</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
 integrity="sha256-FgpCb/KJQlLnF0u91ta32o/NMZxltwRo8QtmkMRdAu8="
 crossorigin="anonymous"></script>

<script src="site.js"></script>

</body>

</html>

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
    const el = $('#counter');
    let name = 'to-do';
    if (data) {
        if (data > 1) {
            name = 'to-dos';
        }
    }
    el.text(name + ': ' + data);
}

```

```

        }
        el.text(data + ' ' + name);
    } else {
        el.html('No ' + name);
    }
}

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';

                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
        }
    });
}

function addNewItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            $('#edit-name').val(item.name);
        }
    });
}

```

```

        $('#edit-id').val(item.id);
        $('#edit-isComplete').val(item.isComplete);
    });
});

$('#spoiler').css({ 'display': 'block' });

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });

    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```

$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';

                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
        }
    });
}

```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```
$.ajax({
    url: uri + '/' + $('#edit-id').val(),
    type: 'PUT',
    accepts: 'application/json',
    contentType: 'application/json',
    data: JSON.stringify(item),
    success: function (result) {
        getData();
    }
});
```

删除待办事项

若要删除待办事项, 请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```
$.ajax({
    url: uri + '/' + id,
    type: 'DELETE',
    success: function (result) {
        getData();
    }
});
```

后续步骤

- 有关使用永久数据库的详细信息, 请参阅:
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)
- 有关部署 API 的信息(包括部署到 Azure 应用服务), 请参阅[托管和部署](#)。
- [查看或下载示例代码](#)。请参阅[如何下载](#)。

使用 ASP.NET Core 和 Visual Studio Code 创建 Web API

2018/5/17 • 17 min to read • [Edit Online](#)

作者 : [Rick Anderson](#) 和 [Mike Wasson](#)

本教程将生成一个用于管理“待办事项”列表的 Web API。不构造 UI。

本教程提供 3 个版本：

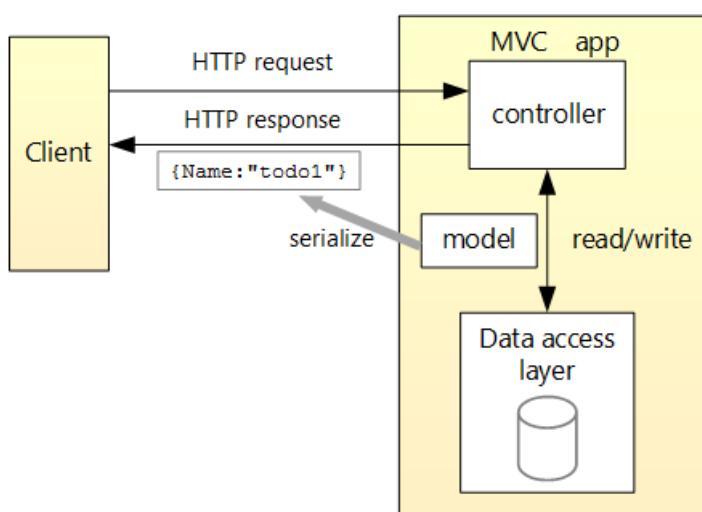
- macOS、Linux、Windows: 使用 Visual Studio Code 创建 Web API (本教程)
- macOS: [使用 Visual Studio for Mac 创建 Web API](#)
- Windows: [使用 Visual Studio for Windows 创建 Web API](#)

概述

本教程将创建以下 API：

API	描述	请求正文	响应正文
GET /api/todo	获取所有待办事项	无	待办事项的数组
GET /api/todo/{id}	按 ID 获取项	无	待办事项
POST /api/todo	添加新项	待办事项	待办事项
PUT /api/todo/{id}	更新现有项	待办事项	无
DELETE /api/todo/{id}	删除项	无	无

下图显示了应用的基本设计。



- 该客户端是使用 Web API(移动应用、浏览器等)的对象。本教程不会创建客户端。[Postman](#) 或 [curl](#) 是用作测试应用的客户端。
- 模型是表示应用程序中的数据的对象。在此示例中，唯一的模型是待办事项。模型表示为 C# 类，也称为 Plain Old C# Object (POCO)。

- 控制器是处理 HTTP 请求并创建 HTTP 响应的对象。此应用程序具有单个控制器。
- 为了简化教程，应用不会使用永久数据库。示例应用将待办事项存储在内存数据库中。

系统必备

Install the following:

- .NET Core SDK 2.0 or later
- Visual Studio Code
- C# for Visual Studio Code
- .NET Core SDK 2.1 RC1 or later
- Visual Studio Code
- C# for Visual Studio Code

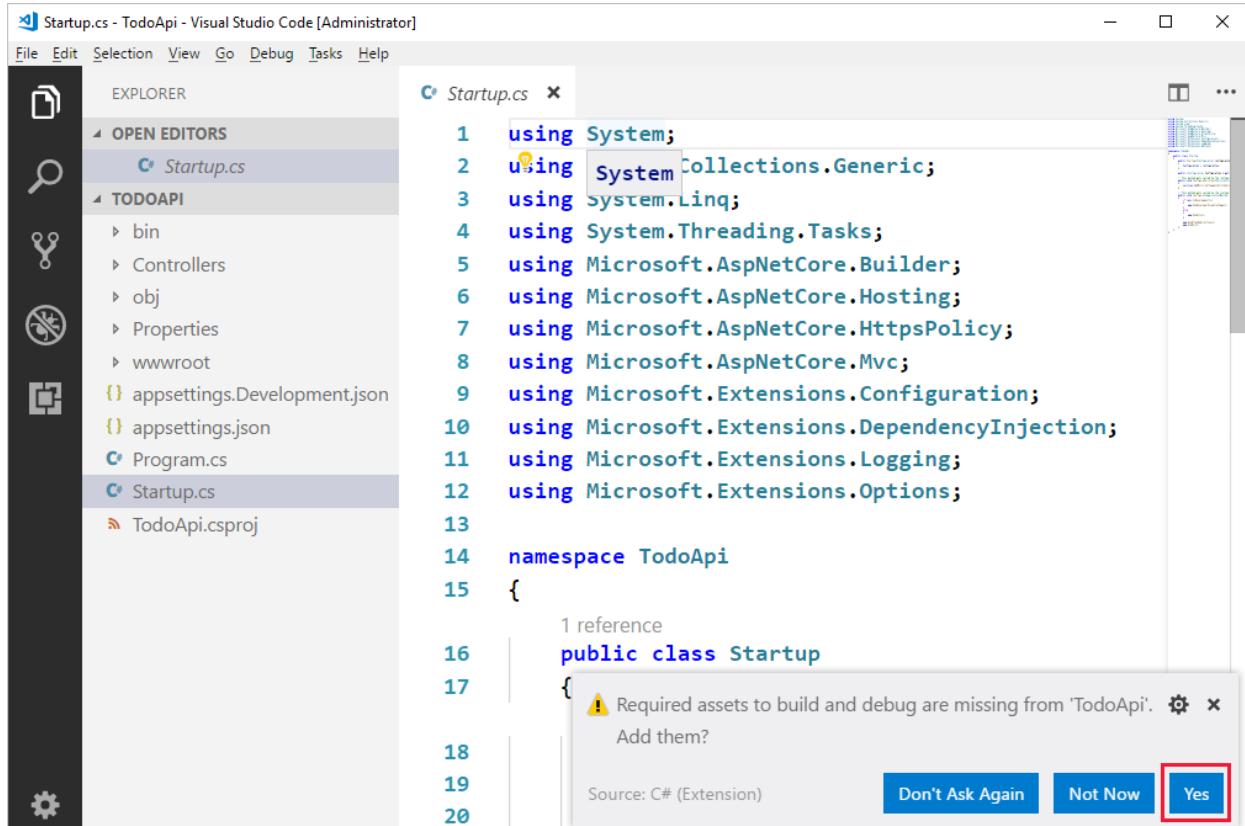
创建项目

从控制台运行以下命令：

```
dotnet new webapi -o TodoApi
code TodoApi
```

在 Visual Studio Code (VS Code) 中打开 TodoApi 文件夹。选择 Startup.cs 文件。

- 对于警告消息 -““TodoApi” 中缺少进行生成和调试所需的资产。是否添加它们？”，请选择“是”
- 对于信息性消息 -“存在未解析的依赖项”，请选择“还原”。



按“调试”(F5) 生成并运行程序。在浏览器中导航到 <http://localhost:5000/api/values>。显示以下输出：

```
["value1","value2"]
```

有关 VS Code 的使用技巧, 请参阅 [Visual Studio Code 帮助](#)。

添加对 Entity Framework Core 的支持

在 ASP.NET Core 2.0 中创建新项目会在 TodoApi.csproj 文件中添加 [Microsoft.AspNetCore.All](#) 包引用:

```
[!code-xml]
```

在 ASP.NET Core 2.1 或更改版本中创建新项目会在 TodoApi.csproj 文件中添加 [Microsoft.AspNetCore.App](#) 包引用:

```
[!code-xml]
```

无需单独安装 [Entity Framework Core InMemory](#) 数据库提供程序。此数据库提供程序允许将 Entity Framework Core 和内存数据库一起使用。

添加模型类

模型是表示应用中的数据的对象。在此示例中, 唯一的模型是待办事项。

添加名为“模型”的文件夹。可将模型类置于项目的任意位置, 但按照惯例会使用“模型”文件夹。

添加带有以下代码的 `TodoItem` 类:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

创建 `TodoItem` 时, 数据库将生成 `Id`。

创建数据库上下文

数据库上下文是为给定数据模型协调 Entity Framework 功能的主类。将通过从 `Microsoft.EntityFrameworkCore.DbContext` 类派生的方式创建此类。

在“模型”文件夹中添加 `TodoContext` 类:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {

        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

注册数据库上下文

在该步骤中，向[依赖关系注入](#)容器注册数据库上下文。向依赖关系注入 (DI) 容器注册的服务（例如数据库上下文）可供控制器使用。

使用[依赖关系注入](#)的内置支持将数据库上下文注册到服务容器。将 Startup.cs 文件的内容替换为以下代码：

[!code-csharp]

[!code-csharp]

前面的代码：

- 删除未使用的代码。
- 指定将内存数据库注入到服务容器中。

添加控制器

在“控制器”文件夹中，创建名为 `TodoController` 的类。用以下代码替代其内容：

[!code-csharp]

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。

[!code-csharp]

前面的代码定义了没有方法的 API 控制器类。在接下来的部分中，将添加方法来实现 API。采用 `[ApiController]` 特性批注类，以启用一些方便的功能。若要详细了解由这些特性启用的功能，请参阅[使用 ApiControllerAttribute 批注类](#)。

控制器的构造函数使用[依赖关系注入](#)将数据库上下文 (`TodoContext`) 注入到控制器中。数据库上下文将在控制器中的每个 `CRUD` 方法中使用。构造函数将一个项（如果不存在）添加到内存数据库。

获取待办事项

若要获取待办事项，请将下面的方法添加到 `TodoController` 类中：

[!code-csharp]

[!code-csharp]

这些方法实现两种 GET 方法：

- `GET /api/todo`
- `GET /api/todo/{id}`

以下是 `GetAll` 方法的 HTTP 响应示例：

```
[  
  {  
    "id": 1,  
    "name": "Item1",  
    "isComplete": false  
  }  
]
```

稍后将在本教程中演示如何使用 [Postman](#) 或 [curl](#) 查看 HTTP 响应。

路由和 URL 路径

`[HttpGet]` 特性表示对 HTTP GET 请求进行响应的方法。每个方法的 URL 路径构造如下所示：

- 在控制器的 `Route` 属性中采用模板字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

- 将 `[controller]` 替换为控制器的名称，即在控制器类名称中去掉“Controller”后缀。对于此示例，控制器类名称为“Todo”控制器，根名称为“todo”。ASP.NET Core 路由不区分大小写。
- 如果 `[HttpGet]` 特性具有路由模板（如 `[HttpGet("/products")]`），则将它追加到路径。此示例不使用模板。有关详细信息，请参阅[使用 Http \[Verb\] 特性的特性路由](#)。

在下面的 `GetById` 方法中，`"{id}"` 是待办事项的唯一标识符的占位符变量。调用 `GetById` 时，它会将 URL 中 `"{id}"` 的值分配给方法的 `id` 参数。

```
[!code-csharp]
```

```
[!code-csharp]
```

`Name = "GetTodo"` 创建具名路由。具名路由：

- 使应用程序使用路由名称创建 HTTP 链接。
- 将在本教程的后续部分中介绍。

返回值

`GetAll` 方法返回一个 `TodoItem` 对象的集合。MVC 自动将对象序列化为 [JSON](#)，并将 JSON 写入响应消息的正文中。在假设没有未经处理的异常的情况下，此方法的响应代码为 200。未经处理的异常将转换为 5xx 错误。

相反， `GetById` 方法返回多个常规的 [IActionResult](#) 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `Ok`，则产生 HTTP 200 响应。

相反， `GetById` 方法返回多个 [ActionResult<T>](#) 类型，它表示一系列返回类型。 `GetById` 具有两个不同的返回类型：

- 如果没有任何项与请求的 ID 匹配，此方法将返回 404 错误。返回 `NotFound` 可以返回 HTTP 404 响应。
- 否则，此方法将返回具有 JSON 响应正文的 200。返回 `item` 则产生 HTTP 200 响应。

启动应用

在 VS Code 中，按 F5 启动应用。导航到 <http://localhost:5000/api/todo>（我们刚刚创建的 `Todo` 控制器）。

使用 jQuery 调用 Web API

在本部分中，添加了 HTML 页面使用 jQuery 调用 Web API。jQuery 启动请求，并用 API 响应中的详细信息更新页面。

配置项目提供静态文件并启用默认文件映射。通过在 Startup.Configure 中调用 `UseStaticFiles` 和 `UseDefaultFiles` 扩展方法完成这一点。有关详细信息，请参阅[静态文件](#)。

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseMvc();
}
```

将一个名为 index.html 的 HTML 文件添加至项目的 wwwroot 目录。用以下标记替代其内容：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Edit">
            <a onclick="closeInput()" aria-label="Close">✖</a>
        </form>
    </div>
</body>
```

```

<p id="counter"></p>

<table>
  <tr>
    <th>Is Complete</th>
    <th>Name</th>
    <th></th>
    <th></th>
  </tr>
  <tbody id="todos"></tbody>
</table>

<script src="https://code.jquery.com/jquery-3.3.1.min.js"
        integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
        crossorigin="anonymous"></script>
<script src="site.js"></script>
</body>
</html>

```

将名为 site.js 的 JavaScript 文件添加至项目的 wwwroot 目录。用以下代码替代其内容：

```

const uri = 'api/todo';
let todos = null;
function getCount(data) {
  const el = $('#counter');
  let name = 'to-do';
  if (data) {
    if (data > 1) {
      name = 'to-dos';
    }
    el.text(data + ' ' + name);
  } else {
    el.html('No ' + name);
  }
}

$(document).ready(function () {
  getData();
});

function getData() {
  $.ajax({
    type: 'GET',
    url: uri,
    success: function (data) {
      $('#todos').empty();
      getCount(data.length);
      $.each(data, function (key, item) {
        const checked = item.isComplete ? 'checked' : '';
        $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
          '<td>' + item.name + '</td>' +
          '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
          '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
          '</tr>').appendTo($('#todos'));
      });
      todos = data;
    }
  });
}

function addItem() {
  const item = {
    'name': $('#add-name').val(),
    'isComplete': false
  };
}

```

```

$.ajax({
    type: 'POST',
    accepts: 'application/json',
    url: uri,
    contentType: 'application/json',
    data: JSON.stringify(item),
    error: function (jqXHR, textStatus, errorThrown) {
        alert('here');
    },
    success: function (result) {
        getData();
        $('#add-name').val('');
    }
});

function deleteItem(id) {
    $.ajax({
        url: uri + '/' + id,
        type: 'DELETE',
        success: function (result) {
            getData();
        }
    });
}

function editItem(id) {
    $.each(todos, function (key, item) {
        if (item.id === id) {
            $('#edit-name').val(item.name);
            $('#edit-id').val(item.id);
            $('#edit-isComplete').val(item.isComplete);
        }
    });
    $('#spoiler').css({ 'display': 'block' });
}

$('.my-form').on('submit', function () {
    const item = {
        'name': $('#edit-name').val(),
        'isComplete': $('#edit-isComplete').is(':checked'),
        'id': $('#edit-id').val()
    };

    $.ajax({
        url: uri + '/' + $('#edit-id').val(),
        type: 'PUT',
        accepts: 'application/json',
        contentType: 'application/json',
        data: JSON.stringify(item),
        success: function (result) {
            getData();
        }
    });

    closeInput();
    return false;
});

function closeInput() {
    $('#spoiler').css({ 'display': 'none' });
}

```

可能需要更改 ASP.NET Core 项目的启动设置，以便对 HTML 页面进行本地测试。打开项目“属性”目录中的 launchSettings.json。删除 `launchUrl` 以便在项目的默认文件 index.html 强制打开应用。

有多种方式可以获取 jQuery。在前面的代码片段中，库是从 CDN 中加载的。此示例是一个使用 jQuery 调用 API 的完整 CRUD 示例。此实例中有很多其他功能可以丰富你的体验。以下是关于调用 API 的说明。

获取待办事项的列表

若要获取待办事项列表，请将 HTTP GET 请求发送到 /api/todo。

jQuery `ajax` 函数将 AJAX 请求发送至 API，这将返回代表对象或数组的 JSON。此函数可以处理所有形式的 HTTP 交互、将 HTTP 请求发送至指定的 `url`。`GET` 被用作 `type`。如果请求成功，则调用 `success` 回调函数。在该回调中使用待办事项信息更新 DOM。

```
$(document).ready(function () {
    getData();
});

function getData() {
    $.ajax({
        type: 'GET',
        url: uri,
        success: function (data) {
            $('#todos').empty();
            getCount(data.length);
            $.each(data, function (key, item) {
                const checked = item.isComplete ? 'checked' : '';

                $('<tr><td><input disabled="true" type="checkbox" ' + checked + '></td>' +
                    '<td>' + item.name + '</td>' +
                    '<td><button onclick="editItem(' + item.id + ')">Edit</button></td>' +
                    '<td><button onclick="deleteItem(' + item.id + ')">Delete</button></td>' +
                    '</tr>').appendTo($('#todos'));
            });
            todos = data;
        }
    });
}
```

添加待办事项

若要添加代办实现，请将 HTTP POST 请求发送至 /api/todo/。请求正文应包含待办对象。`ajax` 函数使用 `POST` 调用 API。对于 `POST` 和 `PUT` 请求，请求正文表示发送至 API 的数据。API 需要 JSON 请求正文。将 `accepts` 和 `contentType` 设为 `application/json`，以便分别对接收和发送的媒体类型进行分类。使用 `JSON.stringify` 将数据转换为 JSON 对象。当 API 返回成功状态的代码时，将调用 `getData` 函数来更新 HTML 表。

```

function addItem() {
    const item = {
        'name': $('#add-name').val(),
        'isComplete': false
    };

    $.ajax({
        type: 'POST',
        accepts: 'application/json',
        url: uri,
        contentType: 'application/json',
        data: JSON.stringify(item),
        error: function (jqXHR, textStatus, errorThrown) {
            alert('here');
        },
        success: function (result) {
            getData();
            $('#add-name').val('');
        }
    });
}

```

更新待办事项

待办事项的更新与添加非常类似，因为两者都依赖于请求正文。这种情况下，两者间真正的区别在于添加该项的唯一标识符时会更改 `url`，且 `type` 为 `PUT`。

```

$.ajax({
    url: uri + '/' + $('#edit-id').val(),
    type: 'PUT',
    accepts: 'application/json',
    contentType: 'application/json',
    data: JSON.stringify(item),
    success: function (result) {
        getData();
    }
});

```

删除待办事项

若要删除待办事项，请将 AJAX 调用上的 `type` 设为 `DELETE` 并指定该项在 URL 中的唯一标识符。

```

$.ajax({
    url: uri + '/' + id,
    type: 'DELETE',
    success: function (result) {
        getData();
    }
});

```

实现其他的 CRUD 操作

在以下部分中，将 `Create`、`Update` 和 `Delete` 方法添加到控制器。

创建

添加以下 `Create` 方法：

[!code-csharp]

正如 `[HttpPost]` 属性所指示，前面的代码是 HTTP POST 方法。`[FromBody]` 特性告诉 MVC 从 HTTP 请求正文获取待办事项的值。

[!code-csharp]

正如 [HttpPost] 属性所指示, 前面的代码是 HTTP POST 方法。MVC 从 HTTP 请求正文获取待办事项的值。

CreatedAtRoute 方法:

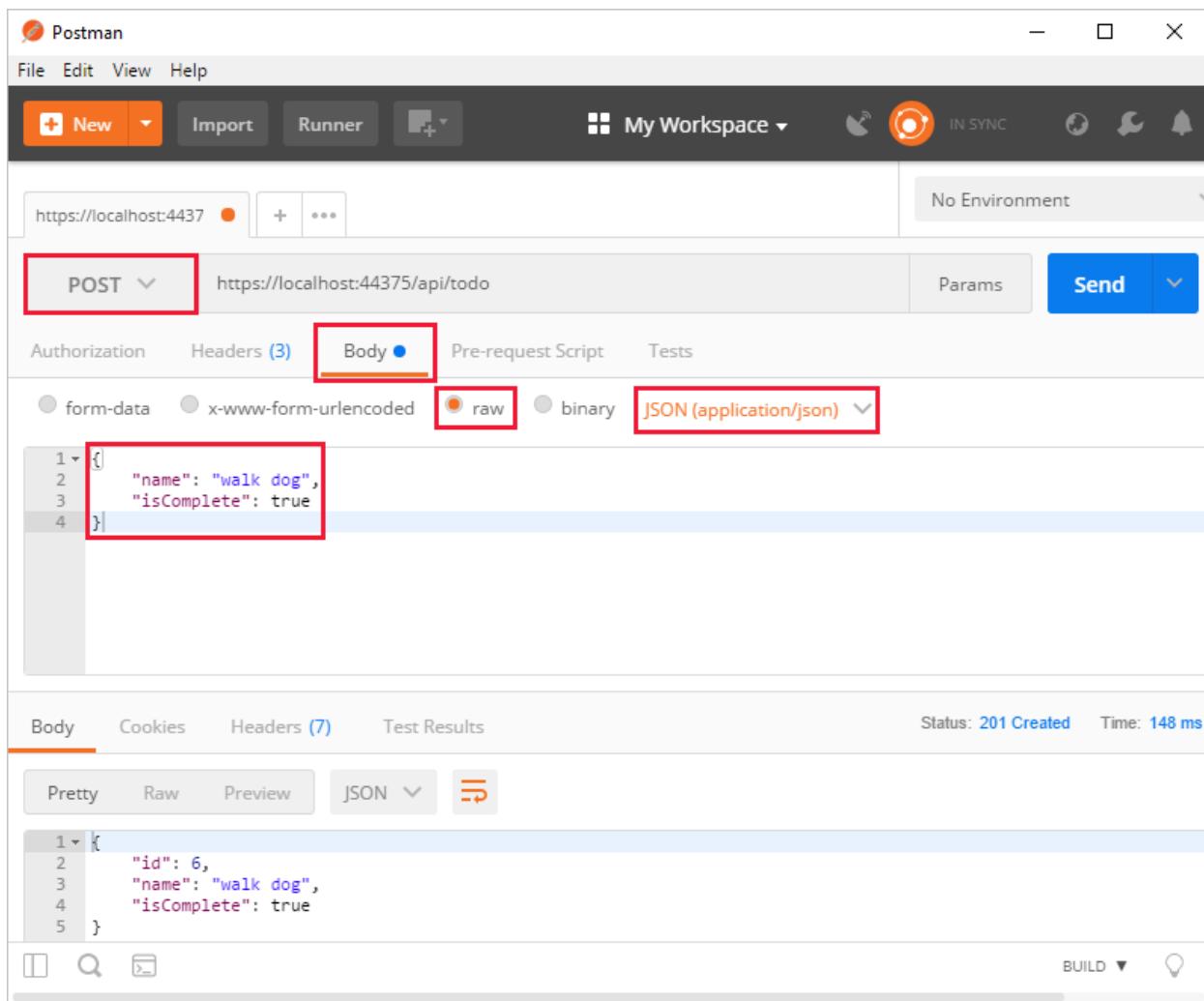
- 返回 201 响应。HTTP 201 是在服务器上创建新资源的 HTTP POST 方法的标准响应。
- 向响应添加位置标头。位置标头指定新建的待办事项的 URI。请参阅 10.2.2 201 已创建。
- 使用名为 route 的“GetTodo”来创建 URL。已在 `GetById` 中定义名为 route 的“GetTodo”:

[!code-csharp]

[!code-csharp]

使用 Postman 发送创建请求

- 启动该应用程序。
- 打开 Postman。



- 更新 localhost URL 中的端口号。
- 将 HTTP 方法设置为 POST。
- 单击“正文”选项卡。
- 选择“原始”单选按钮。
- 将类型设置为 JSON (application/json)
- 输入包含待办事项的请求正文, 类似以下 JSON:

```
{  
  "name": "walk dog",  
  "isComplete": true  
}
```

- 单击“发送”按钮。

提示

如果单击“发送”后没有响应，则禁用“SSL 证书验证”选项。在“文件”>“设置”下可以找到该选项。在禁用该设置后，再次单击“发送”按钮。

单击“响应”窗格中的“标头”选项卡，然后复制位置标头值：

The screenshot shows the Postman application interface. A POST request is being made to `https://localhost:44375/api/todo`. The request body is set to raw JSON:

```
1 {  
2   "name": "walk dog",  
3   "isComplete": true  
4 }
```

The "Headers" tab is selected in the response section, and its value is highlighted with a red box:

Content-Type → application/json; charset=utf-8

Date → Fri, 27 Apr 2018 18:32:32 GMT

Location → <https://localhost/api/Todo/6>

位置标头 URI 可用于访问新项。

更新

添加以下 `Update` 方法：

```
[!code-csharp]
```

```
[!code-csharp]
```

`Update` 与 `Create` 类似，但是使用的是 HTTP PUT。响应是 204(无内容)。根据 HTTP 规范，PUT 请求需要客户端发送整个更新的实体，而不仅仅是增量。若要支持部分更新，请使用 HTTP PATCH。

使用 Postman 将待办事项的名称更新为“带猫出去散步”：

The screenshot shows the Postman application interface. At the top, there are tabs for 'File', 'Edit', 'View', and 'Help'. Below the tabs, there are buttons for 'New', 'Import', 'Runner', and 'My Workspace'. The 'My Workspace' dropdown is set to 'IN SYNC'. On the right side of the header, there are icons for environment management, sync status, and notifications.

In the main workspace, there are two tabs: 'https://localhost:4437' and 'https://localhost:4437'. A red box highlights the 'PUT' method dropdown. The URL 'https://localhost:44375/api/todo/1' is entered in the address bar. To the right of the URL, there are buttons for 'Params' and 'Send'.

Below the URL, there are tabs for 'Authorization', 'Headers (3)', 'Body (●)', 'Pre-request Script', and 'Tests'. The 'Body' tab is selected, indicated by a blue dot. A red box highlights the 'Body' tab. Under the 'Body' tab, there are four options: 'form-data', 'x-www-form-urlencoded', 'raw', and 'binary'. The 'raw' option is selected, indicated by a blue dot, and a red box highlights it. To the right of 'raw', there is a dropdown menu set to 'JSON (application/json)'.

The 'Body' section contains a code editor with the following JSON payload:

```
1 {  
2   "name": "walk cat",  
3   "isComplete": true  
4 }
```

Below the code editor, there are tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results'. The 'Body' tab is selected, indicated by an orange underline. To the right, the status is shown as 'Status: 204 No Content' and 'Time: 70 ms'. Below these tabs, there are buttons for 'Pretty', 'Raw', 'Preview', 'Text', and a copy icon.

删除

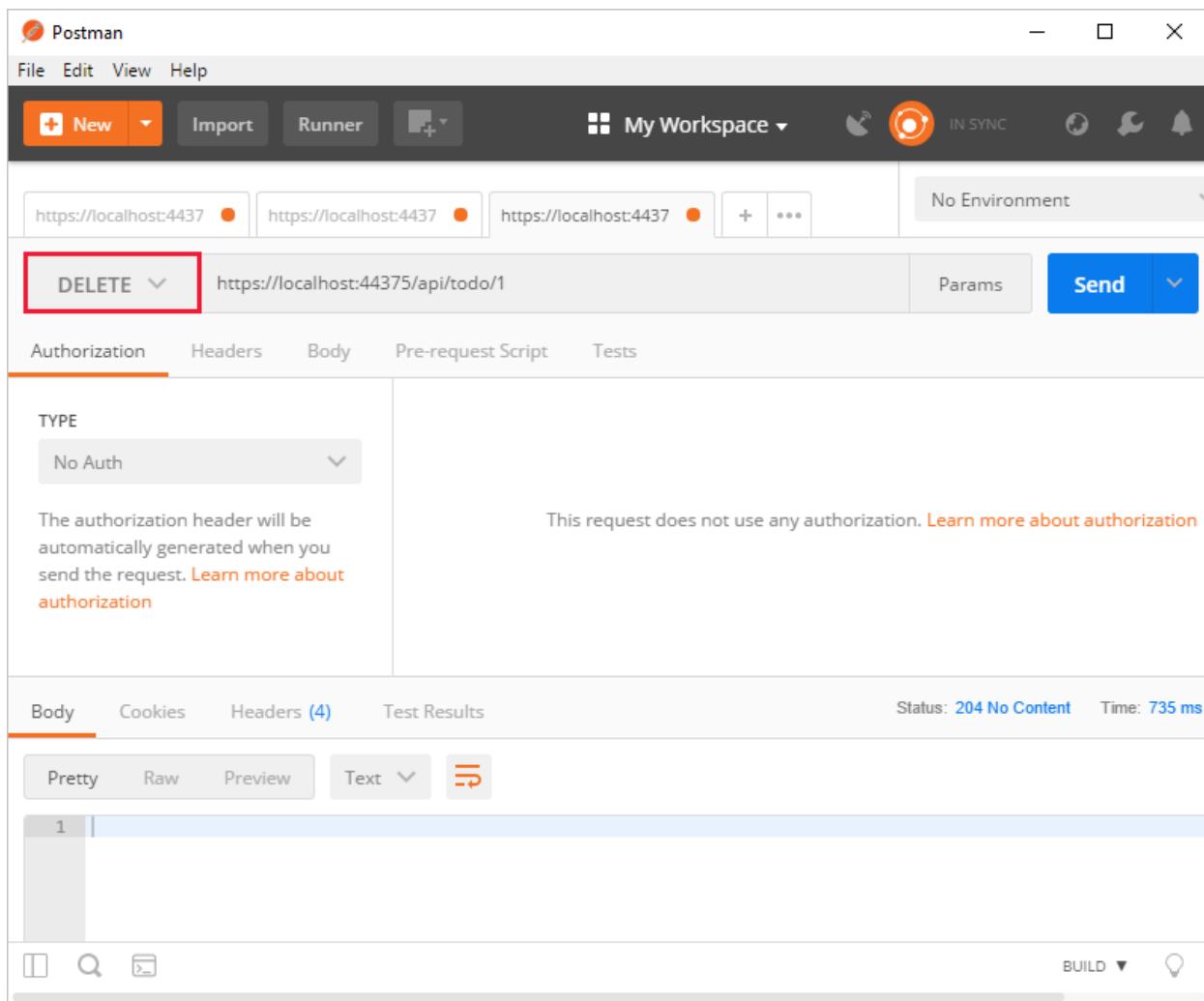
添加以下 `Delete` 方法：

```
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);
    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();
    return NoContent();
}
```

`Delete` 响应是 204(无内容)。

使用 Postman 删除待办事项：



Visual Studio Code 帮助

- [入门](#)
- [调试](#)
- [集成终端](#)
- [键盘快捷键
 - \[macOS 键盘快捷方式\]\(#\)
 - \[Linux 键盘快捷键\]\(#\)
 - \[Windows 键盘快捷键\]\(#\)](#)

后续步骤

- 有关使用永久数据库的详细信息, 请参阅:
 - [使用 ASP.NET Core 创建 Razor 页面 Web 应用](#)
 - [在 ASP.NET Core 中使用数据](#)
- [使用 Swagger 的 ASP.NET Core Web API 帮助页](#)
- [路由到控制器操作](#)
- [使用 ASP.NET Core 构建 Web API](#)
- [控制器操作返回类型](#)
- 有关部署 API 的信息(包括部署到 Azure 应用服务), 请参阅[托管和部署](#)。

- 查看或下载示例代码。请参阅[如何下载](#)。

使用 ASP.NET Core 为本机移动应用创建后端服务

2018/5/14 • 9 min to read • [Edit Online](#)

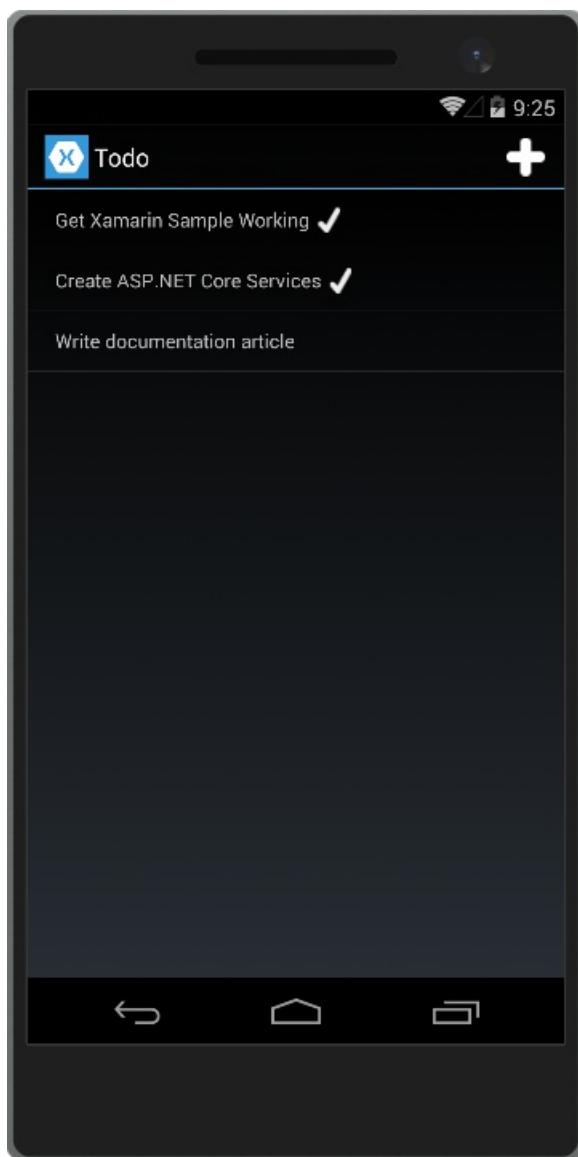
作者: [Steve Smith](#)

移动应用可与 ASP.NET Core 后端服务轻松通信。

[查看或下载后端服务代码示例](#)

本机移动应用示例

本教程演示如何创建使用 ASP.NET Core MVC 支持本机移动应用的后端服务。它使用 [Xamarin Forms ToDoRest 应用](#) 作为其本机客户端，其中包括 Android、iOS、Windows Universal 和 Window Phone 设备的单独本机客户端。你可以遵循链接中的教程来创建本机应用程序（并安装需要的免费 Xamarin 工具），以及下载 Xamarin 示例解决方案。Xamarin 示例包含一个 ASP.NET Web API 2 服务项目，使用本文中的 ASP.NET Core 应用替换（客户端无需进行任何更改）。



功能

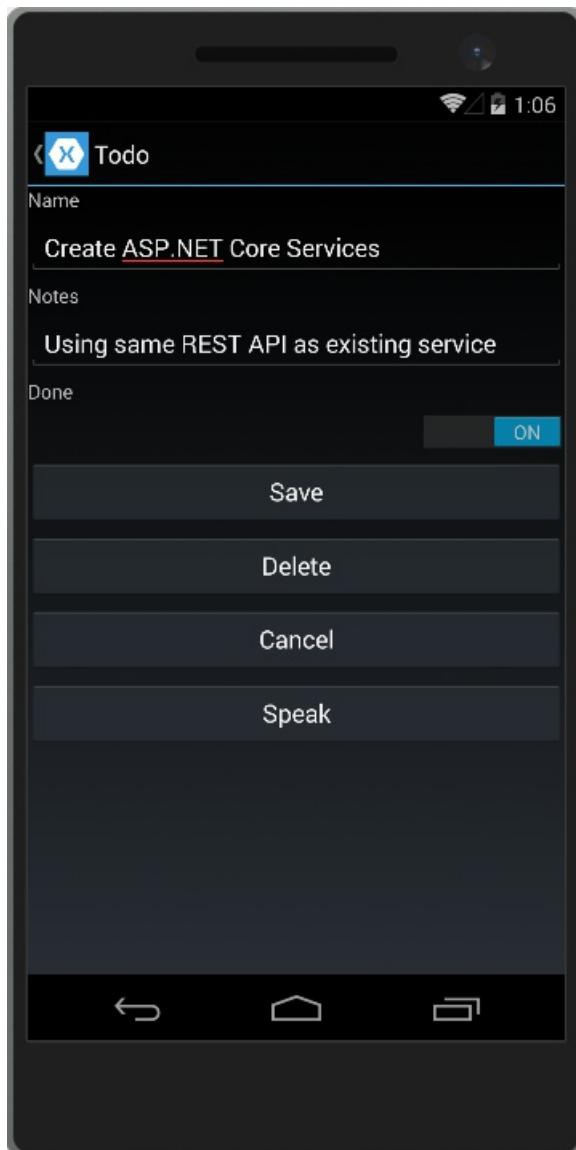
ToDoRest 应用支持列出、添加、删除和更新待办事项。每个项都有一个 ID、Name(名称)、Notes(说明)以及一个指示该项是否已完成的属性 Done。

待办事项的主视图如上所示，列出每个项的名称，并使用复选标记指示它是否已完成。

点击  图标打开“添加项”对话框：



点击主列表屏幕上的项将打开一个编辑对话框，在其中可以修改项的名称、说明以及是否完成，或删除项目：



此示例默认配置为使用托管在 developer.xamarin.com 上的后端服务，允许只读操作。若要使用在你计算机上运行的下一节创建的 ASP.NET Core 应用对其进行测试，你需要更新应用程序的 `RestUrl` 常量。导航到 `ToDoREST` 项目，然后打开 `Constants.cs` 文件。使用包含计算机 IP 的 URL 地址替换 `RestUrl`（不是 `localhost` 或 `127.0.0.1`，因为此地址用于从设备模拟器中，而不是从你的计算机中访问）。请包括端口号（`5000`）。为了测试你的服务能否在设备上正常运行，请确保没有活动的防火墙阻止访问此端口。

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

创建 ASP.NET Core 项目

在 Visual Studio 中创建一个新的 ASP.NET Core Web 应用程序。选择 Web API 模板和 No Authentication（无身份验证）。将项目命名为 `ToDoApi`。

Select a template:

ASP.NET Core Templates



Empty



Web API



Web Application

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET MVC Views and Controllers.

[Learn more](#)

[Change Authentication](#)

Authentication: **No Authentication**

Microsoft Azure

Host in the cloud

[App Service](#) ▾

OK

Cancel

对于向端口 5000 进行的请求，应用程序均需作出响应。更新 Program.cs，使其包含 `.UseUrls("http://*:5000")`，以便实现以下操作：

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

注意

请确保直接运行应用程序，而不是在 IIS Express 后运行，因为在默认情况下，后者会忽略非本地请求。从命令提示符处运行 `dotnet run`，或从 Visual Studio 工具栏中的“调试目标”下拉列表中选择应用程序名称配置文件。

添加一个模型类来表示待办事项。使用 `[Required]` 属性标记必需字段：

```
using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

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

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

        public bool Done { get; set; }
    }
}
```

API 方法需要通过某种方式处理数据。使用原始 Xamarin 示例所用的 `IToDoRepository` 接口：

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

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}
```

在此示例中，该实现仅使用一个专用项集合：

```
using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
        }
    }
}
```

```
public ToDoItem Find(string id)
{
    return _toDoList.FirstOrDefault(item => item.ID == id);
}

public void Insert(ToDoItem item)
{
    _toDoList.Add(item);
}

public void Update(ToDoItem item)
{
    var todoItem = this.Find(item.ID);
    var index = _toDoList.IndexOf(todoItem);
    _toDoList.RemoveAt(index);
    _toDoList.Insert(index, item);
}

public void Delete(string id)
{
    _toDoList.Remove(this.Find(id));
}

private void InitializeData()
{
    _toDoList = new List<ToDoItem>();

    var todoItem1 = new ToDoItem
    {
        ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
        Name = "Learn app development",
        Notes = "Attend Xamarin University",
        Done = true
    };

    var todoItem2 = new ToDoItem
    {
        ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
        Name = "Develop apps",
        Notes = "Use Xamarin Studio/Visual Studio",
        Done = false
    };

    var todoItem3 = new ToDoItem
    {
        ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
        Name = "Publish apps",
        Notes = "All app stores",
        Done = false,
    };

    _toDoList.Add(todoItem1);
    _toDoList.Add(todoItem2);
    _toDoList.Add(todoItem3);
}
}
```

在 *Startup.cs* 中配置该实现:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository, ToDoRepository>();
}
```

现可创建 `ToDoItemsController`。

提示

有关创建 Web API 的详细信息, 请参阅[使用 ASP.NET Core MVC 和 Visual Studio 生成首个 Web API](#)。

创建控制器

在项目中添加新控制器 `ToDoItemsController`。它应继承 `Microsoft.AspNetCore.Mvc.Controller`。添加 `[Route]` 属性以指示控制器将处理路径以 `api/todoitems` 开始的请求。路由中的 `[controller]` 标记会被控制器的名称代替(省略 `Controller` 后缀), 这对全局路由特别有用。详细了解 [路由](#)。

控制器需要 `IToDoRepository` 才能正常运行;通过控制器的构造函数请求该类型的实例。在运行时, 此实例将使用框架对 [依赖关系注入](#) 的支持来提供。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly IToDoRepository _ToDoRepository;

        public ToDoItemsController(IToDoRepository ToDoRepository)
        {
            _ToDoRepository = ToDoRepository;
        }
    }
}
```

此 API 支持四个不同的 HTTP 谓词来执行对数据源的 CRUD(创建、读取、更新、删除)操作。最简单的是读取操作, 它对应于 HTTP GET 请求。

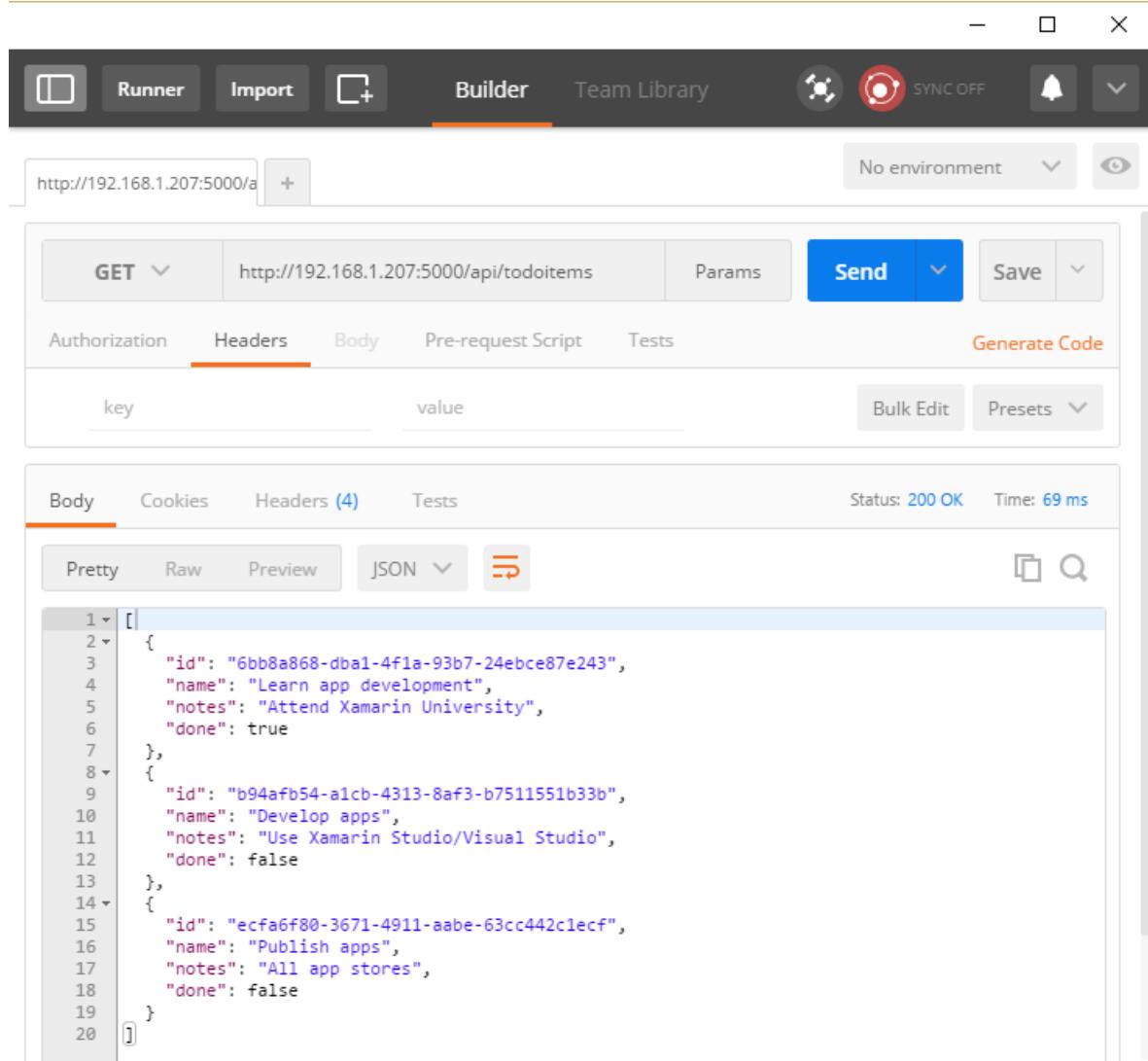
读取项目

要请求项列表, 可对 `List` 方法使用 GET 请求。`[HttpGet]` 方法的 `List` 属性指示此操作应仅处理 GET 请求。此操作的路由是在控制器上指定的路由。你不一定必须将操作名称用作路由的一部分。你只需确保每个操作都有唯一的和明确的路由。路由属性可以分别应用在控制器和方法级别, 以此生成特定的路由。

```
[HttpGet]
public IActionResult List()
{
    return Ok(_ToDoRepository.All);
}
```

`List` 方法返回 200 OK 响应代码和所有 ToDo 项, 并序列化为 JSON。

你可以使用多种工具测试新的 API 方法, 如 [Postman](#), 如此处所示:



The screenshot shows the Postman application window. At the top, there are tabs for Runner, Import, Builder (which is selected), and Team Library. On the right side, there are icons for Sync (OFF), a bell, and a dropdown menu. Below the tabs, the URL is set to <http://192.168.1.207:5000/api/todoitems>. The main interface shows a 'GET' request method selected. Below the URL, there are tabs for Authorization, Headers (which is selected), Body, Pre-request Script, Tests, and Generate Code. Under Headers, there is a table with columns 'key' and 'value'. At the bottom, there are buttons for Bulk Edit and Presets. The status bar at the bottom indicates 'Status: 200 OK' and 'Time: 69 ms'. The Body section displays the JSON response from the API call, which is a list of three todo items:

```
1 [  
2 {  
3   "id": "6bb8a868-dba1-4f1a-93b7-24ebce87e243",  
4   "name": "Learn app development",  
5   "notes": "Attend Xamarin University",  
6   "done": true  
7 },  
8 {  
9   "id": "b94afb54-a1cb-4313-8af3-b7511551b33b",  
10  "name": "Develop apps",  
11  "notes": "Use Xamarin Studio/Visual Studio",  
12  "done": false  
13 },  
14 {  
15  "id": "ecfa6f80-3671-4911-aabe-63cc442c1ecf",  
16  "name": "Publish apps",  
17  "notes": "All app stores",  
18  "done": false  
19 }]  
20 ]
```

创建项目

按照约定, 创建新数据项映射到 HTTP POST 谓词。 `Create` 方法具有应用于该对象的 `[HttpPost]` 属性, 并接受 `ToDoItem` 实例。由于 `item` 参数将在 POST 的正文中传递, 因此该参数用 `[FromBody]` 属性修饰。

在该方法中, 会检查项的有效性和之前是否存在于数据存储, 并且如果没有任何问题, 则使用存储库添加。检查 `ModelState.IsValid` 将执行 [模型验证](#), 应该在每个接受用户输入的 API 方法中执行此步骤。

```
[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _ToDoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TodoItemIDInUse.ToString());
        }
        _ToDoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

示例中使用一个枚举，后者包含传递到移动客户端的错误代码：

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

使用 Postman 测试添加新项，选择 POST 谓词并在请求正文中以 JSON 格式提供新对象。你还应添加一个请求标头指定 `Content-Type` 为 `application/json`。

The screenshot shows the Postman application interface. At the top, there are tabs for Runner, Import, Builder (which is selected), Team Library, and various status indicators like SYNC OFF and a bell icon. Below the tabs, the URL is set to `http://192.168.1.207:5000/a`. The main area shows a POST request to `http://192.168.1.207:5000/api/todoitems`. The Body tab is selected, showing a raw JSON payload:

```
1 {  
2     "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",  
3     "Name": "A Test Item",  
4     "Notes": "asdf",  
5     "Done": false  
6 }
```

Below the request, the response section shows a status of 200 OK and a time of 227 ms. The Body tab is selected again, displaying the received JSON response:

```
1 {  
2     "id": "6bb8b868-dba1-4f1a-93b7-24ebce87243",  
3     "name": "A Test Item",  
4     "notes": "asdf",  
5     "done": false  
6 }
```

该方法返回在响应中新建的项。

更新项目

通过使用 HTTP PUT 请求来修改记录。除了此更改之外, `Edit` 方法几乎与 `Create` 完全相同。请注意, 如果未找到记录, 则 `Edit` 操作将返回 `NotFound` (404) 响应。

```
[HttpPut]  
public IActionResult Edit([FromBody] ToDoItem item)  
{  
    try  
    {  
        if (item == null || !ModelState.IsValid)  
        {  
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());  
        }  
        var existingItem = _ToDoRepository.Find(item.ID);  
        if (existingItem == null)  
        {  
            return NotFound(ErrorCode.RecordNotFound.ToString());  
        }  
        _ToDoRepository.Update(item);  
    }  
    catch (Exception)  
    {  
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());  
    }  
    return NoContent();  
}
```

若要使用 Postman 进行测试，将谓词更改为 PUT。在请求正文中指定要更新的对象数据。

The screenshot shows the Postman application window. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), 'Team Library', and various status indicators like 'SYNC OFF'. The main area has a URL input field containing 'http://192.168.1.207:5000/api/todolist' and a 'Send' button. Below the URL, there's a table for parameters with columns 'key' and 'value'. Under the 'Body' tab, the 'raw' option is selected, and the JSON payload is:

```
1 {  
2     "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",  
3     "Name": "An UPDATED Test Item",  
4     "Notes": "Some updated notes",  
5     "Done": true  
6 }
```

Below the body, the response section shows a status of 204 No Content and a time of 91 ms. The 'Pretty' view of the response is empty.

为了与预先存在的 API 保持一致，此方法在成功时返回 `NoContent` (204) 响应。

删除项目

删除记录可以通过向服务发出 DELETE 请求并传递要删除项的 ID 来完成。与更新一样，请求的项不存在时会收到 `NotFound` 响应。请求成功会得到 `NoContent` (204) 响应。

```
[HttpDelete("{id}")]  
public IActionResult Delete(string id)  
{  
    try  
    {  
        var item = _todoRepository.Find(id);  
        if (item == null)  
        {  
            return NotFound(ErrorCode.RecordNotFound.ToString());  
        }  
        _todoRepository.Delete(id);  
    }  
    catch (Exception)  
    {  
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());  
    }  
    return NoContent();  
}
```

请注意，在测试删除功能时，请求正文中不需要任何内容。

The screenshot shows the Postman application interface. At the top, there are tabs for Runner, Import, Builder (which is selected), and Team Library. There are also icons for sync, notifications, and a dropdown menu. The URL bar shows 'http://192.168.1.207:5000/a'. Below the URL bar, there are buttons for 'Send' and 'Save'. The main area is divided into sections: Authorization, Headers (2), Body (selected), Pre-request Script, Tests, and Generate Code. Under Headers, there are two entries: 'form-data' and 'x-www-form-urlencoded'. Under Body, the type is set to 'raw' and 'JSON (application/json)'. The body content area is empty. Below the request section, the response section shows 'Status: 204 No Content' and 'Time: 91 ms'. The response body is also empty.

常见的 Web API 约定

开发应用程序的后端服务时，你将想要使用一组一致的约定或策略来处理横切关注点。例如，在上面所示服务中，针对不存在的特定记录的请求会收到 `NotFound` 响应，而不是 `BadRequest` 响应。同样，对于此服务，传递模型绑定类型的命令始终检查 `ModelState.IsValid` 并为无效的模型类型返回 `BadRequest`。

一旦为 Api 指定通用策略，一般可以将其封装在 `Filter(筛选器)`。详细了解 [如何封装 ASP.NET Core MVC 应用程序中的通用 API 策略](#)。

ASP.NET Core 基础知识

2018/5/17 • 9 min to read • [Edit Online](#)

ASP.NET Core 应用程序是在其 `Main` 方法中创建 Web 服务器的控制台应用：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

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

namespace aspnetcoreapp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

`Main` 方法调用 `WebHost.CreateDefaultBuilder`，后者按照生成器模式来创建 Web 应用程序主机。生成器提供定义 Web 服务器（例如，`UseKestrel`）和启动类（`UseStartup`）的方法。在前面的例子中，自动分配了 `Kestrel` Web 服务器。ASP.NET Core 的 Web 主机尝试在 IIS 上运行（如果可用）。对于其他 Web 服务器（如 `HTTP.sys`），可通过调用相应的扩展方法来使用。在下一节对 `UseStartup` 进行了更深入的介绍。

`IWebHostBuilder` 是 `WebHost.CreateDefaultBuilder` 调用的返回类型，它提供了许多可选方法。其中的一些方法包括用于在 `HTTP.sys` 中托管应用的 `UseHttpSys`，以及用于指定根内容目录的 `UseContentRoot`。`Build` 和 `Run` 方法生成 `IWebHost` 对象，该对象托管应用并开始侦听 HTTP 请求。

启动

`WebHostBuilder` 上的 `UseStartup` 方法为你的应用指定 `Startup` 类：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

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

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

`Startup` 类用于定义请求处理管道和配置应用所需的任何服务。`Startup` 必须是公共类，并包含以下方法：

```
public class Startup
{
    // This method gets called by the runtime. Use this method
    // to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
    }

    // This method gets called by the runtime. Use this method
    // to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
    }
}
```

`ConfigureServices` 定义应用所使用的[服务](#)(如 ASP.NET Core MVC、Entity Framework Core 和标识)。`Configure` 定义请求管道的[中间件](#)。

有关详细信息，请参阅[应用程序启动](#)。

内容根

内容根是应用所使用的任何内容的基路径，如视图、[Razor 页面](#) 和静态资产。默认情况下，内容根与用于托管应用的可执行文件的应用程序基路径相同。

Web 根

应用的 Web 根是项目中的目录，其中包含公共资源、CSS 等静态资源、JavaScript 和图形文件。

依赖关系注入(服务)

服务是应用中常用的组件。可以通过[依存关系注入\(DI\)](#)来获取服务。ASP.NET Core 包括默认支持[构造函数注入](#)的本机控制反转(IoC)容器。可根据需要替换默认本机容器。DI 除了具备松散耦合优势以外，还可以使服务(例如，[日志记录](#))在整个应用中可用。

有关详细信息，请参阅[依存关系注入](#)。

中间件

在 ASP.NET Core 中，使用[中间件](#)来撰写请求管道。ASP.NET Core 中间件在 `HttpContext` 上执行异步逻辑，然后调用序列中的下一个中间件或直接终止请求。通过在 `Configure` 方法中调用 `UseXYZ` 扩展方法来添加名为“XYZ”的中间件组件。

ASP.NET Core 包含一组丰富的内置中间件：

- [静态文件](#)
- [路由](#)
- [身份验证](#)
- [响应压缩中间件](#)
- [URL 重写中间件](#)

可以将任何基于 [OWIN](#) 的中间件与 ASP.NET Core 应用结合使用，也可以编写自己的自定义中间件。

有关详细信息，请参阅[中间件和 .NET 的开放 Web 接口\(OWIN\)](#)。

启动 HTTP 请求

有关使用 `IHttpClientFactory` 访问 `HttpClient` 实例以发出 HTTP 请求的信息, 请参阅[启动 HTTP 请求](#)。

环境

环境(如“开发”环境和“生产”环境)是 ASP.NET Core 的高级概念, 可使用环境变量进行设置。

有关详细信息, 请参阅[使用多个环境](#)。

配置

ASP.NET Core 基于名称/值对使用配置模型。配置模型不基于 `System.Configuration` 或 `web.config`。配置从一组有序的配置提供程序获取设置。内置配置提供程序支持各种文件格式(XML、JSON、INI)和环境变量, 从而实现基于环境的配置。也可以编写你自己的自定义配置提供程序。

有关详细信息, 请参阅[配置](#)。

日志记录

ASP.NET Core 支持适用于各种日志记录提供程序的日志记录 API。内置提供程序支持向一个或多个目标发送日志。可使用第三方记录框架。

[日志记录](#)

错误处理

ASP.NET Core 的内置功能可处理应用中的错误, 包括开发人员异常页、自定义错误页、静态状态代码页和启动异常处理。

有关详细信息, 请参阅[如何处理错误](#)。

路由

ASP.NET Core 提供将应用请求路由到路由处理程序的功能。

有关详细信息, 请参阅[路由](#)。

文件提供程序

ASP.NET Core 通过使用文件提供程序抽象化文件系统访问, 文件提供程序可提供一个跨平台处理文件的通用界面。

有关详细信息, 请参阅[文件提供程序](#)。

静态文件

静态文件中间件为静态文件(如 HTML、CSS、映像和 JavaScript)提供服务。

有关详细信息, 请参阅[静态文件](#)。

宿主

ASP.NET Core 应用可配置和启动一个主机, 负责应用启动和生存期管理。

有关详细信息, 请参阅[托管](#)。

会话和应用程序状态

会话状态是 ASP.NET Core 中的一项功能，可用于在用户浏览 Web 应用时保存和存储用户数据。

有关详细信息，请参阅[会话和应用程序状态](#)。

服务器

ASP.NET Core 托管模型不直接侦听请求。托管模型依赖 HTTP 服务器实现将请求转发到应用。转发的请求被打包为一组可通过接口进行访问的功能对象。ASP.NET Core 包含托管的跨平台 Web 服务器，名为 [Kestrel](#)。Kestrel 通常在生产 Web 服务器（如 [IIS](#) 或 [Nginx](#)）后台运行。Kestrel 可作为边缘服务器运行。

有关详细信息，请参阅[服务器](#)和下列主题：

- [Kestrel](#)
- [ASP.NET Core 模块](#)
- [HTTPsys](#)（以前称为 [WebListener](#)）

全球化和本地化

使用 ASP.NET Core 创建多语言网站，可让网站拥有更多受众。ASP.NET Core 提供的服务和中间件可将网站本地化为不同的语言和文化。

有关详细信息，请参阅[全球化和本地化](#)。

请求功能

与 HTTP 请求和响应相关的 Web 服务器实现详细信息在接口中定义。服务器实现和中间件使用这些接口来创建和修改应用的托管管道。

有关详细信息，请参阅[请求功能](#)。

后台任务

后台任务作为托管服务实现。托管服务是一个类，具有实现 [IHostedService](#) 接口的后台任务逻辑。

有关详细信息，请参阅[使用托管服务的后台任务](#)。

.NET 的开放 Web 接口 (OWIN)

ASP.NET Core 支持 .NET 的开放 Web 接口 (OWIN)。OWIN 允许 Web 应用从 Web 服务器分离。

有关详细信息，请参阅[.NET 的开放 Web 接口 \(OWIN\)](#)。

WebSockets

[WebSocket](#) 是一个协议，支持通过 TCP 连接建立持久的双向信道。它可用于聊天、股票报价和游戏等应用，以及 Web 应用中需要实时功能的任何位置。ASP.NET Core 支持 Web 套接字功能。

有关详细信息，请参阅[WebSockets](#)。

Microsoft.AspNetCore.All 元包

ASP.NET Core 的 [Microsoft.AspNetCore.All](#) 元包包括：

- ASP.NET Core 团队支持的所有包。
- Entity Framework Core 支持的所有包。

- ASP.NET Core 和 Entity Framework Core 使用的内部和第三方依赖关系。

有关详细信息, 请参阅 [Microsoft.AspNetCore.All 元包](#)。

.NET Core 与 .NET Framework 运行时

ASP.NET Core 应用可以面向 .NET Core 或 .NET Framework 运行时。

有关详细信息, 请参阅[在 .NET Core 和 .NET Framework 之间进行选择](#)。

在 ASP.NET Core 和 ASP.NET 之间进行选择

有关在 ASP.NET Core 和 ASP.NET 之间进行选择的详细信息, 请参阅[在 ASP.NET Core 和 ASP.NET 之间进行选择](#)。

ASP.NET Core 中的应用程序启动

2018/5/14 • 9 min to read • [Edit Online](#)

作者: [Steve Smith](#)、[Tom Dykstra](#) 和 [Luke Latham](#)

`Startup` 类配置服务和应用的请求管道。

Startup 类

ASP.NET Core 应用使用 `Startup` 类，按照约定命名为 `Startup`。`Startup` 类:

- 可选择性地包括 `ConfigureServices` 方法以配置应用的服务。
- 必须包括 `Configure` 方法以创建应用的请求处理管道。

当应用启动时，运行时调用 `ConfigureServices` 和 `Configure` :

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

通过 `WebHostBuilderExtensions`、`UseStartup<TStartup>` 方法指定 `Startup` 类:

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

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

`Startup` 类构造函数接受由主机定义的依赖关系。在 `Startup` 类中[注入依赖关系](#)的常见用途为注入:

- `IHostingEnvironment` 以按环境配置服务。
- `IConfiguration` 以在启动过程中配置应用。

```

public class Startup
{
    public Startup(IHostingEnvironment env, IConfiguration config)
    {
        HostingEnvironment = env;
        Configuration = config;
    }

    public IHostingEnvironment HostingEnvironment { get; }
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        // Configuration is available during startup. Examples:
        // Configuration["key"]
        // Configuration["subsection:suboption1"]
    }
}

```

注入 `IHostingEnvironment` 的替代方法是使用基于约定的方法。该应用可以为不同的环境(例如 `StartupDevelopment`) 定义单独的 `Startup` 类，并在运行时选择适当的 `startup` 类。优先考虑名称后缀与当前环境相匹配的类。如果应用在开发环境中运行并包含 `Startup` 类和 `StartupDevelopment` 类，则使用 `StartupDevelopment` 类。有关详细信息，请参阅[使用多个环境](#)。

若要详细了解 `WebHostBuilder`，请参阅[承载](#)主题。有关在启动过程中处理错误的信息，请参阅[启动异常处理](#)。

ConfigureServices 方法

`ConfigureServices` 方法是：

- Optional
- 在 `Configure` 方法配置应用服务之前，由 Web 主机调用。
- 其中按常规设置[配置选项](#)。

将服务添加到服务容器，使其在应用和 `Configure` 方法中可用。这些服务通过[依赖关系注入](#)或 `IApplicationBuilder.ApplicationServices` 解析。

Web 主机可能会在调用 `Startup` 方法之前配置某些服务。有关详细信息，请参阅[承载](#)主题。

对于需要大量设置的功能，`IServiceCollection` 上有 `Add[Service]` 扩展方法。典型 Web 应用将为实体框架、标识和 MVC 注册服务：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

ASP.NET Core MVC 的 SetCompatibilityVersion

`SetCompatibilityVersion` 方法允许应用选择加入或退出 ASP.NET MVC Core 2.1+ 中引入的潜在中断行为变更。这些潜在的中断行为变更通常取决于 MVC 子系统的行为方式以及运行时调用“代码”的方式。通过选择加入，你将获取最新的行为以及 ASP.NET Core 的长期行为。

以下代码将兼容模式设置为 ASP.NET Core 2.1：

```
[!code-csharp>Main]
```

建议使用最新版本 (`CompatibilityVersion.Version_2_1`) 来测试应用程序。我们预计大多数应用程序不会使用最新版本进行中断行为变更。

调用 `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)` 的应用程序会被阻止进行 ASP.NET Core 2.1 MVC 和更高的 2.x 版本中引入的潜在中断行为变更。该阻止操作：

- 不适用于所有 2.1 和更高版本的更改，它的目标是潜在地中断 MVC 子系统中的 ASP.NET Core 运行时行为变更。
- 不会扩展到下一个主要版本。

不调用 `SetCompatibilityVersion` 的 ASP.NET Core 2.1 和更高 2.x 版本的应用程序的默认兼容性是 2.0 兼容性。即，未调用 `SetCompatibilityVersion` 与调用 `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)` 相同。

以下代码将兼容模式设置为 ASP.NET Core 2.1(以下行为除外)：

- `AllowCombiningAuthorizeFilters`
- `InputFormatterExceptionPolicy`

```
[!code-csharp>Main]
```

对于遇到中断行为变更的应用，请使用适当的兼容性开关：

- 允许使用最新版本并选择退出特定的中断行为变更。
- 请用些时间更新应用，以便其适用于最新更改。

`MvcOptions` 类源注释充分地阐述了更改的内容以及为什么更改对大多数用户来说是一种改进。

将来会推出 [ASP.NET Core 3.0 版本](#)。在 3.0 版本中，将删除兼容性开关支持的旧行为。我们认为这些积极的变化几乎使所有用户受益。现在通过引入这些更改，大多数应用可以立即受益，其他人员将有时间更新其应用程序。

Startup 中可用的服务

Web 主机提供 `Startup` 类构造函数可用的某些服务。应用通过 `ConfigureServices` 添加其他服务。然后，主机和应用服务都可以在 `Configure` 和整个应用程序中使用。

Configure 方法

`Configure` 方法用于指定应用响应 HTTP 请求的方式。可通过将中间件组件添加到 `IApplicationBuilder` 实例来配置请求管道。`Configure` 方法可使用 `IApplicationBuilder`，但未在服务容器中注册。承载创建 `IApplicationBuilder` 并将其直接传递给 `Configure`（[引用源](#)）。

[ASP.NET Core 模板](#) 配置支持开发人员异常页、[BrowserLink](#)、错误页、静态文件和 ASP.NET MVC 的管道：

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

    app.UseStaticFiles();

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

每个 `Use` 扩展方法将中间件组件添加到请求管道。例如，`UseMvc` 扩展方法将[路由中间件](#)添加到请求管道，并将 [MVC](#) 配置为默认处理程序。

请求管道中的每个中间件组件负责调用管道中的下一个组件，或在适当情况下使链发生短路。如果中间件链中未发生短路，则每个中间件都有第二次机会在将请求发送到客户端前处理该请求。

其他服务（如 `IHostingEnvironment` 和 `ILoggerFactory`），也可以在方法签名中指定。如果指定，其他服务如果可用，将被注入。

有关如何使用 `IApplicationBuilder` 和中间件处理顺序的详细信息，请参阅[中间件](#)。

便利方法

可使用 `ConfigureServices` 和 `Configure` 便利方法，而不是指定 `Startup` 类。多次调用 `ConfigureServices` 将追加到另一个。多次调用 `Configure` 将使用上一个方法调用。

```

public class Program
{
    public static IHostingEnvironment HostingEnvironment { get; set; }
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                HostingEnvironment = hostingContext.HostingEnvironment;
                Configuration = config.Build();
            })
            .ConfigureServices(services =>
            {
                services.AddMvc();
            })
            .Configure(app =>
            {
                if (HostingEnvironment.IsDevelopment())
                {
                    app.UseDeveloperExceptionPage();
                }
                else
                {
                    app.UseExceptionHandler("/Error");
                }

                // Configuration is available during startup. Examples:
                // Configuration["key"]
                // Configuration["subsection:suboption1"]

                app.UseMvcWithDefaultRoute();
                app.UseStaticFiles();
            })
            .Build();
    }
}

```

Startup 筛选器

在应用的 [Configure](#) 中间件管道的开头或末尾使用 [IStartupFilter](#) 来配置中间件。[IStartupFilter](#) 有助于确保中间件在应用请求处理管道的开始或结束时由库添加的中间件之前或之后运行。

[IStartupFilter](#) 实现单个方法(即 [Configure](#))，该方法接收并返回 [Action<IAApplicationBuilder>](#)。

[IAApplicationBuilder](#) 定义用于配置应用请求管道的类。有关详细信息，请参阅[使用 IApplicationBuilder 创建中间件管道](#)。

在请求管道中，每个 [IStartupFilter](#) 实现一个或多个中间件。筛选器按照添加到服务容器的顺序调用。筛选器可在将控件传递给下一个筛选器之前或之后添加中间件，从而附加到应用管道的开头或末尾。

[示例应用\(如何下载\)](#)演示如何向 [IStartupFilter](#) 注册中间件。示例应用包含一个中间件，该中间件从查询字符串参数中设置选项值：

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;
    private IOptions<AppOptions> _injectedOptions;

    public RequestSetOptionsMiddleware(
        RequestDelegate next, IOptions<AppOptions> injectedOptions)
    {
        _next = next;
        _injectedOptions = injectedOptions;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        Console.WriteLine("RequestSetOptionsMiddleware.Invoke");

        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrWhiteSpace(option))
        {
            _injectedOptions.Value.Option = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

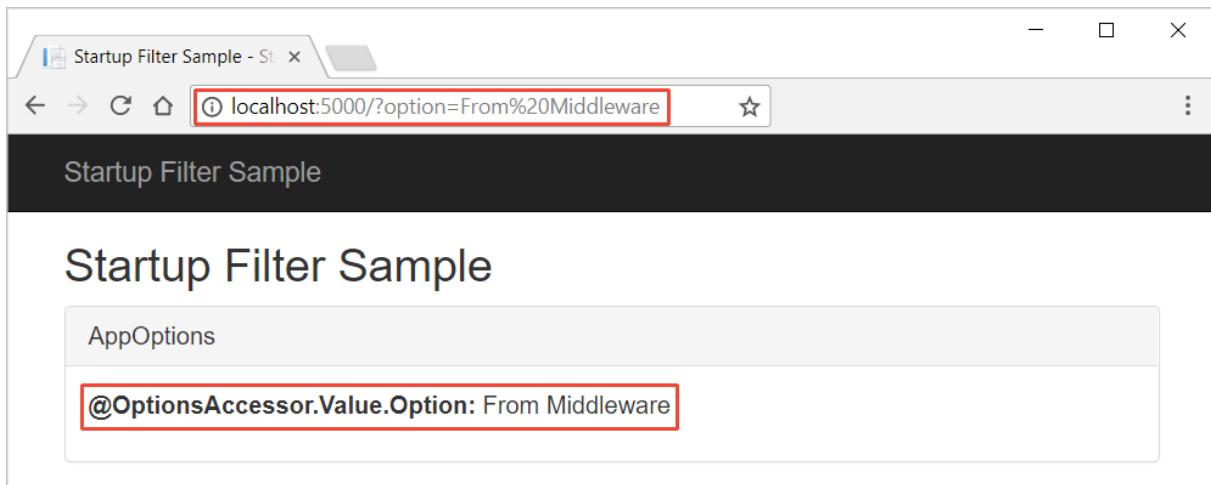
在 `RequestSetOptionsStartupFilter` 类中配置 `RequestSetOptionsMiddleware`：

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

在 `ConfigureServices` 的服务容器中注册 `IStartupFilter`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IStartupFilter, RequestSetOptionsStartupFilter>();
    services.AddMvc();
}
```

当提供 `option` 的查询字符串参数时，中间件在 MVC 中间件呈现响应之前处理分配值：



中间件执行顺序由 `IStartupFilter` 注册顺序设置：

- 多个 `IStartupFilter` 实现可能与相同的对象进行交互。如果顺序很重要，请将它们的 `IStartupFilter` 服务注册进行排序，以匹配其中间件应有的运行顺序。
- 库可能添加包含一个或多个 `IStartupFilter` 实现的中间件，这些实现在向 `IStartupFilter` 注册的其他应用中间件之前或之后运行。若要在库的 `IStartupFilter` 添加中间件之前调用 `IStartupFilter` 中间件，请在将库添加到服务容器之前定位服务注册。若要在此后调用，请在添加库之后定位服务注册。

其他资源

- [承载](#)
- [使用多个环境](#)
- [中间件](#)
- [日志记录](#)
- [配置](#)
- [StartupLoader 类:FindStartupType 方法\(引用源\)](#)

在 ASP.NET Core 依赖注入

2018/5/14 • 20 min to read • [Edit Online](#)

作者: [Steve Smith](#) 和 [Scott Addie](#)

ASP.NET Core 的设计从头至尾以支持和利用依赖注入为目标。ASP.NET Core 应用程序可以通过将内置框架服务注入 Startup 类的方法中来对其进行利用，应用程序服务也可以进行注入配置。由 ASP.NET Core 提供的默认服务容器提供了一个最小功能集合，并非用于替换其他容器。

[查看或下载示例代码\(如何下载\)](#)

什么是依赖注入？

依赖注入 (DI) 是一种用于在对象与其协作者或依赖项之间实现松散耦合的技术。该技术不是直接实例化协作者或使用静态引用，而是以某种方式向类提供该类执行其操作所需的对象。大多数情况下，类通过其构造函数声明它们的依赖项，从而允许它们遵循[显式依赖关系原则](#)。此方法称为“构造函数注入”。

在遵循 DI 原则来设计类时，由于它们与其协作者没有直接的、硬编码的依赖关系，所以它们之间的耦合更为松散。这遵循[依赖倒置原则](#)，其中指出“高级模块不应依赖于低级模块；同时两者都应依赖于抽象”。类请求在该类被构造时提供给它们的抽象（通常是 `interfaces`），而不是引用特定的实现。将依赖关系提取到接口中并将这些接口的实现作为参数提供也是[策略设计模式](#)的一个例子。

当系统被设计为使用 DI 时，许多类通过其构造函数（或属性）来请求它们的依赖项，所以有一个专门用来创建这些类以及它们相关的依赖项的类是很有帮助的。这些类统称为容器，或更具体地说，[控制反转 \(IoC\)](#) 容器或依赖注入 (DI) 容器。容器本质上是一个工厂，它负责提供从中请求的类型的实例。如果给定的类型已声明它具有依赖项，并且容器已经被配置为提供这些依赖类型，那么它将在创建请求实例时创建依赖项。通过这种方式，可以为类提供复杂的依赖关系图，而无需任何硬编码对象构造。除了创建包含依赖项的对象外，容器通常还可以管理应用程序中的对象生命周期。

ASP.NET Core 包含一个简单的内置容器（表示为 `IServiceProvider` 接口），默认情况下，该容器支持构造函数注入，ASP.NET 可通过 DI 提供某些服务。ASP.NET 的容器指其作为 服务 管理的类型。本文其余部分中，[服务](#) 指的是由 ASP.NET Core 的 IoC 容器管理的类型。您可以在应用程序的 `Startup` 类中的 `ConfigureServices` 方法中配置内置容器的服务。

注意

Martin Fowler 写了另一篇全面介绍[控制反转容器和依赖注入模式](#)的文章。Microsoft 模式和实践也对[依赖关系注入](#)进行了很好的描述。

注意

本文介绍了适用于所有 ASP.NET 应用程序的依赖注入。[依赖关系注入和控制器](#) 中介绍了 MVC 控制器内的依赖关系注入。

构造函数注入行为

构造函数注入要求相关构造函数是公共的。否则，应用程序会引发

`InvalidOperationException` 异常：

找不到适合类型 `YourType` 的构造函数。请确保该类型是具体的，并为公共构造函数的所有参数注册服务。

构造函数注入要求只存在一个适用的构造函数。支持构造函数重载，但其参数可以全部通过依赖注入来实现的重载只能存在一个。如果存在多个，应用程序将引发

`InvalidOperationException` 异常：

已在类型 `YourType` 中找到多个接受所有给定的参数类型的构造函数。应该只有一个适用的构造函数。

构造函数可以接受依赖注入没有提供的参数，但这些参数必须支持默认值。例如：

```
// throws InvalidOperationException: Unable to resolve service for type
// 'System.String'...
public CharactersController(ICharacterRepository characterRepository, string
title)
{
    _characterRepository = characterRepository;
    _title = title;
}

// runs without error
public CharactersController(ICharacterRepository characterRepository, string
title = "Characters")
{
    _characterRepository = characterRepository;
    _title = title;
}
```

使用框架提供的服务

`Startup` 类中的 `ConfigureServices` 方法负责定义应用程序将使用的服务，包括 Entity Framework Core 和 ASP.NET Core MVC 之类的平台功能。最初，提供给 `ConfigureServices` 的 `IServiceCollection` 定义了以下服务（具体取决于[配置主机的方式](#)）：

服务类型	生存期
<code>Microsoft.AspNetCore.Hosting.IHostingEnvironment</code>	单一实例
<code>Microsoft.Extensions.Logging.ILoggerFactory</code>	单一实例
<code>Microsoft.Extensions.Logging.ILogger<T></code>	单一实例
<code>Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory</code>	暂时
<code>Microsoft.AspNetCore.Http.IHttpContextFactory</code>	暂时

服务类型	生存期
<code>Microsoft.Extensions.Options.IOptions<T></code>	单一实例
<code>System.Diagnostics.DiagnosticSource</code>	单一实例
<code>System.Diagnostics.DiagnosticListener</code>	单一实例
<code>Microsoft.AspNetCore.Hosting.IStartupFilter</code>	暂时
<code>Microsoft.Extensions.ObjectPool.ObjectPoolProvider</code>	单一实例
<code>Microsoft.Extensions.Options.IConfigureOptions<T></code>	暂时
<code>Microsoft.AspNetCore.Hosting.Server.IServer</code>	单一实例
<code>Microsoft.AspNetCore.Hosting.IStartup</code>	单一实例
<code>Microsoft.AspNetCore.Hosting.IApplicationLifetime</code>	单一实例

下面的示例介绍了如何使用多个扩展方法(如 `AddDbContext`、`AddIdentity` 和 `AddMvc`)将附加服务添加到容器中。

```
// This method gets called by the runtime. Use this method to add services to
// the container.
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity< ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient< IEmailSender, AuthMessageSender>();
    services.AddTransient< ISmsSender, AuthMessageSender>();
}
```

ASP.NET 提供的功能和中间件(如 MVC)，遵循使用单个 `AddServiceName` 扩展方法注册该功能请求的所有服务的约定。

提示

你可以通过其参数列表在 `Startup` 方法内请求某些框架提供的服务，请参阅[应用程序启动](#)了解详细信息。

注册服务

可以按如下方式注册自己的应用程序服务。第一个泛型类型表示将从容器请求的类型(通常是一个接口)。第二个泛型类型表示将由容器实例化并用于满足此类请求的具体类型。

```
services.AddTransient<IEmailSender, AuthMessageSender>();  
services.AddTransient<ISmsSender, AuthMessageSender>();
```

注意

每个 `services.Add<ServiceName>` 扩展方法添加(并可能配置)服务。例如,
`services.AddMvc()` 添加 MVC 需要的服务。建议遵循此约定, 将扩展方法放置在
`Microsoft.Extensions.DependencyInjection` 命名空间中, 以封装服务注册的组。

`AddTransient` 方法用于将抽象类型映射到为每个需要它的对象分别实例化的具体服务。这称为服务的 [生存期](#), 其他生存期选项如下所述。为注册的每个服务选择适当的生存期非常重要。是否应该向每个请求它的类提供一个新的服务实例? 是否在整个给定的 Web 请求中使用一个实例? 或是否应该在应用程序生存期内使用单例?

在本文示例中, 有一个显示字符名称的简单控制器, 名为 `CharactersController`。其 `Index` 方法会显示存储在应用程序中的当前字符列表, 如果不存在, 则使用少量字符初始化该集合。请注意, 尽管此应用程序为其持久化使用 Entity Framework Core 和 `ApplicationDbContext`, 但在控制器中没有任何明显的表现。相反, 特定的数据访问机制已经被抽象为一个接口, 即 `ICharacterRepository`, 它遵循 [仓储模式](#)。通过构造函数请求 `ICharacterRepository` 实例, 并将其分配给私有字段, 然后根据需要使用它访问字符。

```
public class CharactersController : Controller
{
    private readonly ICharacterRepository _characterRepository;

    public CharactersController(ICharacterRepository characterRepository)
    {
        _characterRepository = characterRepository;
    }

    // GET: /characters/
    public IActionResult Index()
    {
        PopulateCharactersIfNoneExist();
        var characters = _characterRepository.ListAll();

        return View(characters);
    }

    private void PopulateCharactersIfNoneExist()
    {
        if (!_characterRepository.ListAll().Any())
        {
            _characterRepository.Add(new Character("Darth Maul"));
            _characterRepository.Add(new Character("Darth Vader"));
            _characterRepository.Add(new Character("Yoda"));
            _characterRepository.Add(new Character("Mace Windu"));
        }
    }
}
```

ICharacterRepository 定义了控制器使用 Character 实例时需要的两种方法。

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

namespace DependencyInjectionSample.Interfaces
{
    public interface ICharacterRepository
    {
        IEnumerable<Character> ListAll();
        void Add(Character character);
    }
}
```

此接口又由一个具体类型实现，即在运行时使用的 CharacterRepository 。

注意

DI 与 CharacterRepository 类一起使用的方式是一个通用模型，您可以为所有应用程序服务遵循此模型，而不仅仅在“仓储库”或数据访问类中。

```

using System.Collections.Generic;
using System.Linq;
using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Models
{
    public class CharacterRepository : ICharacterRepository
    {
        private readonly ApplicationDbContext _dbContext;

        public CharacterRepository(ApplicationDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public IEnumerable<Character> ListAll()
        {
            return _dbContext.Characters.AsEnumerable();
        }

        public void Add(Character character)
        {
            _dbContext.Characters.Add(character);
            _dbContext.SaveChanges();
        }
    }
}

```

请注意, `CharacterRepository` 在其构造函数中请求一个 `ApplicationDbContext`。依赖注入以这种链式方式被使用并不罕见, (每个被请求的依赖进而请求它自己的依赖)。容器将负责解决图中所有的依赖关系, 并返回完全解析后的服务。

注意

创建请求的对象, 和它需要的所有对象, 以及这些对象需要的所有对象, 这有时被称为 **对象图**。同样, 必须被解析的依赖关系的集合通常被称为**依赖关系树**或**依赖项关系图**。

在这种情况下, `Startup` 和 `ApplicationDbContext` 必须在 `ICharacterRepository` 类的 `ConfigureServices` 中的服务容器中注册。`ApplicationDbContext` 通过调用扩展方法 `AddDbContext<T>` 进行配置。下面的代码演示了 `CharacterRepository` 类型的注册。

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase()
    );

    // Add framework services.
    services.AddMvc();

    // Register application services.
    services.AddScoped<ICharacterRepository, CharacterRepository>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new
        Operation(Guid.Empty));
    services.AddTransient<OperationService, OperationService>();
}

```

应使用 `Scoped` 生命周期将实体框架上下文添加到服务容器中。如果您使用上面所示的帮助方法，则会自动执行此操作。使用实体框架的仓储库应使用相同的生命周期。

警告

需要注意的主要危险是从单例解析 `Scoped` 服务。在此类情况下，当处理后续请求时，服务可能会处于不正确的状态。

具有依赖关系的服务应在容器中对它们进行注册。如果服务构造函数需要一个基元（如 `string`），可使用[配置](#)和[选项模式](#)将其注入。

服务生存期和注册选项

可使用以下生存期配置 ASP.NET 服务：

暂时

暂时生存期服务是每次请求时创建的。这种生存期适合轻量级、无状态的服务。

作用域 (`Scoped`)

作用域生存期服务以每个请求一次的方式创建。

警告

如果在中间件内使用有作用域的服务，请将该服务注入至 `Invoke` 或 `InvokeAsync` 方法。请不要通过构造函数注入进行注入，因为它会强制服务的行为与单一实例类似。

单例

单例生存期服务在第一次被请求时或者 `ConfigureServices` 运行时（如果在其中指定实例）创建，然后每个后续请求使用同一实例。如果应用程序需要单例行为，建议允许服务容器管理服务的生存期，而不是实现单例设计模式并在类中自行管理对象的生存期。

服务可以使用多种方式注册到容器中。我们已经看到了如何通过指定要使用的具体类型来注册一个给定类型的服务实现。此外，可以指定一个工厂，然后将其用于按需创建实例。第三种方法是直接指定要使用的类型的实例，在这种情况下，容器将永远不会尝试创建实例（也不会释放实例）。

为了演示这些生存期和注册选项之间的差异，请考虑一个简单的接口，将一个或多个任务表示为具有唯一标识符 `OperationId` 的操作。根据此服务的生存期配置方式，容器将向请求的类提供相同或不同的实例。为了明确正在请求哪个生存期，我们将为每个生命周期选项创建一个类型：

```
using System;

namespace DependencyInjectionSample.Interfaces
{
    public interface IOperation
    {
        Guid OperationId { get; }
    }

    public interface IOperationTransient : IOperation
    {
    }

    public interface IOperationScoped : IOperation
    {
    }

    public interface IOperationSingleton : IOperation
    {
    }

    public interface IOperationSingletonInstance : IOperation
    {
    }
}
```

我们使用单个类 `Operation` 来实现这些接口，它在构造函数中接受一个 `Guid`，如果没有提供，则使用一个新的 `Guid`。

接下来，在 `ConfigureServices` 中，根据其命名的生存期，将每个类型添加到容器中：

```
services.AddScoped<ICharacterRepository, CharacterRepository>();
services.AddTransient<IOperationTransient, Operation>();
services.AddScoped<IOperationScoped, Operation>();
services.AddSingleton<IOperationSingleton, Operation>();
services.AddSingleton<IOperationSingletonInstance>(new
    Operation(Guid.Empty));
services.AddTransient<OperationService, OperationService>();
}
```

请注意，`IOperationSingletonInstance` 服务正在使用一个具有已知 ID `Guid.Empty` 的特定实例，因此可以明确何时在使用此类型（其 `Guid` 将全部为零）。我们已注册了依赖于每个其他 `Operation` 类型的 `OperationService`，因此对每个 `Operation` 类型来说，可以在请求中明确该服务是会获得与控制器相同的实例，还是获得一个新实例。此服务的全部作用就是将其依赖项作为属性公开，以便它们可以显示在视图中。

```
using DependencyInjectionSample.Interfaces;

namespace DependencyInjectionSample.Services
{
    public class OperationService
    {
        public IOperationTransient TransientOperation { get; }
        public IOperationScoped ScopedOperation { get; }
        public IOperationSingleton SingletonOperation { get; }
        public IOperationSingletonInstance SingletonInstanceOperation { get; }

        public OperationService(IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance instanceOperation)
        {
            TransientOperation = transientOperation;
            ScopedOperation = scopedOperation;
            SingletonOperation = singletonOperation;
            SingletonInstanceOperation = instanceOperation;
        }
    }
}
```

为了演示对应用程序的各个独立请求之内和之间的对象生存期，该示例包含一个 `OperationsController`，它请求每种 `IOperation` 类型以及一个 `operationService`。
`Index` 操作随后显示所有控制器和服务的 `operationId` 值。

```

using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;
using Microsoft.AspNetCore.Mvc;

namespace DependencyInjectionSample.Controllers
{
    public class OperationsController : Controller
    {
        private readonly OperationService _operationService;
        private readonly IOperationTransient _transientOperation;
        private readonly IOperationScoped _scopedOperation;
        private readonly IOperationSingleton _singletonOperation;
        private readonly IOperationSingletonInstance
            _singletonInstanceOperation;

        public OperationsController(OperationService operationService,
            IOperationTransient transientOperation,
            IOperationScoped scopedOperation,
            IOperationSingleton singletonOperation,
            IOperationSingletonInstance singletonInstanceOperation)
        {
            _operationService = operationService;
            _transientOperation = transientOperation;
            _scopedOperation = scopedOperation;
            _singletonOperation = singletonOperation;
            _singletonInstanceOperation = singletonInstanceOperation;
        }

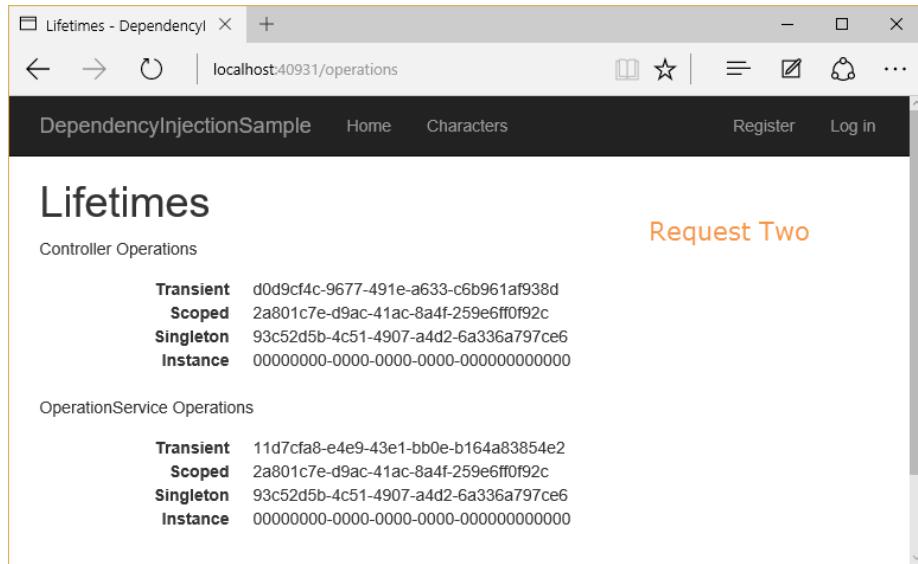
        public IActionResult Index()
        {
            // ViewBag contains controller-requested services
            ViewBag.Transient = _transientOperation;
            ViewBag.Scoped = _scopedOperation;
            ViewBag.Singleton = _singletonOperation;
            ViewBag.SingletonInstance = _singletonInstanceOperation;

            // Operation service has its own requested services
            ViewBag.Service = _operationService;
            return View();
        }
    }
}

```

现在对该控制器操作发起了两个单独的请求：

Lifetime Category	Transient	Scoped	Singleton	Instance
Transient	e6fee2c8-2122-4d10-aa05-cb376042e2c7	661bff78-5ecb-4758-ae43-ec22a3e0babe	93c52d5b-4c51-4907-a4d2-6a336a797ce6	00000000-0000-0000-0000-000000000000
Scoped	a379336b-3fd0-49ac-b176-bae7c27c5de5	661bff78-5ecb-4758-ae43-ec22a3e0babe	93c52d5b-4c51-4907-a4d2-6a336a797ce6	00000000-0000-0000-0000-000000000000
Singleton	93c52d5b-4c51-4907-a4d2-6a336a797ce6	93c52d5b-4c51-4907-a4d2-6a336a797ce6	93c52d5b-4c51-4907-a4d2-6a336a797ce6	00000000-0000-0000-0000-000000000000
Instance	00000000-0000-0000-0000-000000000000	00000000-0000-0000-0000-000000000000	00000000-0000-0000-0000-000000000000	00000000-0000-0000-0000-000000000000



观察请求内和请求之间哪些 `OperationId` 值不同。

- 暂时对象始终不同；向每个控制器和每项服务提供一个新的实例。
- 限定对象在请求内是相同的，但在不同的请求中是不同的
- 单例对象对每个对象和每个请求都是相同的（不管 `ConfigureServices` 是否提供了一个实例）

解析应用程序作用域中有作用域的服务

采用 `IServiceScopeFactory.CreateScope` 创建 `IServiceScope`，以解析应用程序作用域中有作用域的服务。此方法可以用于在启动时访问有作用域的服务以便运行初始化任务。以下示例演示如何在 `Program.Main` 中获取 `MyScopedService` 的上下文：

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var serviceScope = host.Services.CreateScope())
    {
        var services = serviceScope.ServiceProvider;

        try
        {
            var serviceContext = services.GetRequiredService<MyScopedService>();
            // Use the context here
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred.");
        }
    }

    host.Run();
}
```

作用域验证

如果在 ASP.NET Core 2.0 或更高版本上的开发环境中运行应用，默认的服务提供程序会执行检查，从而确认以下内容：

- 没有从根服务提供程序直接或间接解析到有作用域的服务。
- 未将有作用域的服务直接或间接注入到单一实例。

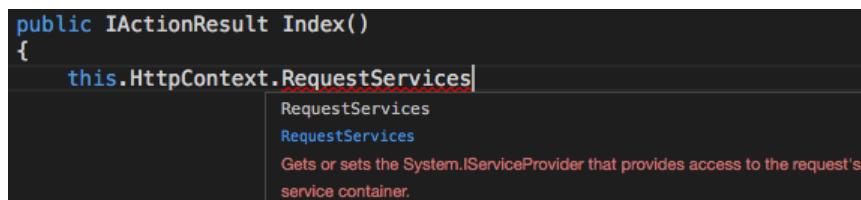
调用 `BuildServiceProvider` 时，会创建根服务提供程序。在启动提供程序和应用时，根服务提供程序的生存期对应于应用/服务的生存期，并在关闭应用时释放。

有作用域的服务由创建它们的容器释放。如果作用域创建于根容器，则该服务的生存会有效地提升至单一实例，因为根容器只会在应用/服务关闭时将其释放。验证服务作用域，将在调用 `BuildServiceProvider` 时收集这类情况。

有关详细信息，请参阅[托管主题中的作用域验证](#)。

请求服务

在 `HttpContext` 中的 ASP.NET 请求内可用的服务通过 `RequestServices` 集合进行公开。



请求服务表示你作为应用程序的一部分配置的服务和请求。当对象指定依赖关系时，`RequestServices`（而不是 `ApplicationServices`）中的类型将满足这些要求。

通常情况下，不应直接使用这些属性，而应该通过类的构造函数来请求所需类的类型，并让框架注入这些依赖关系。这将生成更易于测试且耦合更松散的类（请参阅[测试和调试](#)）。

注意

与访问 `RequestServices` 集合相比，以构造函数参数的形式请求依赖项是更优先的选择。

设计能够进行依赖注入的服务

你应将自己的服务设计为可以使用依赖注入来获取其协作对象。这意味着避免在服务中使用有状态的静态方法调用（这会导致称为[静态粘贴](#)的代码异味）和依赖类的直接实例化。在选择是实例化一个类型还是通过依赖注入来请求它时，记住这个短语可能会有所帮助：[新增即粘附](#)。遵循[面向对象设计的SOLID原则](#)，你往往会觉得小型、良好分解和易于测试的类。

如果发现你的类往往具有太多的依赖项需要注入怎么办？这通常表明你的类正试图做太多的事情，而且可能违反了 [SRP-单一责任原则](#)。看看是否可以通过将某些职责移动到一个新类来重构类。请记住，你的 `Controller` 类应关注用户界面问题，因此业务规则和数据访问实现细节应保留在适用于这些[分离的关注点](#)的类中。

具体就数据访问而言，可以将 `DbContext` 注入到你的控制器（假设你已向 `ConfigureServices` 中的服务容器中添加了 EF）。一些开发人员更愿意使用数据库的存储库接口而不是直接注入 `DbContext`。使用接口将数据访问逻辑封装于一处可以最大程度减少数据更改时需要更改的位置数量。

释放服务

容器将为其创建的 `IDisposable` 类型调用 `Dispose`。但是，如果您自己将一个实例

添加到容器，它将不会被释放。

示例：

```
// Services implement IDisposable:  
public class Service1 : IDisposable {}  
public class Service2 : IDisposable {}  
public class Service3 : IDisposable {}  
  
public interface ISomeService {}  
public class SomeServiceImplementation : ISomeService, IDisposable {}  
  
public void ConfigureServices(IServiceCollection services)  
{  
    // container will create the instance(s) of these types and will dispose  
    them  
    services.AddScoped<Service1>();  
    services.AddSingleton<Service2>();  
    services.AddSingleton<ISomeService>(sp => new SomeServiceImplementation());  
  
    // container didn't create instance so it will NOT dispose it  
    services.AddSingleton<Service3>(new Service3());  
    services.AddSingleton(new Service3());  
}
```

注意

在 1.0 版中，容器将对所有 `IDisposable` 对象调用 `dispose`，包括那些并非由它创建的对象。

替换默认服务容器

内置服务容器专门服务框架和基于其上生成的大多数应用程序的基本需求。但是，开发人员可以使用自己喜欢的容器替换内置容器。`ConfigureServices` 方法通常返回 `void`，但是，如果更改其签名以返回 `IServiceProvider`，则可以配置和返回不同的容器。有许多可用于 .NET 的 IOC 容器。在此示例中，将使用 [Autofac](#) 包。

首先，安装适当的容器包：

- `Autofac`
- `Autofac.Extensions.DependencyInjection`

接下来，在 `ConfigureServices` 中配置容器并返回 `IServiceProvider`：

```
public IServiceProvider ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc();  
    // Add other framework services  
  
    // Add Autofac  
    var containerBuilder = new ContainerBuilder();  
    containerBuilder.RegisterModule<DefaultModule>();  
    containerBuilder.Populate(services);  
    var container = containerBuilder.Build();  
    return new AutofacServiceProvider(container);  
}
```

注意

使用第三方 DI 容器时，必须更改 `ConfigureServices`，以便其返回 `IServiceProvider`，而不是 `void`。

最后，将 Autofac 在 `DefaultModule` 中配置为正常：

```
public class DefaultModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<CharacterRepository>().As<ICharacterRepository>();
    }
}
```

在运行时，将使用 Autofac 来解析类型，并注入依赖。了解有关使用 [Autofac](#) 和 [ASP.NET Core](#) 的更多信息。

线程安全

单例服务需要是线程安全的。如果单例服务依赖于一个瞬时服务，那么瞬时服务可能也需要是线程安全的，具体取决于单例使用它的方式。

建议

使用依赖注入时，请记住以下建议：

- DI 适用于具有复杂的依赖关系的对象。控制器、服务、适配器和仓储都是可能添加到 DI 中的对象示例。
- 避免在 DI 中直接存储数据和配置。例如，用户的购物车通常不应添加到服务容器中。配置应使用 [选项模型](#)。同样，避免“数据持有者”对象，也就是仅为实现对某些其他对象的访问而存在的对象。如果可能，最好通过 DI 请求所需的实际项目。
- 避免静态访问服务。
- 应用程序代码中避免服务位置。
- 避免静态访问 `HttpContext`。

注意

如所有建议组合一样，你可能会遇到必须忽略一个的情况。我们发现例外情况很罕见 - 大多数是框架本身内部非常特殊的情况。

请记住，依赖注入是静态/全局对象访问模式的替代方法。如果将其与静态对象访问混合使用，则无法实现 DI 的优点。

其他资源

- [应用程序启动](#)
- [测试和调试](#)
- [基于工厂的中间件激活](#)
- [在 ASP.NET Core 中使用依赖关系注入编写干净代码 \(MSDN\)](#)

- Container-Managed Application Design, Prelude: Where does the Container Belong?(容器托管的应用程序设计, 序言:容器属于何处?)
- Explicit Dependencies Principle(显式依赖关系原则)
- Inversion of Control Containers and the Dependency Injection Pattern(控制反转容器和依赖关系注入模式)

ASP.NET Core 中间件

2018/5/14 • 11 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Steve Smith](#)

[查看或下载示例代码\(如何下载\)](#)

什么是中间件？

中间件是一种装配到应用程序管道以处理请求和响应的软件。每个组件:

- 选择是否将请求传递到管道中的下一个组件。
- 可在调用管道中的下一个组件前后执行工作。

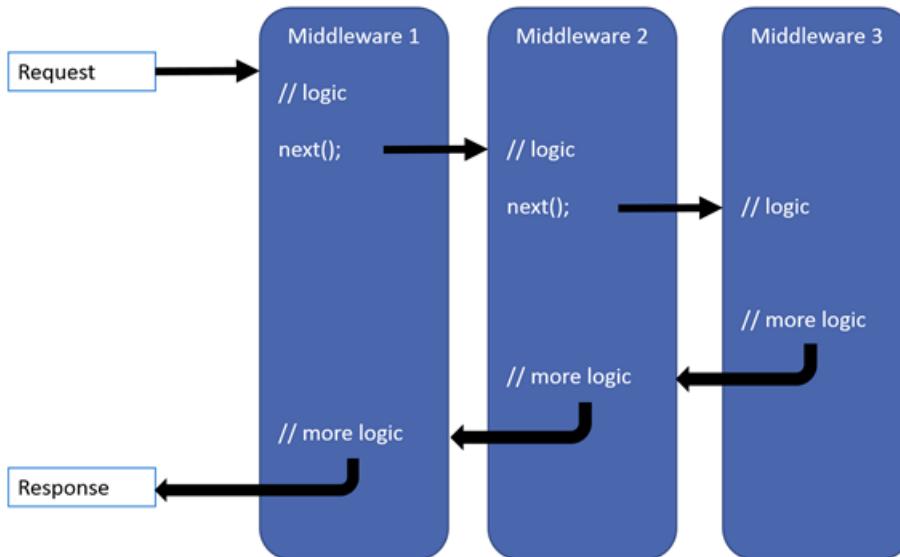
请求委托用于生成请求管道。请求委托处理每个 HTTP 请求。

使用 [Run](#)、[Map](#) 和 [Use](#) 扩展方法来配置请求委托。可将一个单独的请求委托并行指定为匿名方法(称为并行中间件)，或在可重用的类中对其进行定义。这些可重用的类和并行匿名方法即为中间件或中间件组件。请求管道中的每个中间件组件负责调用管道中的下一个组件，或在适当情况下使链发生短路。

[将 HTTP 模块迁移到中间件](#)介绍了 ASP.NET Core 和 ASP.NET 4.x 中请求管道之间的差异，并提供了更多的中间件示例。

使用 [IApplicationBuilder](#) 创建中间件管道

ASP.NET Core 请求管道包含一系列相继调用的请求委托，如下图所示(执行过程遵循黑色箭头):



每个委托均可在下一个委托前后执行操作。此外，委托还可以决定不将请求传递给下一个委托，这就是对请求管道进行短路。通常需要短路，因为这样可以避免不必要的工作。例如，静态文件中间件可以返回静态文件请求并使管道的其余部分短路。需要尽早在管道中调用异常处理委托，以便它们可以捕获在管道的后期阶段所发生的异常。

尽可能简单的 ASP.NET Core 应用设置了处理所有请求的单个请求委托。这种情况不包括实际请求管道。调用单个匿名函数以响应每个 HTTP 请求。

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

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

第一个 `app.Run` 委托终止了管道。

可使用 `app.Use` 将多个请求委托链接在一起。`next` 参数表示管道中的下一个委托。(请记住, 可通过不调用 `next` 参数使管道短路。)通常可在下一个委托前后执行操作, 如以下示例所示:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

警告

在向客户端发送响应后, 请勿调用 `next.Invoke`。响应启动后, 针对 `HttpResponse` 的更改将引发异常。例如, 设置标头、状态代码等更改将引发异常。调用 `next` 后写入响应正文:

- 可能导致违反协议。例如, 写入的长度超过规定的 `content-length`。
- 可能损坏正文格式。例如, 向 CSS 文件中写入 HTML 页脚。

`HttpResponse.HasStarted` 是一个有用的提示, 指示是否已发送标头和/或已写入正文。

中间件排序

向 `Configure` 方法添加中间件组件的顺序定义了针对请求调用这些组件的顺序, 以及响应的相反顺序。此排序对于安全性、性能和功能至关重要。

`Configure` 方法(如下所示)添加以下中间件组件:

- 异常/错误处理
- 静态文件服务器
- 身份验证

4. MVC

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void Configure(IApplicationBuilder app)
{
    app.UseExceptionHandler("/Home/Error"); // Call first to catch exceptions
                                                // thrown in the following middleware.

    app.UseStaticFiles();                      // Return static files and end pipeline.

    app.UseAuthentication();                   // Authenticate before you access
                                                // secure resources.

    app.UseMvcWithDefaultRoute();             // Add MVC to the request pipeline.
}
```

在以上代码中，`UseExceptionHandler` 是添加到管道的第一个中间件组件，因此，该组件可捕获在后面的调用中发生的任何异常。

因为尽早在管道中调用静态文件中间件，因此该组件可处理请求并使引致短路，而无需通过剩余组件。静态文件中间件不提供授权检查。可公开访问由静态文件中间件服务的任何文件，包括 `wwwroot` 下的文件。请参阅[静态文件](#)，了解如何保护静态文件。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果静态文件中间件未处理请求，则请求将被传递给执行身份验证的标识中间件（`app.UseAuthentication`）。标识不使未经身份验证的请求短路。虽然标识对请求进行身份验证，但仅在 MVC 选择特定 Razor 页或控制器和操作后，才发生授权（和拒绝）。

以下示例演示中间件排序，其中静态文件的请求在响应压缩中间件前由静态文件中间件进行处理。静态文件未通过此中间件排序进行压缩。可压缩来自 `UseMvcWithDefaultRoute` 的 MVC 响应。

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();          // Static files not compressed
                                  // by middleware.

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

Use、Run 和 Map

使用 `Use`、`Run` 和 `Map` 配置 HTTP 管道。`Use` 方法可使管道短路（即不调用 `next` 请求委托）。`Run` 是一种约定，并且某些中间件组件可公开在管道末尾运行的 `Run[Middleware]` 方法。

`Map*` 扩展用作约定来创建管道分支。`Map` 基于给定请求路径的匹配项来创建请求管道分支。如果请求路径以给定路径开头，则执行分支。

```

public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}

```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应：

请求	响应
localhost:1234	来自非 Map 委托的 Hello。
localhost:1234/map1	Map 测试 1
localhost:1234/map2	Map 测试 2
localhost:1234/map3	来自非 Map 委托的 Hello。

使用 `Map` 时，将从 `HttpRequest.Path` 中删除匹配的线段，并针对每个请求将该线段追加到 `HttpRequest.PathBase`。

`MapWhen` 基于给定谓词的结果创建请求管道分支。`Func<HttpContext, bool>` 类型的任何谓词均可用于将请求映射到管道的新分支。在以下示例中，谓词用于检测查询字符串变量 `branch` 是否存在：

```

public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                    HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}

```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应：

请求	响应
localhost:1234	来自非 Map 委托的 Hello。
localhost:1234/?branch=master	使用的分支 = 主要分支

`Map` 支持嵌套，例如：

```

app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a"
        //...
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b"
        //...
    });
});

```

此外，`Map` 还可同时匹配多个段，例如：

```
app.Map("/level1/level2", HandleMultiSeg);
```

内置中间件

ASP.NET Core 附带以下中间件组件，以及用于添加这些组件的顺序的说明：

中间件	描述	顺序
身份验证	提供身份验证支持。	在需要 <code>HttpContext.User</code> 之前。 OAuth 回叫的终端。

中间件	描述	顺序
CORS	配置跨域资源共享。	在使用 CORS 的组件之前。
诊断	配置诊断。	在生成错误的组件之前。
ForwardedHeaders/HttpOverrides	将代理标头转发到当前请求。	在使用更新的字段(示例:Scheme、Host、ClientIP、Method)的组件之前。
响应缓存	提供对缓存响应的支持。	在需要缓存的组件之前。
响应压缩	提供对压缩响应的支持。	在需要压缩的组件之前。
RequestLocalization	提供本地化支持。	在对本地化敏感的组件之前。
路由	定义和约束请求路由。	用于匹配路由的终端。
会话	提供对管理用户会话的支持。	在需要会话的组件之前。
静态文件	为提供静态文件和目录浏览提供支持。	如果请求与文件匹配, 则为终端。
URL 重写	提供对重写 URL 和重定向请求的支持。	在使用 URL 的组件之前。
WebSockets	启用 WebSockets 协议。	在接受 WebSocket 请求所需的组件之前。

写入中间件

通常, 中间件封装在类中, 并且通过扩展方法公开。请考虑以下中间件, 该中间件通过查询字符串设置当前请求的区域性:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use((context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return next();
        });

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

注意：以上示例代码用于演示创建中间件组件。有关 ASP.NET Core 的内置本地化支持，请参阅[全球化和本地化](#)。

可通过传入区域性（如 `http://localhost:7997/?culture=no`）测试中间件。

以下代码将中间件委托移动到类：

```
using Microsoft.AspNetCore.Http;
using System.Globalization;
using System.Threading.Tasks;

namespace Culture
{
    public class RequestCultureMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestCultureMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public Task InvokeAsync(HttpContext context)
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return this._next(context);
        }
    }
}
```

注意

在 ASP.NET Core 1.x 中，中间件 `Task` 方法的名称必须是 `Invoke`。在 ASP.NET Core 2.0 或更高版本中，该名称可以为 `Invoke` 或 `InvokeAsync`。

以下扩展方法通过 `IApplicationBuilder` 公开中间件：

```
using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}
```

以下代码通过 `Configure` 调用中间件：

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

中间件应通过在其构造函数中公开其依赖项来遵循[显式依赖项原则](#)。在每个应用程序生存期构造一次中间件。如果需要与请求中的中间件共享服务，请参阅下面讲述的按请求依赖项。

中间件组件可通过构造函数参数从依赖关系注入解析其依赖项。此外，[UseMiddleware<T>](#) 还可直接接受其他参数。

按请求依赖项

由于中间件是在应用启动时构造的，而不是按请求构造的，因此在每个请求过程中，中间件构造函数使用的范围内生存期服务不与其他依赖关系注入类型共享。如果必须在中间件和其他类型之间共享范围内服务，请将这些服务添加到 `Invoke` 方法的签名。`Invoke` 方法可接受由依赖关系注入填充的其他参数。例如：

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext, IMyScopedService svc)
    {
        svc.MyProperty = 1000;
        await _next(httpContext);
    }
}
```

其他资源

- [将 HTTP 模块迁移到中间件](#)
- [应用程序启动](#)
- [请求功能](#)
- [基于工厂的中间件激活](#)
- [第三方容器中的中间件激活](#)

ASP.NET Core 中间件

2018/5/14 • 11 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Steve Smith](#)

[查看或下载示例代码\(如何下载\)](#)

什么是中间件？

中间件是一种装配到应用程序管道以处理请求和响应的软件。每个组件：

- 选择是否将请求传递到管道中的下一个组件。
- 可在调用管道中的下一个组件前后执行工作。

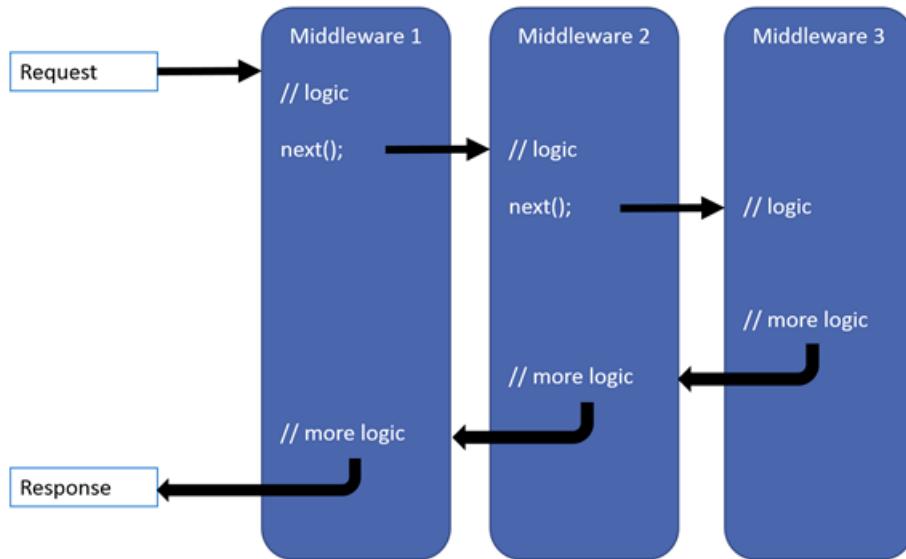
请求委托用于生成请求管道。请求委托处理每个 HTTP 请求。

使用 [Run](#)、[Map](#) 和 [Use](#) 扩展方法来配置请求委托。可将一个单独的请求委托并行指定为匿名方法(称为并行中间件)，或在可重用的类中对其进行定义。这些可重用的类和并行匿名方法即为中间件或中间件组件。请求管道中的每个中间件组件负责调用管道中的下一个组件，或在适当情况下使链发生短路。

[将 HTTP 模块迁移到中间件](#)介绍了 ASP.NET Core 和 ASP.NET 4.x 中请求管道之间的差异，并提供了更多的中间件示例。

使用 [IApplicationBuilder](#) 创建中间件管道

ASP.NET Core 请求管道包含一系列相继调用的请求委托，如下图所示(执行过程遵循黑色箭头)：



每个委托均可在下一个委托前后执行操作。此外，委托还可以决定不将请求传递给下一个委托，这就是对请求管道进行短路。通常需要短路，因为这样可以避免不必要的工作。例如，静态文件中间件可以返回静态文件请求并使管道的其余部分短路。需要尽早在管道中调用异常处理委托，以便它们可以捕获在管道的后期阶段所发生的异常。

尽可能简单的 ASP.NET Core 应用设置了处理所有请求的单个请求委托。这种情况不包括实际请求管道。调用单个匿名函数以响应每个 HTTP 请求。

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}

```

第一个 `app.Run` 委托终止了管道。

可使用 `app.Use` 将多个请求委托链接在一起。`next` 参数表示管道中的下一个委托。(请记住, 可通过不调用 `next` 参数使管道短路。)通常可在下一个委托前后执行操作, 如以下示例所示:

```

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}

```

警告

在向客户端发送响应后, 请勿调用 `next.Invoke`。响应启动后, 针对 `HttpResponse` 的更改将引发异常。例如, 设置标头、状态代码等更改将引发异常。调用 `next` 后写入响应正文:

- 可能导致违反协议。例如, 写入的长度超过规定的 `content-length`。
- 可能损坏正文格式。例如, 向 CSS 文件中写入 HTML 页脚。

`HttpResponse.HasStarted` 是一个有用的提示, 指示是否已发送标头和/或已写入正文。

中间件排序

向 `Configure` 方法添加中间件组件的顺序定义了针对请求调用这些组件的顺序, 以及响应的相反顺序。此排序对于安全性、性能和功能至关重要。

`Configure` 方法(如下所示)添加以下中间件组件:

1. 异常/错误处理
2. 静态文件服务器
3. 身份验证

4. MVC

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void Configure(IApplicationBuilder app)
{
    app.UseExceptionHandler("/Home/Error"); // Call first to catch exceptions
                                                // thrown in the following middleware.

    app.UseStaticFiles();                      // Return static files and end pipeline.

    app.UseAuthentication();                   // Authenticate before you access
                                                // secure resources.

    app.UseMvcWithDefaultRoute();             // Add MVC to the request pipeline.
}
```

在以上代码中，`UseExceptionHandler` 是添加到管道的第一个中间件组件，因此，该组件可捕获在后面的调用中发生的任何异常。

因为尽早在管道中调用静态文件中间件，因此该组件可处理请求并使引致短路，而无需通过剩余组件。静态文件中间件不提供授权检查。可公开访问由静态文件中间件服务的任何文件，包括 wwwroot 下的文件。请参阅[静态文件](#)，了解如何保护静态文件。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果静态文件中间件未处理请求，则请求将被传递给执行身份验证的标识中间件（`app.UseAuthentication`）。标识不使未经身份验证的请求短路。虽然标识对请求进行身份验证，但仅在 MVC 选择特定 Razor 页或控制器和操作后，才发生授权（和拒绝）。

以下示例演示中间件排序，其中静态文件的请求在响应压缩中间件前由静态文件中间件进行处理。静态文件未通过此中间件排序进行压缩。可压缩来自 `UseMvcWithDefaultRoute` 的 MVC 响应。

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();          // Static files not compressed
                                  // by middleware.

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

Use、Run 和 Map

使用 `Use`、`Run` 和 `Map` 配置 HTTP 管道。`Use` 方法可使管道短路（即不调用 `next` 请求委托）。`Run` 是一种约定，并且某些中间件组件可公开在管道末尾运行的 `Run[Middleware]` 方法。

`Map*` 扩展用作约定来创建管道分支。`Map` 基于给定请求路径的匹配项来创建请求管道分支。如果请求路径以给定路径开头，则执行分支。

```

public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}

```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应：

请求	响应
localhost:1234	来自非 Map 委托的 Hello。
localhost:1234/map1	Map 测试 1
localhost:1234/map2	Map 测试 2
localhost:1234/map3	来自非 Map 委托的 Hello。

使用 `Map` 时，将从 `HttpRequest.Path` 中删除匹配的线段，并针对每个请求将该线段追加到 `HttpRequest.PathBase`。

`MapWhen` 基于给定谓词的结果创建请求管道分支。`Func<HttpContext, bool>` 类型的任何谓词均可用于将请求映射到管道的新分支。在以下示例中，谓词用于检测查询字符串变量 `branch` 是否存在：

```

public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                    HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}

```

下表使用前面的代码显示来自 `http://localhost:1234` 的请求和响应：

请求	响应
localhost:1234	来自非 Map 委托的 Hello。
localhost:1234/?branch=master	使用的分支 = 主要分支

`Map` 支持嵌套，例如：

```

app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a"
        //...
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b"
        //...
    });
});

```

此外，`Map` 还可同时匹配多个段，例如：

```
app.Map("/level1/level2", HandleMultiSeg);
```

内置中间件

ASP.NET Core 附带以下中间件组件，以及用于添加这些组件的顺序的说明：

中间件	描述	顺序
身份验证	提供身份验证支持。	在需要 <code>HttpContext.User</code> 之前。OAuth 回叫的终端。

中间件	描述	顺序
CORS	配置跨域资源共享。	在使用 CORS 的组件之前。
诊断	配置诊断。	在生成错误的组件之前。
ForwardedHeaders/HttpOverrides	将代理标头转发到当前请求。	在使用更新的字段(示例:Scheme、Host、ClientIP、Method)的组件之前。
响应缓存	提供对缓存响应的支持。	在需要缓存的组件之前。
响应压缩	提供对压缩响应的支持。	在需要压缩的组件之前。
RequestLocalization	提供本地化支持。	在对本地化敏感的组件之前。
路由	定义和约束请求路由。	用于匹配路由的终端。
会话	提供对管理用户会话的支持。	在需要会话的组件之前。
静态文件	为提供静态文件和目录浏览提供支持。	如果请求与文件匹配, 则为终端。
URL 重写	提供对重写 URL 和重定向请求的支持。	在使用 URL 的组件之前。
WebSockets	启用 WebSockets 协议。	在接受 WebSocket 请求所需的组件之前。

写入中间件

通常, 中间件封装在类中, 并且通过扩展方法公开。请考虑以下中间件, 该中间件通过查询字符串设置当前请求的区域性:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use((context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return next();
        });

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

注意：以上示例代码用于演示创建中间件组件。有关 ASP.NET Core 的内置本地化支持，请参阅[全球化和本地化](#)。

可通过传入区域性（如 `http://localhost:7997/?culture=no`）测试中间件。

以下代码将中间件委托移动到类：

```
using Microsoft.AspNetCore.Http;
using System.Globalization;
using System.Threading.Tasks;

namespace Culture
{
    public class RequestCultureMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestCultureMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public Task InvokeAsync(HttpContext context)
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            return this._next(context);
        }
    }
}
```

注意

在 ASP.NET Core 1.x 中，中间件 `Task` 方法的名称必须是 `Invoke`。在 ASP.NET Core 2.0 或更高版本中，该名称可以为 `Invoke` 或 `InvokeAsync`。

以下扩展方法通过 `IApplicationBuilder` 公开中间件：

```
using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}
```

以下代码通过 `Configure` 调用中间件：

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

中间件应通过在其构造函数中公开其依赖项来遵循[显式依赖项原则](#)。在每个应用程序生存期构造一次中间件。如果需要与请求中的中间件共享服务，请参阅下面讲述的按请求依赖项。

中间件组件可通过构造函数参数从依赖关系注入解析其依赖项。此外，[UseMiddleware<T>](#) 还可直接接受其他参数。

按请求依赖项

由于中间件是在应用启动时构造的，而不是按请求构造的，因此在每个请求过程中，中间件构造函数使用的范围内生存期服务不与其他依赖关系注入类型共享。如果必须在中间件和其他类型之间共享范围内服务，请将这些服务添加到 [Invoke](#) 方法的签名。[Invoke](#) 方法可接受由依赖关系注入填充的其他参数。例如：

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext, IMyScopedService svc)
    {
        svc.MyProperty = 1000;
        await _next(httpContext);
    }
}
```

其他资源

- [将 HTTP 模块迁移到中间件](#)
- [应用程序启动](#)
- [请求功能](#)
- [基于工厂的中间件激活](#)
- [第三方容器中的中间件激活](#)

ASP.NET Core 中基于工厂的中间件激活

2018/5/14 • 2 min to read • [Edit Online](#)

作者: [Luke Latham](#)

`IMiddlewareFactory`/`IMiddleware` 是中间件激活的扩展点。

`UseMiddleware` 扩展方法检查中间件的已注册类型是否实现 `IMiddleware`。如果是，则使用在容器中注册的 `IMiddlewareFactory` 实例来解析 `IMiddleware` 实现，而不使用基于约定的中间件激活逻辑。中间件在应用的服务容器中注册为作用域或瞬态服务。

优点：

- 按请求(作用域服务的注入)激活
- 让中间件强类型化

`IMiddleware` 按请求激活，因此作用域服务可以注入到中间件的构造函数中。

[查看或下载示例代码\(如何下载\)](#)

示例应用演示了使用以下两种方式激活的中间件：

- 约定。有关使用约定激活中间件的详细信息，请参阅 [中间件](#) 主题。
- `IMiddleware` 实现。默认的 `MiddlewareFactory` 类可激活中间件。

这两种中间件实现的功能相同，并能记录由查询字符串参数 (`key`) 提供的值。中间件使用插入的数据库上下文(作用域服务)将查询字符串值记录在内存中数据库。

IMiddleware

`IMiddleware` 定义应用的请求管道的中间件。`InvokeAsync(HttpContext, RequestDelegate)` 方法处理请求，并返回代表中间件执行的 `Task`。

使用约定激活的中间件：

```

public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, AppDbContext db)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            db.Add(new Request()
            {
                DT = DateTime.UtcNow,
                MiddlewareActivation = "ConventionalMiddleware",
                Value = keyValue
            });
        }

        await db.SaveChangesAsync();
    }

    await _next(context);
}

```

使用 `MiddlewareFactory` 激活的中间件：

```

public class IMiddlewareMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public IMiddlewareMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
            {
                DT = DateTime.UtcNow,
                MiddlewareActivation = "IMiddlewareMiddleware",
                Value = keyValue
            });
        }

        await _db.SaveChangesAsync();
    }

    await next(context);
}

```

程序会为中间件创建扩展：

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<ConventionalMiddleware>();
    }

    public static IApplicationBuilder UseIMiddlewareMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<IMiddlewareMiddleware>();
    }
}
```

无法通过 `UseMiddleware` 将对象传递给工厂激活的中间件：

```
public static IApplicationBuilder UseIMiddlewareMiddleware(
    this IApplicationBuilder builder, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at runtime.
    return builder.UseMiddleware<IMiddlewareMiddleware>(option);
}
```

将工厂激活的中间件添加到 `Startup.cs` 的内置容器中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>
        options.UseInMemoryDatabase("InMemoryDb"));

    services.AddTransient<IMiddlewareMiddleware>();

    services.AddMvc();
}
```

两个中间件均在 `Configure` 的请求处理管道中注册：

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

    app.UseConventionalMiddleware();
    app.UseIMiddlewareMiddleware();

    app.UseStaticFiles();
    app.UseMvc();
}
```

IMiddlewareFactory

[IMiddlewareFactory](#) 提供中间件的创建方法。中间件工厂实现在容器中注册为作用域服务。

可在 [Microsoft.AspNetCore.Http](#) 包(引用源)中找到默认的 `IMiddlewareFactory` 实现(即`MiddlewareFactory`)。

其他资源

- [中间件](#)
- [第三方容器中的中间件激活](#)

使用 ASP.NET Core 中的第三方容器激活中间件

2018/5/14 • 2 min to read • [Edit Online](#)

作者: [Luke Latham](#)

本文演示如何使用 `IMiddlewareFactory` 和 `IMiddleware` 作为使用第三方容器激活中间件的可扩展点。有关 `IMiddlewareFactory` 和 `IMiddleware` 的介绍性信息, 请参阅[基于工厂的中间件激活](#)主题。

[查看或下载示例代码\(如何下载\)](#)

示例应用演示了使用 `IMiddlewareFactory`、`SimpleInjectorMiddlewareFactory` 实现激活的中间件。此示例使用 [Simple Injector](#) 依赖项注入 (DI) 容器。

此示例的中间件实现记录了查询字符串参数 (`key`) 提供的值。中间件使用插入的数据库上下文(有作用域的服务)将查询字符串值记录在内存中数据库。

注意

此示例应用仅出于演示目的使用 [Simple Injector](#)。不认可使用 Simple Injector。Simple Injector 文档中描述的中间件激活方法和 Simple Injector 维护人员推荐的 GitHub 问题。有关详细信息, 请参阅 [Simple Injector 文档](#)和 [Simple Injector GitHub 存储库](#)。

IMiddlewareFactory

`IMiddlewareFactory` 提供中间件的创建方法。

在示例应用中, 实现了中间件工厂以创建 `SimpleInjectorActivatedMiddleware` 实例。中间件工厂使用 Simple Injector 容器来解析中间件:

```
public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;

    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

`IMiddleware` 定义应用的请求管道的中间件。

由 `IMiddlewareFactory` 实现 (`Middleware/SimpleInjectorActivatedMiddleware.cs`) 激活的中间件:

```

public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
            {
                DT = DateTime.UtcNow,
                MiddlewareActivation = "SimpleInjectorActivatedMiddleware",
                Value = keyValue
            });
        }

        await _db.SaveChangesAsync();
    }

    await next(context);
}

```

为中间件创建扩展 (Middleware/MiddlewareExtensions.cs):

```

public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}

```

`Startup.ConfigureServices` 必须执行多项任务:

- 设置 Simple Injector 容器。
- 注册工厂和中间件。
- 为 Razor 页面的 Simple Injector 容器创建应用的数据上下文。

```

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

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
        optionsBuilder.UseInMemoryDatabase("InMemoryDb");
        return new AppDbContext(optionsBuilder.Options);
    }, Lifestyle.Scoped);

    _container.Register<SimpleInjectorActivatedMiddleware>();

    _container.Verify();
}

```

中间件在 `Startup.Configure` 的请求处理管道中注册：

```

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

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseMvc();
}

```

其他资源

- [中间件](#)
- [基于工厂的中间件激活](#)
- [Simple Injector GitHub 存储库](#)
- [Simple Injector 文档](#)

在 ASP.NET Core 中处理静态文件

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Scott Addie](#)

静态文件(如 HTML、CSS、图像和 JavaScript)是 ASP.NET Core 应用直接提供给客户端的资产。需要进行一些配置才能提供这些文件。

[查看或下载示例代码\(如何下载\)](#)

提供静态文件

静态文件存储在项目的 Web 根目录中。默认目录是 <content_root>/wwwroot, 但可通过 [UseWebRoot](#) 方法更改目录。有关详细信息, 请参阅[内容根目录](#)和[Web 根目录](#)。

应用的 Web 主机必须识别内容根目录。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

采用 `WebHost.CreateDefaultBuilder` 方法可将内容根目录设置为当前目录:

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

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

可通过 Web 根目录的相关路径访问静态文件。例如, Web 应用程序项目模板包含 wwwroot 文件夹中的多个文件夹:

- **wwwroot**
 - **css**
 - **images**
 - **js**

用于访问 images 子文件夹中的文件的 URI 格式为 `http://<server_address>/images/<image_file_name>`。例如, *<http://localhost:9189/images/banner3.svg>*。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果以 .NET Framework 为目标, 请将 `Microsoft.AspNetCore.StaticFiles` 包添加到项目。如果以 .NET Core 为目标, 请将 `Microsoft.AspNetCore.All` 元包加入此包。

配置提供静态文件的[中间件](#)。

提供 Web 根目录内的文件

调用 `Startup.Configure` 中的 `UseStaticFiles` 方法：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

无参数 `UseStaticFiles` 方法重载将 Web 根目录中的文件标记为可用。以下标记引用 `wwwroot/images/banner1.svg`：

```

```

提供 Web 根目录外的文件

考虑一个目录层次结构，其中要提供的静态文件位于 Web 根目录之外：

- **wwwroot**
 - **css**
 - **images**
 - **js**
- **MyStaticFiles**
 - **images**
 - `banner1.svg`

按如下方式配置静态文件中间件后，请求可访问 `banner1.svg` 文件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });
}
```

在前面的代码中，`MyStaticFiles` 目录层次结构通过 `StaticFiles` URI 段公开。请求 `http://<server_address>/StaticFiles/images/banner1.svg` 提供 `banner1.svg` 文件。

以下标记引用 `MyStaticFiles/images/banner1.svg`：

```

```

设置 HTTP 响应标头

`StaticFileOptions` 对象可用于设置 HTTP 响应标头。除配置从 Web 根目录提供静态文件外，以下代码还设置 `Cache-Control` 标头：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions
    {
        OnPrepareResponse = ctx =>
        {
            // Requires the following import:
            // using Microsoft.AspNetCore.Http;
            ctx.Context.Response.Headers.Append("Cache-Control", "public,max-age=600");
        }
    });
}
```

[HeaderDictionaryExtensions.Append](#) 方法存在于 [Microsoft.AspNetCore.Http](#) 包中。

可公开缓存这些文件 10 分钟(600 秒)：

▼ Response Headers [view source](#)

Accept-Ranges: bytes
Cache-Control: public,max-age=600
Content-Length: 143058
Content-Type: image/png
Date: Wed, 01 Feb 2017 01:27:01 GMT
ETag: "1d1909cf92bacd2"
Last-Modified: Thu, 07 Apr 2016 07:13:24 GMT

静态文件授权

静态文件中间件不提供授权检查。可公开访问由静态文件中间件提供的任何文件，包括 wwwroot 下的文件。根据授权提供文件：

- 将文件存储在 wwwroot 和静态文件中间件可访问的任何目录之外并
- 通过有授权的操作方法提供文件。返回 [FileResult](#) 对象：

```
[Authorize]
public IActionResult BannerImage()
{
    var file = Path.Combine(Directory.GetCurrentDirectory(),
                           "MyStaticFiles", "images", "banner1.svg");

    return PhysicalFile(file, "image/svg+xml");
}
```

启用目录浏览

通过目录浏览，Web 应用的用户可查看目录列表和指定目录中的文件。出于安全考虑，目录浏览默认处于禁用状态(请参阅[注意事项](#))。调用 [Startup.Configure](#) 中的 [UseDirectoryBrowser](#) 方法来启用目录浏览：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

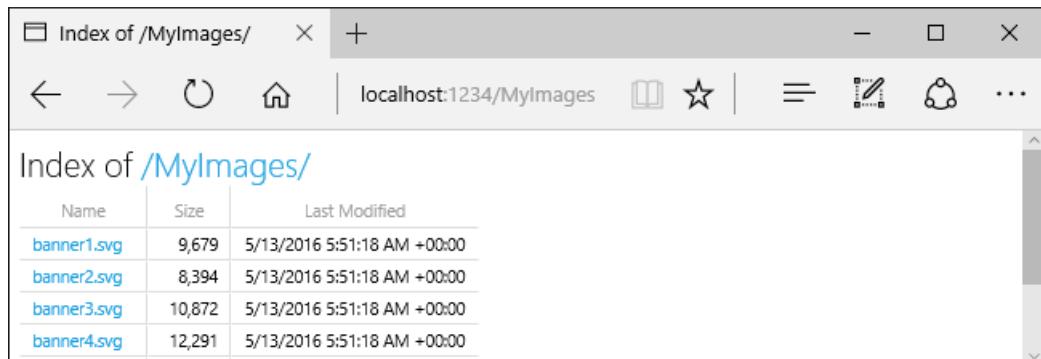
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

调用 `Startup.ConfigureServices` 中的 `AddDirectoryBrowser` 方法来添加所需服务：

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

上述代码允许使用 URL `http://<server_address>/MyImages` 浏览 `wwwroot/images` 文件夹的目录，并链接到每个文件和文件夹：



有关启用浏览时的安全风险，请参阅[注意事项](#)。

请注意以下示例中的两个 `UseStaticFiles` 调用。第一个调用提供 `wwwroot` 文件夹中的静态文件。第二个调用使用 URL `http://<server_address>/MyImages` 浏览 `wwwroot/images` 文件夹的目录：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

提供默认文档

设置默认主页为访问者访问网站时提供了逻辑起点。若要在用户不完全限定 URI 的情况下提供默认页面，请调用 `Startup.Configure` 中的 `UseDefaultFiles` 方法：

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

重要事项

要提供默认文件，必须在 `UseStaticFiles` 前调用 `UseDefaultFiles`。`UseDefaultFiles` 实际上用于重写 URL，不提供文件。通过 `UseStaticFiles` 启用静态文件中间件来提供文件。

使用 `UseDefaultFiles` 请求文件夹搜索：

- default.htm
- default.html
- index.htm
- index.html

将请求视为完全限定 URI，提供在列表中找到的第一个文件。浏览器 URL 继续反映请求的 URI。

以下代码将默认文件名更改为 mydefault.html：

```
public void Configure(IApplicationBuilder app)
{
    // Serve my app-specific default file, if present.
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
    app.UseStaticFiles();
}
```

UseFileServer

`UseFileServer` 结合了 `UseStaticFiles`、`UseDefaultFiles` 和 `UseDirectoryBrowser` 的功能。

以下代码提供静态文件和默认文件。未启用目录浏览。

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

以下代码通过启用目录浏览基于无参数重载进行构建：

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

考虑以下目录层次结构：

- **wwwroot**
 - **css**
 - **images**
 - **js**
- **MyStaticFiles**
 - **images**
 - banner1.svg
 - default.html

以下代码启用静态文件、默认文件和及 `MyStaticFiles` 的目录浏览：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseFileServer(new FileServerOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles",
        EnableDirectoryBrowsing = true
    });
}
```

`EnableDirectoryBrowsing` 属性值为 `true` 时必须调用 `AddDirectoryBrowser`：

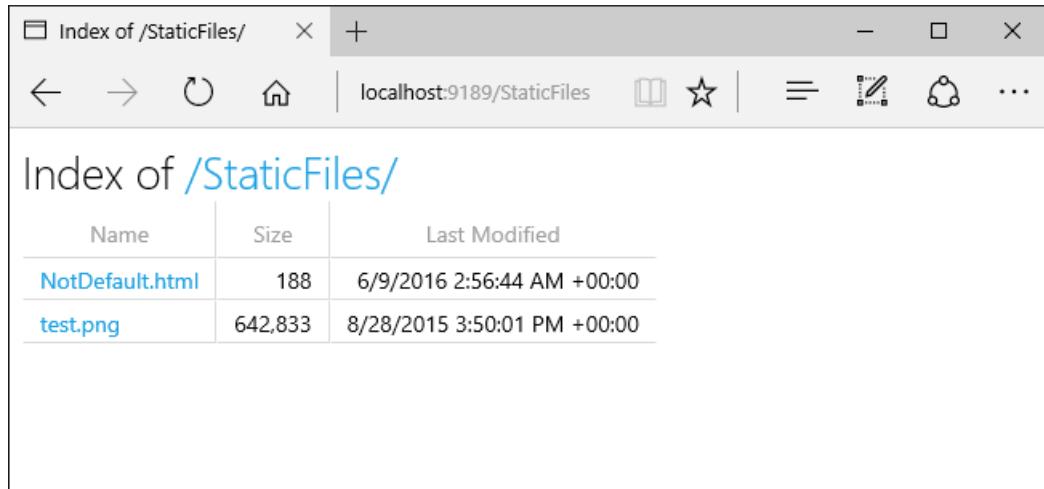
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

使用文件层次结构和前面的代码，URL 解析如下：

URI	响应
http://<server_address>/StaticFiles/images/banner1.svg	MyStaticFiles/images/banner1.svg
http://<server_address>/StaticFiles	MyStaticFiles/default.html

如果 `MyStaticFiles` 目录中不存在默认命名文件，则 `http://<server_address>/StaticFiles` 返回包含可单击链

接的目录列表：



注意

`UseDefaultFiles` 和 `UseDirectoryBrowser` 使用不带尾部反斜杠的 URL `http://<server_address>/StaticFiles` 触发客户端并重定向到 `http://<server_address>/StaticFiles/`。注意添加尾部反斜杠。无尾部反斜杠，文档中的相对 URL 被视为无效。

FileExtensionContentTypeProvider

`FileExtensionContentTypeProvider` 类包含 `Mappings` 属性，用作文件扩展名到 MIME 内容类型的映射。在以下示例中，多个文件扩展名注册到了已知的 MIME 类型。替换了 `.rtf` 扩展名，删除了 `.mp4`。

```
public void Configure(IApplicationBuilder app)
{
    // Set up custom content types - associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages",
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

请参阅 [MIME 内容类型](#)。

非标准内容类型

静态文件中间件可理解近 400 种已知文件内容类型。如果用户请求未知文件类型的文件，则静态文件中间件会返回 HTTP 404(未找到)响应。如果启用了目录浏览，则会显示该文件的链接。URI 返回 HTTP 404 错误。

以下代码提供未知类型，并以图像形式呈现未知文件：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}
```

使用前面的代码，请求的文件含未知内容类型时，以图像形式返回请求。

警告

启用 `ServeUnknownFileTypes` 存在安全风险。它默认处于禁用状态，不建议使用。`FileExtensionContentTypeProvider` 提供了更安全的替代方法来提供含非标准扩展名的文件。

注意事项

警告

`UseDirectoryBrowser` 和 `UseStaticFiles` 可能会泄漏机密。强烈建议在生产中禁用目录浏览。请仔细查看 `UseStaticFiles` 或 `UseDirectoryBrowser` 启用了哪些目录。整个目录及其子目录均可公开访问。将适合公开的文件存储在专用目录中，如 `<content_root>/wwwroot`。将这些文件与 MVC 视图、Razor 页面(仅限 2.x)和配置文件等分开

- 使用 `UseDirectoryBrowser` 和 `UseStaticFiles` 公开的内容的 URL 受大小写和基础文件系统字符限制的影响。例如，Windows 不区分大小写 — macOS 和 Linux 却要区分。
- 托管于 IIS 中的 ASP.NET Core 应用使用 [ASP.NET Core 模块](#) 将所有请求转发到应用，包括静态文件请求。未使用 IIS 静态文件处理程序。在模块处理请求前，处理程序没有机会处理请求。
- 在 IIS Manager 中完成以下步骤，删除服务器或网站级别的 IIS 静态文件处理程序：
 1. 转到“模块”功能。
 2. 在列表中选择 `StaticFileModule`。
 3. 单击“操作”侧栏中的“删除”。

警告

如果启用了 IIS 静态文件处理程序且 ASP.NET Core 模块配置不正确，则会提供静态文件。例如，如果未部署 `web.config` 文件，则会发生这种情况。

- 将代码文件(包括 `.cs` 和 `.cshtml`)放在应用项目的 Web 根目录之外。这样就在应用的客户端内容和基于服务器的代码间创建了逻辑分隔。可以防止服务器端代码泄漏。

其他资源

- 中间件
- [ASP.NET Core 简介](#)

ASP.NET Core 中的路由

2018/5/14 • 24 min to read • [Edit Online](#)

撰写者: [Ryan Nowak](#)、[Steve Smith](#) 和 [Rick Anderson](#)

路由功能负责将传入请求映射到路由处理程序。路由在 ASP.NET 应用中定义，并在应用启动时进行配置。路由可以选择从请求包含的 URL 中提取值，然后这些值便可用于处理请求。使用 ASP.NET 应用中的路由信息，路由功能还能生成要映射到路由处理程序的 URL。因此，路由可以根据 URL 查找路由处理程序，或者根据路由处理程序信息查找给定路由处理程序对应的 URL。

重要事项

本文档介绍较低级别的 ASP.NET Core 路由。有关 ASP.NET Core MVC 路由的信息，请参阅[路由到控制器操作](#)

[查看或下载示例代码\(如何下载\)](#)

路由基础知识

路由使用路由 ([IRouter](#) 的实现) 来:

- 将传入请求映射到路由处理程序
- 生成响应中使用的 URL

通常情况下，应用具有一个路由集合。请求到达时，将按顺序处理路由集合。传入请求通过对路由集合中的每个可用路由调用 `RouteAsync` 方法来查找与请求 URL 匹配的路由。与此相反，响应可根据路由信息使用路由生成 URL(例如，用于重定向或链接)，并因此避免需要硬编码 URL，这有助于可维护性。

路由通过 `RouterMiddleware` 类连接到[中间件](#)管道。ASP.NET Core MVC 向中间件管道添加路由，作为其配置的一部分。若要了解如何使用路由作为独立组件，请参阅[使用路由中间件](#)。

URL 匹配

URL 匹配是一个过程，通过该过程，路由可向处理程序调度传入请求。此过程通常基于 URL 路径中的数据，但可进行扩展以考虑请求中的任何数据。向单独的处理程序调度请求的功能是缩放应用程序的大小和复杂性的关键。

传入请求将进入 `RouterMiddleware`，后者将对序列中的每个路由调用 `RouteAsync` 方法。`IRouter` 实例将选择是否通过将 `HttpContext.Handler` 设置为非空 `RequestDelegate` 来处理请求。如果路由为请求设置处理程序，路由处理将停止，处理程序将被调用以处理该请求。如果尝试了所有路由，且请求未找到任何处理程序，中间件将调用 `next`，请求管道中的下一个中间件将被调用。

`RouteAsync` 的主要输入是与当前请求关联的 `HttpContext`。`Handler` 和 `RouteData` 是将在路由匹配后设置的输出。

`RouteAsync` 期间的匹配还可根据迄今完成的请求处理将 `RouteData` 的属性设为适当的值。如果路由与请求匹配，`RouteData` 将包含有关结果的重要状态信息。

`RouteData.Values` 是从路由中生成的路由值的字典。这些值通常通过标记 URL 来确定，可用来接受用户输入，或者在应用程序内作出进一步的调度决策。

`RouteData.DataTokens` 是一个与匹配的路由相关的其他数据的属性包。提供 `DataTokens` 以支持将状态数据与每个路由相关联，以便应用程序稍后可根据所匹配的路由作出决策。这些值是开发者定义的，不会影响通过任何方式路由的行为。此外，存储于数据令牌中的值可以属于任何类型，与路由值相反，后者必须能够轻松转换

为字符串，或从字符串进行转换。

`RouteData.Routers` 是参与成功匹配请求的路由的列表。路由可以嵌套在另一路由内，并且 `Routers` 属性可以通过导致匹配的逻辑路由树反映该路径。通常情况下，`Routers` 中的第一项是路由集合，应该用于生成 URL。`Routers` 中的最后一项是匹配的路由处理程序。

URL 生成

URL 生成是通过其可根据一组路由值创建 URL 路径的过程。这需要考虑处理程序与访问它们的 URL 之间的逻辑分隔。

URL 遵循类似的迭代过程，但开头是将用户或框架代码调用到路由集合的 `GetVirtualPath` 方法中。然后，每个路由会有序地调用其 `GetVirtualPath` 方法，直到返回非空的 `VirtualPathData`。

`GetVirtualPath` 的主输入有：

- `VirtualPathContext.HttpContext`
- `VirtualPathContext.Values`
- `VirtualPathContext.AmbientValues`

路由主要使用 `Values` 和 `AmbientValues` 提供的路由值来决定可能生成 URL 的位置以及要包括的值。

`AmbientValues` 是路由值的集合，这些值是通过将当前请求与路由系统相匹配而产生的。与此相反，`Values` 是指定如何为当前操作生成所需 URL 的路由值。当路由需要获取与当前上下文关联的服务或其他数据时，需提供 `HttpContext`。

提示: 将 `Values` 作为 `AmbientValues` 的一组替代值。URL 生成尝试重复使用当前请求中的路由值，以便轻松使用相同路由或路由值生成链接的 URL。

`GetVirtualPath` 的输出是 `VirtualPathData`。`VirtualPathData` 是 `RouteData` 的并行值，它包含输出 URL 的 `VirtualPath` 以及路由应该设置的某些其他属性。

`VirtualPathData.VirtualPath` 属性包含路由生成的虚拟路径。你可能需要进一步处理路径，具体取决于你的需求。例如，如果要在 HTML 中呈现生成的 URL，需要预先计算应用程序的基路径。

`VirtualPathData.Router` 是对已成功生成 URL 的路由的引用。

`VirtualPathData.DataTokens` 属性是生成 URL 的路由的其他相关数据的字典。它是 `RouteData.DataTokens` 的并行值。

创建路由

路由提供 `Route` 类，作为 `IRouter` 的标准实现。`Route` 使用 route template 语法来定义调用 `RouteAsync` 时将针对 URL 路径进行匹配的模式。调用 `GetVirtualPath` 时，`Route` 将使用相同的路由模板生成 URL。

大多数应用程序通过调用 `MapRoute` 或 `IRouteBuilder` 上定义的一种类似的方法来创建路由。所有方法都将创建 `Route` 的实例，并将其添加到路由集合中。

注意: `MapRoute` 不采用任何路由处理程序参数，它只添加将由 `DefaultHandler` 处理的路由。由于默认处理程序是 `IRouter`，它可能决定不处理该请求。例如，Microsoft ASP.NET MVC 通常被配置为默认处理程序，仅处理与可用控制器和操作匹配的请求。要了解路由到 MVC 的详细信息，请参阅[路由到控制器操作](#)。

下面是典型 ASP.NET MVC 路由定义使用的一个 `MapRoute` 调用示例：

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id?}");
```

此模板将匹配类似 `/Products/Details/17` 的 URL 路径并提取路由值

{ controller = Products, action = Details, id = 17 }。路由值是通过将 URL 路径拆分成段，并且将每段与路由模板中的路由参数名称相匹配来确定的。路由参数已命名。它们是通过将参数名称括在大括号 {} 中进行定义的。

上述模板还可匹配 URL 路径 /，并且将生成值 { controller = Home, action = Index }。这是因为 {controller} 和 {action} 路由参数具有默认值，而 id 路由参数是可选的。路由参数名称为参数定义默认值后，等号 = 后将有一个值。路由参数名称后的问号 ? 将参数定义为可选。具有默认值“始终”的路由参数将在路由匹配时生成路由值，如果没有对应的 URL 路径段，可选参数不会生成路由值。

有关路由模板功能和语法的详细说明，请参阅 [route-template-reference](#)。

此示例包括路由约束：

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id:int}");
```

此模板将匹配类似 /Products/Details/17 而不是 /Products/Details/Apples 的 URL 路径。路由参数定义 {id:int} 为 id 路由参数定义路由约束。路由约束实现 IRouteConstraint 并检查路由值，以验证它们。在此示例中，路由值 id 必须可转换为整数。有关框架提供的路由约束的更详细说明，请参阅 [route-constraint-reference](#)。

其他 MapRoute 重载接受 constraints、dataTokens 和 defaults 的值。MapRoute 的此类附加参数被定义为 object 类型。这些参数的典型用法是传递匿名类型化对象，其中匿名类型的属性名匹配路由参数名称。

以下两个示例可创建等效路由：

```
routes.MapRoute(  
    name: "default_route",  
    template: "{controller}/{action}/{id?}",  
    defaults: new { controller = "Home", action = "Index" });  
  
routes.MapRoute(  
    name: "default_route",  
    template: "{controller=Home}/{action=Index}/{id?}");
```

提示：定义约束和默认值的内联语法对简单路由可以更方便。但是，存在内联语法不支持的功能，例如数据令牌。

此示例演示其他几个功能：

```
routes.MapRoute(  
    name: "blog",  
    template: "Blog/{*article}",  
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

此模板将匹配类似 /Blog/All-About-Routing/Introduction 的 URL 路径并提取值 { controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }。controller 和 action 的默认路由值由路由生成，即便模板中没有对应的路由参数。可在路由模板中指定默认值。根据路由参数名称前的星号 外观，article 路由参数被定义为全方位 *。全方位路由参数可捕获 URL 路径的其余部分，也能匹配空白字符串。

此示例可添加路由约束和数据令牌：

```

routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });

```

此模板将匹配类似 `/Products/5` 的 URL 路径并提取值 `{ controller = Products, action = Details, id = 5 }` 和数据令牌 `{ locale = en-US }`。

Locals		
Name	Value	Type
Response	{Microsoft.AspNetCore.Http.Internal.DefaultHttpResponse}	Microsoft
RouteData	{Microsoft.AspNetCore.Routing.RouteData}	Microsoft
DataTokens	{Microsoft.AspNetCore.Routing.RouteValueDictionary}	Microsoft
Comparer	{System.OrdinalComparer}	System.
Count	1	int
Keys	{string[1]}	System.C
[0]	"locale"	string
Values	{object[1]}	System.C
[0]	"en-US"	object {s
Non-Public me		
Results View	Expanding the Results View will enumerate the IEnumera	
Routers	Count = 3	System.C
[0]	{Microsoft.AspNetCore.Routing.RouteCollection}	Microsoft
[1]	{en-US/Products/{id}}	Microsoft
[2]	{Microsoft.AspNetCore.Mvc.Internal.MvcRouteHandler}	Microsoft

Output Autos Watch 1

URL 生成

`Route` 类还可通过将一组路由值与其路由模板组合来生成 URL。从逻辑上来说，这是匹配 URL 路径的反向过程。

提示：为更好了解 URL 生成，请想象你要生成的 URL，然后考虑路由模板将如何匹配该 URL。将生成什么值？这大致相当于 URL 在 `Route` 类中的生成方式。

此示例使用基本的 ASP.NET MVC 样式路由：

```

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

```

使用路由值 `{ controller = Products, action = List }`，此路由将生成 URL `/Products/List`。路由值将替换为相应的路由参数，以形成 URL 路径。由于 `id` 属于可选路由参数，因此它可以没有值。

使用路由值 `{ controller = Home, action = Index }`，此路由将生成 URL `/`。提供的路由值匹配默认值，因此可以安全忽略这些值对应的段。请注意，已生成的两个 URL 将往返这个路由定义，并生成用于生成该 URL 的相同路由值。

提示：使用 ASP.NET MVC 应用应该使用 `UrlHelper` 生成 URL，而不是直接调用到路由。

有关 URL 生成过程的更多详细信息，请参阅 [url-generation-reference](#)。

使用路由中间件

添加 NuGet 包“Microsoft.AspNetCore.Routing”。

将路由添加到 `Startup.cs` 中的服务容器：

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

必须在 `Startup` 类的 `Configure` 方法中配置路由。下面的示例使用这些 API:

- `RouteBuilder`
- `Build`
- `MapGet` 仅匹配 HTTP GET 请求
- `UseRouter`

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    var trackPackageRouteHandler = new RouteHandler(context =>
    {
        var routeValues = context.GetRouteData().Values;
        return context.Response.WriteAsync(
            $"Hello! Route values: {string.Join(", ", routeValues)}");
    });

    var routeBuilder = new RouteBuilder(app, trackPackageRouteHandler);

    routeBuilder.MapRoute(
        "Track Package Route",
        "package/{operation:regex^(track|create|detonate)$}/{id:int}");

    routeBuilder.MapGet("hello/{name}", context =>
    {
        var name = context.GetRouteValue("name");
        // This is the route handler when HTTP GET "hello/<anything>" matches
        // To match HTTP GET "hello/<anything>/<anything>",
        // use routeBuilder.MapGet("hello/{*name}")
        return context.Response.WriteAsync($"Hi, {name}!");
    });

    var routes = routeBuilder.Build();
    app.UseRouter(routes);
}
```

下表显示了具有给定 URL 的响应。

URI	响应
/package/create/3	Hello! Route values: [operation, create], [id, 3]
/package/track/-3	Hello! Route values: [operation, track], [id, -3]
/package/track/-3/	Hello! Route values: [operation, track], [id, -3]
/package/track/	<贯穿, 无任何匹配>
GET /hello/Joe	Hi, Joe!
POST /hello/Joe	<贯穿, 仅匹配 HTTP GET>
GET /hello/Joe/Smith	<贯穿, 无任何匹配>

如果要配置单个路由, 请在 `IRouter` 实例中调用 `app.UseRouter` 传入。无需调用 `RouteBuilder`。

框架可提供一组扩展方法, 用于创建如下路由:

- `MapRoute`
- `MapGet`
- `MapPost`
- `MapPut`
- `MapDelete`
- `MapVerb`

某些方法(如 `MapGet`)需要提供 `RequestDelegate`。路由匹配时, `RequestDelegate` 将用作路由处理器。此系列中的其他方法允许配置中间件管道, 它将被用作路由处理器。如果 Map 方法不接受处理器(如 `MapRoute`), 则它将使用 `DefaultHandler`。

`Map[Verb]` 方法将使用约束来将路由限制为方法名称中的 HTTP 谓词。有关示例, 请参阅 [MapGet](#) 和 [MapVerb](#)。

路由模板参考

如果路由找到匹配项, 大括号 (`{ }`) 内的令牌定义将绑定的路由参数。可在路由段中定义多个路由参数, 但必须由文本值隔开。例如, `{controller=Home}{action=Index}` 不是有效的路由, 因为 `{controller}` 和 `{action}` 之间没有文本值。这些路由参数必须具有名称, 且可能指定了其他属性。

路由参数以外的文本(例如 `{id}`)和路径分隔符 `/` 必须匹配 URL 中的文本。文本匹配区分大小写, 并且基于 URL 路径已解码的表示形式。若要匹配文本路由参数分隔符 `{` 或 `}`, 可通过重复该字符(`{}{}` 或 `{}{}`)对其进行转义。

尝试捕获具有可选文件扩展名的文件名的 URL 模式还有其他注意事项。例如, 使用模板

`files/{filename}.{ext?}`, 如果 `filename` 和 `ext` 同时存在, 将同时填充这两个值。如果 URL 中仅存在 `filename`, 则路由匹配, 因为尾随句点 `.` 是可选的。以下 URL 与此路由相匹配:

- `/files/myFile.txt`
- `/files/myFile.`
- `/files/myFile`

你可以使用 `*` 字符作为路由参数的前缀, 以绑定到 URI 的其余部分, 这称之为调用全方位参数。例如, `blog/*slug` 将匹配以 `/blog` 开头且其后带有任何值(将分配给 `slug` 路由值)的 URI。全方位参数还可以匹配空字符串。

路由参数可能具有指定的默认值, 方法是在参数名称后使用 `=` 隔开以指定默认值。例如, `{controller=Home}` 将 `Home` 定义为 `controller` 的默认值。如果参数的 URL 中不存在任何值, 则使用默认值。除默认值外, 路由参数可能是可选的(如 `id?` 中所示, 通过将 `?` 追加到参数名称末尾)。可选和“具有默认值”的区别在于具有默认值的路由参数始终会生成一个值, 而可选参数仅当提供值时才会具有一个值。

路由参数也可能具有约束, 必须匹配从 URL 中绑定的路由值。在路由参数后面添加一个冒号 `:` 和约束名称可指定路由参数上的内联约束。如果约束需要参数, 将以在约束名称后括在括号中的形式 `()` 提供。通过追加另一个冒号 `:` 和约束名称, 可指定多个内联约束。约束名称将传递给 `IInlineConstraintResolver` 服务, 以创建 `IRouteConstraint` 的实例, 用于处理 URL。例如, 路由模板 `blog/{article:minlength(10)}` 使用参数 `10` 指定 `minlength` 约束。有关路由约束的详细说明以及框架提供的约束列表, 请参阅 [route-constraint-reference](#)。

下表演示某些路由模板及其行为。

路由模板	匹配 URL 示例	说明
hello	/hello	仅匹配单个路径 /hello
{Page=Home}	/	匹配并将 Page 设置为 Home
{Page=Home}	/Contact	匹配并将 Page 设置为 Contact
{controller}/{action}/{id?}	/Products/List	映射到 Products 控制器和 List 操作
{controller}/{action}/{id?}	/Products/Details/123	映射到 Products 控制器和 Details 操作。将 id 设置为 123
{controller=Home}/{action=Index}/{id?}	/	映射到 Home 控制器和 Index 方法；将忽略 id。

使用模板通常是进行路由最简单的方法。还可在路由模板外指定约束和默认值。

提示：启用 [日志记录](#) 查看内置路由实现（如 `Route`）如何匹配请求。

路由约束参考

如果 `Route` 匹配传入 URL 的语法并将 URL 路径标记化为路由值，将执行路由约束。通常情况下，路由约束检查通过路由模板关联的路由值，并对该值是否可接受作出简单的是/否决策。某些路由约束使用路由值以外的数据来考虑是否可以路由请求。例如，`HttpMethodRouteConstraint` 可以根据其 HTTP 谓词接受或拒绝请求。

警告

避免使用输入验证约束，因为这样意味着无效输入将导致 404 错误（未找到），而不是含相应错误消息的 400 错误。路由约束应用来消除类似路由间的歧义，而不是验证特定路由的输入。

下表演示某些路由约束及其预期行为。

约束	示例	匹配项示例	说明
<code>int</code>	<code>{id:int}</code>	<code>123456789</code> , <code>-123456789</code>	匹配任何整数
<code>bool</code>	<code>{active:bool}</code>	<code>true</code> , <code>FALSE</code>	匹配 <code>true</code> 或 <code>false</code> (区分大小写)
<code>datetime</code>	<code>{dob:datetime}</code>	<code>2016-12-31</code> , <code>2016-12-31 7:32pm</code>	匹配有效的 <code>DateTime</code> 值 (位于固定区域性中 - 查看警告)
<code>decimal</code>	<code>{price:decimal}</code>	<code>49.99</code> , <code>-1,000.01</code>	匹配有效的 <code>decimal</code> 值 (位于固定区域性中 - 查看警告)
<code>double</code>	<code>{weight:double}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	匹配有效的 <code>double</code> 值(位于固定区域性中 - 查看警告)

约束	示例	匹配项示例	说明
<code>float</code>	<code>{weight:float}</code>	<code>1.234</code> , <code>-1,001.01e8</code>	匹配有效的 <code>float</code> 值(位于固定区域性中 - 查看警告)
<code>guid</code>	<code>{id:guid}</code>	<code>CD2C1638-1638-72D5-1638-DEADBEEF1638</code> , <code>{CD2C1638-1638-72D5-1638-DEADBEEF1638}</code>	匹配有效的 <code>Guid</code> 值
<code>long</code>	<code>{ticks:long}</code>	<code>123456789</code> , <code>-123456789</code>	匹配有效的 <code>long</code> 值
<code>minlength(value)</code>	<code>{username:minlength(4)}</code>	<code>Rick</code>	字符串必须至少为 4 个字符
<code>maxlength(value)</code>	<code>{filename:maxlength(8)}</code>	<code>Richard</code>	字符串不得超过 8 个字符
<code>length(length)</code>	<code>{filename:length(12)}</code>	<code>somefile.txt</code>	字符串必须正好为 12 个字符
<code>length(min,max)</code>	<code>{filename:length(8,16)}</code>	<code>somefile.txt</code>	字符串必须至少为 8 个字符, 且不得超过 16 个字符
<code>min(value)</code>	<code>{age:min(18)}</code>	<code>19</code>	整数值必须至少为 18
<code>max(value)</code>	<code>{age:max(120)}</code>	<code>91</code>	整数值不得超过 120
<code>range(min,max)</code>	<code>{age:range(18,120)}</code>	<code>91</code>	整数值必须至少为 18, 且不得超过 120
<code>alpha</code>	<code>{name:alpha}</code>	<code>Rick</code>	字符串必须由一个或多个字母字符(<code>a</code> - <code>z</code> , 区分大小写)组成
<code>regex(expression)</code>	<code>{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}</code>	<code>123-45-6789</code>	字符串必须匹配正则表达式(参见有关定义正则表达式的提示)
<code>required</code>	<code>{name:required}</code>	<code>Rick</code>	用于强制在 URL 生成过程中存在非参数值

警告

验证 URL 的路由约束可以转换为始终使用固定区域性的 CLR 类型(例如 `int` 或 `DateTime`), 它们假定 URL 不可本地化。框架提供的路由约束不会修改存储于路由值中的值。从 URL 中分析的所有路由值都将作为字符串进行存储。例如, [浮点数路由约束](#)会尝试将路由值转换为浮点数, 但转换后的值仅用来验证其是否可转换为浮点数。

正则表达式

ASP.NET Core 框架将向正则表达式构造函数添加

`RegexOptions.IgnoreCase` | `RegexOptions.Compiled` | `RegexOptions.CultureInvariant`。有关这些成员的介绍, 请参阅 [RegexOptions 枚举](#)。

正则表达式与路由和 C# 计算机语言使用的分隔符和令牌相似。必须对正则表达式令牌进行转义。例如，要在路由中使用正则表达式 `^\d{3}-\d{2}-\d{4}$`，需要如 C# 源文件中键入的 `\\" 一样的 \\" 字符，以转义 \\" 字符串转义字符(除非使用 verbatim 字符串文本)。`、`{`、`}`、`"["`和`"]"`字符需要成双进行转义，以转义路由参数分隔符字符。下表显示正则表达式和转义的版本。

表达式	说明
<code>^\d{3}-\d{2}-\d{4}\$</code>	正则表达式
<code>^\\d{3}-\\d{2}-\\d{4}\$</code>	已转义
<code>^[a-z]{2}\$</code>	正则表达式
<code>^[[a-z]]{{2}}\$</code>	已转义

路由中使用的正则表达式通常以 `^` 字符(匹配字符串的起始位置)开头，以 `$` 字符(匹配字符串的结束位置)结尾。`^` 和 `$` 字符可确保正则表达式匹配整个路由参数值。如果没有 `^` 和 `$` 字符，正则表达式将匹配字符串内的所有子字符串，这些字符串通常不是你想要的。下表显示部分示例，并说明它们为何匹配或未能匹配。

表达式	STRING	匹配	注释
<code>[a-z]{2}</code>	hello	是	子字符串匹配
<code>[a-z]{2}</code>	123abc456	是	子字符串匹配
<code>[a-z]{2}</code>	mz	是	匹配表达式
<code>[a-z]{2}</code>	MZ	是	不区分大小写
<code>^[a-z]{2}\$</code>	hello	否	参阅上述 <code>^</code> 和 <code>\$</code>
<code>^[a-z]{2}\$</code>	123abc456	否	参阅上述 <code>^</code> 和 <code>\$</code>

有关正则表达式语法的详细信息，请参阅 [.NET Framework 正则表达式](#)。

若要将参数限制为一组已知的可能值，可使用正则表达式。例如，`{action:regex('^(list|get|create)$')}` 仅将 `action` 路由值匹配到 `list`、`get` 或 `create`。如果传递到约束字典中，字符串“`^(list|get|create)$`”将等效。已传递到约束字典(不与模板内联)且不匹配任何已知约束的约束还将被视为正则表达式。

URL 生成参考

下面的示例演示如何在给定路由值字典和 `RouteCollection` 的情况下生成路由链接。

```

app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        { "operation", "create" },
        { "id", 123 }
    };

    var vpc = new VirtualPathContext(context, null, dictionary, "Track Package Route");
    var path = routes.GetVirtualPath(vpc).VirtualPath;

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync($"<a href='{path}'>Create Package 123</a><br/>");
});

```

上述示例末尾生成的 `VirtualPath` 为 `/package/create/123`。

`VirtualPathContext` 构造函数的第二个参数是环境值的集合。通过限制特定请求上下文中开发者必须指定的值数，环境值可提供便利。当前请求的当前路由值被视为链接生成的环境值。例如，在 ASP.NET MVC 应用中，如果处于 `HomeController` 的 `About` 操作中，则无需指定控制器路由值即可链接到 `Index` 操作（将使用 `Home` 环境值）。

URL 中从左至右，将忽略不匹配参数的环境值，如果显式提供的值替代环境值，该环境值也将被忽略。

显式提供但不匹配任何对象的值将添加到查询字符串。下表显示使用路由模板 `{controller}/{action}/{id?}` 时的结果。

环境值	显式值	结果
controller="Home"	action="About"	<code>/Home/About</code>
controller="Home"	controller="Order",action="About"	<code>/Order/About</code>
controller="Home",color="Red"	action="About"	<code>/Home/About</code>
controller="Home"	action="About",color="Red"	<code>/Home/About?color=Red</code>

如果路由具有不对应参数的默认值，且该值以显示方式提供，则它必须匹配默认值。例如：

```

routes.MapRoute("blog_route", "blog/{*slug}",
    defaults: new { controller = "Blog", action = "ReadPost" });

```

当提供控制器和操作的匹配值时，链接生成将只生成此路由的链接。

ASP.NET Core 中的 URL 重写中间件

2018/5/17 • 20 min to read • [Edit Online](#)

作者:Luke Latham 和 Mikael Mengistu

[查看或下载示例代码\(如何下载\)](#)

URL 重写是根据一个或多个预定义规则修改请求 URL 的行为。URL 重写会在资源位置和地址之间创建一个抽象，使位置和地址不紧密相连。在以下几种方案中，URL 重写很有价值：

- 暂时或永久移动或替换服务器资源，同时维护这些资源的稳定定位符
- 在不同应用或同一应用的不同区域中拆分请求处理
- 删减、添加或重新组织传入请求上的 URL 段
- 优化搜索引擎优化 (SEO) 的公共 URL
- 允许使用友好的公共 URL 帮助用户预测他们将通过链接找到的内容
- 将不安全请求重定向到安全终结点
- 阻止映像盗链

可通过多种方式定义更改 URL 的规则，包括正则表达式、Apache mod_rewrite 模块规则、IIS 重写模块规则以及使用自定义规则逻辑。本文档介绍 URL 重写并说明如何在 ASP.NET Core 应用中使用 URL 重写中间件。

注意

URL 重写可能会降低应用的性能。如果可行，应限制规则的数量和复杂度。

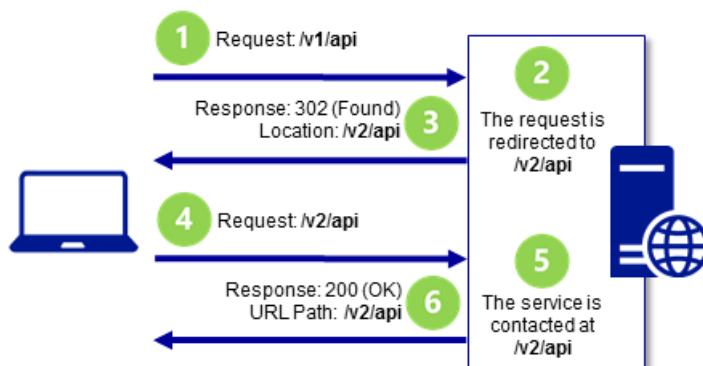
URL 重定向和 URL 重写

URL 重定向和 URL 重写之间的用词差异乍一看可能很细微，但这对于向客户端提供资源具有重要意义。

ASP.NET Core 的 URL 重写中间件能够满足两者的需求。

URL 重定向是客户端操作，指示客户端访问另一个地址的资源。这需要往返服务器。客户端对资源发出新请求时，返回客户端的重定向 URL 会出现在浏览器地址栏。

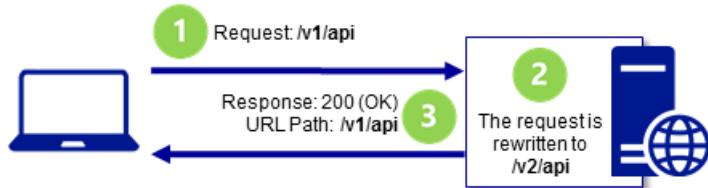
如果将 `/resource` 重定向到 `/different-resource`，客户端会请求 `/resource`。服务器通过指示重定向是临时还是永久的状态代码作出响应，表示客户端应该获取 `/different-resource` 处的资源。客户端在重定向 URL 处对资源执行新的请求。



将请求重定向到不同的 URL 时，可指示重定向是永久的还是临时的。如果资源有一个新的永久性 URL，并且你希望指示客户端所有未来的资源请求都使用新 URL，则使用“301 (永久移动)”状态代码。收到 301 状态代码时，客户端可能会缓存响应。如果重定向是临时的或一般会更改的，则使用“302 (已找到)”状态代码，以使客户端将来

不应存储和重用重定向 URL。有关详细信息，请参阅 [RFC 2616:状态代码定义](#)。

URL 重写是服务器端操作，提供来自不同资源地址的资源。重写 URL 不需要往返服务器。重写的 URL 不会返回客户端，也不会出现在浏览器地址栏。`/resource` 重写到 `/different-resource` 时，客户端会请求 `/resource`，并且服务器会在内部提取 `/different-resource` 处的资源。尽管客户端可能能够检索已重写 URL 处的资源，但是，客户端发出请求并收到响应时，并不会知道已重写 URL 处存在的资源。



URL 重写示例应用

可使用 [URL 重写示例应用](#) 了解 URL 重写中间件的功能。此应用使用重写和重定向规则，并显示重写或重定向 URL。

何时使用 URL 重写中间件

无法结合使用 [URL 重写模块](#) 和 Windows Server 上的 IIS、Apache Server 上的 [Apache mod_rewrite 模块](#)、[Nginx 上的 URL 重写](#)，或者应用托管在 [HTTP.sys 服务器](#)（以前称为 [WebListener](#)）上时，请使用 URL 重写中间件。在 IIS、Apache 或 Nginx 中使用基于服务器的 URL 重写技术的主要原因是，中间件不支持这些模块的全部功能，而且中间件的性能可能与模块的性能不匹配。但是，服务器模块的某些功能不适用于 ASP.NET Core 项目，例如 IIS 重写模块的 `IsFile` 和 `IsDirectory` 约束。在这些情况下，请改为使用中间件。

Package

要将中间件包含在项目中，请添加对 `Microsoft.AspNetCore.Rewrite` 包的引用。此功能适用于面向 ASP.NET Core 1.1 或更高版本的应用。

扩展和选项

通过使用扩展方法为每条规则创建 `RewriteOptions` 类的实例，建立 URL 重写和重定向规则。按所需的处理顺序链接多个规则。使用 `app.UseRewriter(options);` 将 `RewriteOptions` 添加到请求管道时，它会被传递到 URL 重写中间件。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}

```

URL 重定向

使用 `AddRedirect` 将请求重定向。第一个参数包含用于匹配传入 URL 路径的正则表达式。第二个参数是替换字符串。第三个参数(如有)指定状态代码。如不指定状态代码，则默认为“302 (已找到)”，指示资源暂时移动或替换。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}

```

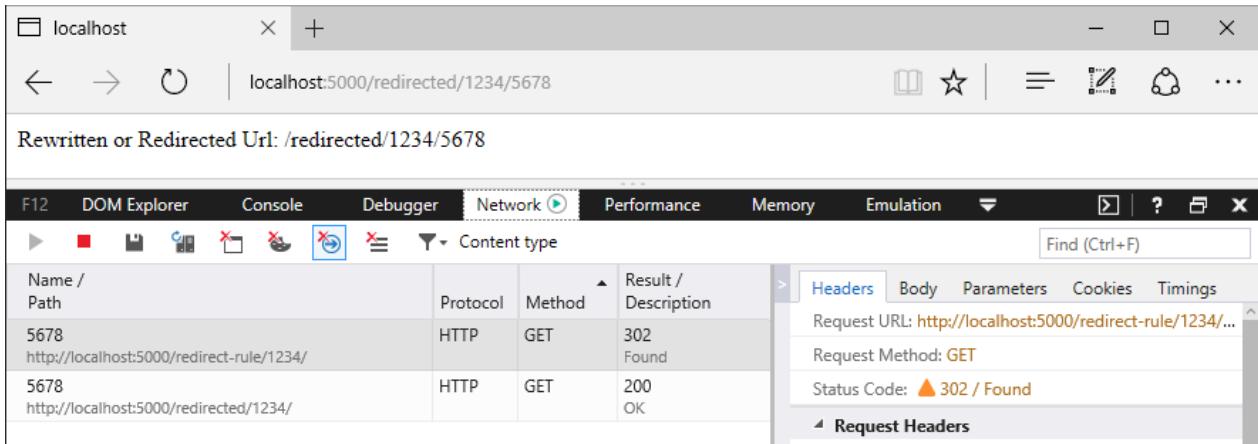
在启用了开发人员工具的浏览器中，向路径为 `/redirect-rule/1234/5678` 的示例应用发出请求。正则表达式匹配 `redirect-rule/(.*)` 上的请求路径，且该路径会被 `/redirected/1234/5678` 替代。重定向 URL 以“302 (已找到)”状态代码发回客户端。浏览器会在浏览器地址栏中出现的重定向 URL 上发出新请求。由于示例应用中的任何规则均不匹配重定向 URL，因此第二个请求会收到来自应用的“200 (正常)”响应，且响应正文会显示此重定向 URL。

重定向 URL 时，系统将向服务器进行一次往返。

警告

建立重定向规则时务必小心。系统会根据应用的每个请求(包括重定向后的请求)对重定向规则进行评估。很容易便会意外创建无限重定向循环。

原始请求: /redirect-rule/1234/5678



The screenshot shows the Microsoft Edge DevTools Network tab. The address bar displays 'localhost:5000/redirected/1234/5678'. The 'Rewritten or Redirected Url' section shows the original URL 'localhost:5000/redirect-rule/1234/5678' and the rewritten URL 'localhost:5000/redirected/1234/5678'. The Network table lists two entries:

Name / Path	Protocol	Method	Result / Description
5678 http://localhost:5000/redirect-rule/1234/	HTTP	GET	302 Found
5678 http://localhost:5000/redirected/1234/	HTTP	GET	200 OK

The 'Headers' panel on the right shows the request URL as 'http://localhost:5000/redirect-rule/1234/...' and the status code as '302 / Found'.

括号内的表达式部分称为“捕获组”。表达式的点 (.) 表示匹配任何字符。星号 (*) 表示零次或多次匹配前面的字符。因此，URL 的最后两个路径段 1234/5678 由捕获组 (.*) 捕获。在请求 URL 中提供的位于 redirect-rule/ 之后的任何值均由此单个捕获组捕获。

在替换字符串中，将捕获组注入带有美元符号 (\$)、后跟捕获序列号的字符串中。获取的第一个捕获组值为 \$1，第二个为 \$2，并且正则表达式中的其他捕获组值将依次继续排列。示例应用的重定向规则正则表达式中只有一个捕获组，因此替换字符串中只有一个注入组，即 \$1。如果应用此规则，URL 将变为 /redirected/1234/5678。

URL 重定向到安全的终结点

使用 `AddRedirectToHttps` 将 HTTP 请求重定向到采用 HTTPS (`https://`) 的相同主机和路径。如不提供状态代码，则中间件默认为“302(已找到)”。如不提供端口，则中间件默认为 `null`，这表示协议更改为 `https://` 且客户端访问端口 443 上的资源。该示例演示如何将状态代码设置为“301(永久移动)”并将端口更改为 5001。

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttps(301, 5001);

    app.UseRewriter(options);
}
```

使用 `AddRedirectToHttpsPermanent` 将不安全的请求重定向到采用安全 HTTPS 协议(端口 443 上的 `https://`)的相同主机和路径。中间件将状态代码设置为“301(永久移动)”。

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent();

    app.UseRewriter(options);
}
```

示例应用能够演示如何使用 `AddRedirectToHttps` 或 `AddRedirectToHttpsPermanent`。将扩展方法添加到 `RewriteOptions`。在任何 URL 上向应用发出不安全的请求。消除自签名证书不受信任的浏览器安全警告。

使用 `AddRedirectToHttps(301, 5001)` 的原始请求: /secure

The screenshot shows the F12 developer tools Network tab. The request URL is `http://localhost:5000/secure`. The response status code is `301 Moved Permanently`. The Headers section shows the `Location` header set to `https://localhost:5001/secure`.

Name / Path	Protocol	Method	Result / Description
secure http://localhost:5000/	HTTP	GET	301 Moved Permanently
secure https://localhost:5001/	HTTPS	GET	200 OK

使用 `AddRedirectToHttpsPermanent` 的原始请求: /secure

The screenshot shows the F12 developer tools Network tab. The request URL is `http://localhost:5000/secure`. The response status code is `301 Moved Permanently`. The Headers section shows the `Location` header set to `https://localhost/secure`.

Name / Path	Protocol	Method	Result / Description
secure http://localhost:5000/	HTTP	GET	301 Moved Permanently
secure https://localhost/	HTTPS	GET	200 OK

URL 重写

使用 `AddRewrite` 创建重写 URL 的规则。第一个参数包含用于匹配传入 URL 路径的正则表达式。第二个参数是替换字符串。第三个参数 `skipRemainingRules: {true|false}` 指示如果当前规则适用，中间件是否要跳过其他重写规则。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}

```

原始请求: /rewrite-rule/1234/5678

Name / Path	Protocol	Method	Result / Description
5678 http://localhost:5000/rewrite-rule/1234/	HTTP	GET	200 OK

Request URL: http://localhost:5000/rewrite-rule/1234/...
Request Method: GET
Status Code: 200 / OK

在正则表达式中首先注意到的是表达式开头的脱字号 (^)。这意味着匹配从 URL 路径的开头处开始。

在上述采用重定向规则 redirect-rule/(.*) 的示例中，正则表达式的开头没有任何脱字号；因此，成功匹配的路径中的任何字符都可以位于 redirect-rule/ 之前。

路径	匹配
/redirect-rule/1234/5678	是
/my-cool-redirect-rule/1234/5678	是
/anotherredirect-rule/1234/5678	是

重写规则 ^rewrite-rule/(\d+)/(\d+) 只能与以 rewrite-rule/ 开头的路径匹配。请注意以下重写规则和以上重定向规则之间的匹配差异。

路径	匹配
/rewrite-rule/1234/5678	是
/my-cool-rewrite-rule/1234/5678	否
/anotherrewrite-rule/1234/5678	否

在表达式的 `^rewrite-rule/` 部分之后，有两个捕获组 `(\d+)/(\d+)`。`\d` 表示与数字匹配。加号 `(+)` 表示与前面的一个或多个字符匹配。因此，URL 必须包含数字加正斜杠加另一个数字的形式。这些捕获组以 `$1` 和 `$2` 的形式注入重写 URL 中。重写规则替换字符串将捕获组放入查询字符串中。重写 `/rewrite-rule/1234/5678` 的请求路径，获取 `/rewritten?var1=1234&var2=5678` 处的资源。如果原始请求中存在查询字符串，则重写 URL 时会保留此字符串。

无需往返服务器来获取资源。如果资源存在，系统会提取资源并以“200 (正常)”状态代码返回给客户端。因为客户端不会被重定向，所以浏览器地址栏中的 URL 不会发生更改。对客户端来说，从未发生过 URL 重写操作。

注意

尽可能使用 `skipRemainingRules: true`，因为匹配规则代价高昂且会减少应用响应时间。对于最快的应用响应：

- 按照从最频繁匹配的规则到最不频繁匹配的规则排列重写规则。
- 如果出现匹配项且无需处理任何其他规则，则跳过剩余规则的处理。

Apache mod_rewrite

使用 `AddApacheModRewrite` 应用 Apache mod_rewrite 规则。请确保将规则文件与应用一起部署。有关 mod_rewrite 规则的详细信息和示例，请参阅 [Apache mod_rewrite](#)。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

`StreamReader` 用于读取 `ApacheModRewrite.txt` 规则文件中的规则。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用将请求从 `/apache-mod-rules-redirect/(.*)` 重定向到 `/redirected?id=$1`。响应状态代码为“302 (已找到)”。

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

原始请求: `/apache-mod-rules-redirect/1234`

The screenshot shows the Network tab of the F12 developer tools. The request URL is `localhost:5000/redirected?id=1234`. The response status is 302 Found, indicating a redirect. The redirected URL is `http://localhost:5000/apache-mod-rules-redirect/1234`. The response body shows the rewritten URL `/redirected?id=1234`.

Name / Path	Protocol	Method	Result / Description
1234 <code>http://localhost:5000/apache-mod-rules-redirect/</code>	HTTP	GET	302 Found
redirected?id=1234 <code>http://localhost:5000/</code>	HTTP	GET	200 OK

受支持的服务器变量

中间件支持下列 Apache mod_rewrite 服务器变量:

- `CONN_REMOTE_ADDR`
- `HTTP_ACCEPT`
- `HTTP_CONNECTION`
- `HTTP_COOKIE`
- `HTTP_FORWARDED`
- `HTTP_HOST`
- `HTTP_REFERER`
- `HTTP_USER_AGENT`
- `HTTPS`
- `IPV6`
- `QUERY_STRING`
- `REMOTE_ADDR`
- `REMOTE_PORT`
- `REQUEST_FILENAME`
- `REQUEST_METHOD`
- `REQUEST_SCHEME`
- `REQUEST_URI`
- `SCRIPT_FILENAME`
- `SERVER_ADDR`
- `SERVER_PORT`
- `SERVER_PROTOCOL`
- `TIME`
- `TIME_DAY`
- `TIME_HOUR`
- `TIME_MIN`
- `TIME_MON`
- `TIME_SEC`

- TIME_WDAY
- TIME_YEAR

IIS URL 重写模块规则

要使用适用于 IIS URL 重写模块的规则, 请使用 `AddIISUrlRewrite`。请确保将规则文件与应用一起部署。当 web.config 文件在 Windows Server IIS 上运行时, 请勿指示中间件使用该文件。使用 IIS 时, 应将这些规则存储在 web.config 之外, 避免与 IIS 重写模块发生冲突。有关 IIS URL 重写模块规则的详细信息和示例, 请参阅 [Using Url Rewrite Module 2.0](#)(使用 URL 重写模块 2.0)和 [URL Rewrite Module Configuration Reference](#)(URL 重写模块配置引用)。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

`StreamReader` 用于读取 `IISUrlRewrite.xml` 规则文件中的规则。

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

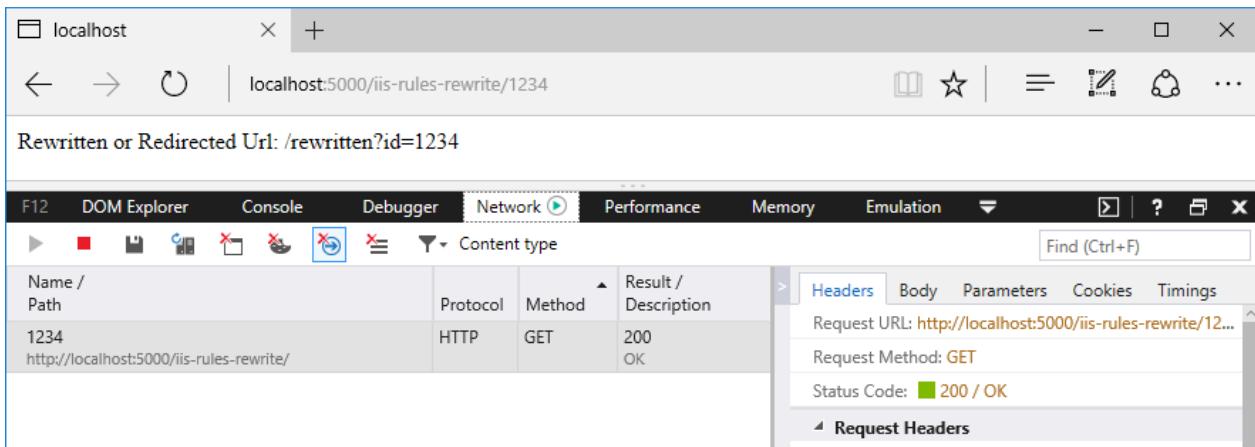
        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用将请求从 `/iis-rules-rewrite/(.*)` 重写为 `/rewritten?id=$1`。以“200 (正常)”状态代码作为响应发送到客户端。

```
<rewrite>
    <rules>
        <rule name="Rewrite segment to id querystring" stopProcessing="true">
            <match url="^iis-rules-rewrite/(.*)$" />
            <action type="Rewrite" url="rewritten?id={R:1}" appendQueryString="false"/>
        </rule>
    </rules>
</rewrite>
```

原始请求: `/iis-rules-rewrite/1234`



如果有配置了服务器级别规则(可对应用产生不利影响)的活动 IIS 重写模块，则可禁用应用的 IIS 重写模块。有关详细信息，请参阅[禁用 IIS 模块](#)。

不支持的功能

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

与 ASP.NET Core 2.x 一同发布的中间件不支持以下 IIS URL 重写模块功能：

- 出站规则
- 自定义服务器变量
- 通配符
- LogRewrittenUrl

受支持的服务器变量

中间件支持下列 IIS URL 重写模块服务器变量：

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

注意

也可通过 `PhysicalFileProvider` 获取 `IFileProvider`。这种方法可为重写规则文件的位置提供更大的灵活性。请确保将重写规则文件部署到所提供的路径的服务器中。

```
PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
```

基于方法的规则

使用 `Add(Action<RewriteContext> applyRule)` 在方法中实现自己的规则逻辑。为了在方法中使用 `HttpContext`, `RewriteContext` 会将其公开。`context.Result` 决定如何处理其他管道进程。

CONTEXT.RESULT	操作
<code>RuleResult.ContinueRules</code> (默认值)	继续应用规则
<code>RuleResult.EndResponse</code>	停止应用规则并发送响应
<code>RuleResult.SkipRemainingRules</code>	停止应用规则并将上下文发送给下一个中间件

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

示例应用演示了如何对以 `.xml` 结尾的路径的请求进行重定向。如果为 `/file.xml` 发出请求，则它将重定向到 `/xmlfiles/file.xml`。状态代码设置为“301 (永久移动)”。对于重定向，必须明确设置响应的状态代码；否则会返回“200 (正常)”状态代码，且在客户端上不会进行重定向。

```

public static void RedirectXMLRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because we're redirecting back to the same app, stop
    // processing if the request has already been redirected
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = StatusCodes.Status301MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}

```

原始请求: /file.xml

The screenshot shows the Microsoft Edge DevTools Network tab. The address bar indicates the request is to localhost:5000/xmlfiles/file.xml. The status bar shows "Rewritten or Redirected Url: /xmlfiles/file.xml". The Network tab details pane shows two entries:

Name / Path	Protocol	Method	Result / Description
file.xml http://localhost:5000/	HTTP	GET	301 Moved Permanently
file.xml http://localhost:5000/xmlfiles/	HTTP	GET	200 OK

Details pane content:

- Request URL: http://localhost:5000/file.xml
- Request Method: GET
- Status Code: 301 / Moved Permanently
- Request Headers

基于 IRule 的规则

使用 `Add(IRule)` 在派生自 `IRule` 的类中实现自己的规则逻辑。通过 `IRule` 为使用基于方法的规则方式提供更大的灵活性。派生类可能包含构造函数，你可在其中传入 `ApplyRule` 方法的参数。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXMLRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

检查示例应用中 `extension` 和 `newPath` 的参数值是否符合多个条件。`extension` 须包含一个值，并且该值必须是 `.png`、`.jpg` 或 `.gif`。如果 `newPath` 无效，则会引发 `ArgumentException`。如果为 `image.png` 发出请求，则它将重定向到 `/png-images/image.png`。如果为 `image.jpg` 发出请求，则它将重定向到 `/jpg-images/image.jpg`。状态代码设置为“301 (永久移动)”，`context.Result` 设置为停止处理规则并发送响应。

```

public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.png|jpg|gif$"))
        {
            throw new ArgumentException("Invalid extension", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"^/[A-Za-z0-9]+"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(_newPath))
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension, StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = StatusCodes.Status301MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}

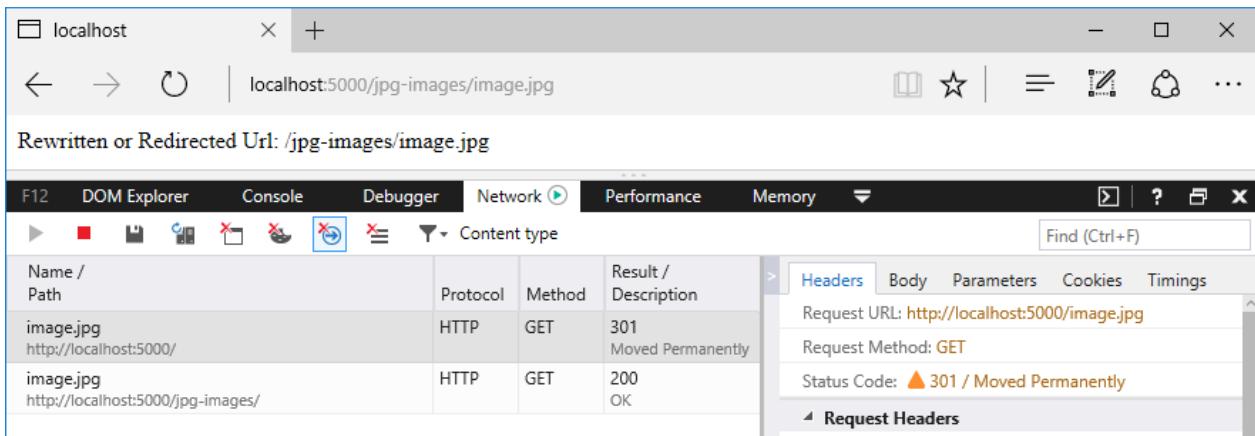
```

原始请求: /image.png

The screenshot shows a Microsoft Edge browser window. The address bar displays "localhost" and the URL "localhost:5000/png-images/image.png". Below the address bar, a status bar indicates "Rewritten or Redirected Url: /png-images/image.png". The browser interface includes standard navigation buttons (back, forward, search) and a toolbar with icons for refresh, stop, and other functions. At the bottom, the F12 developer tools Network tab is open, showing a table of network requests. The table has columns for Name / Path, Protocol, Method, Result / Description, Headers, Body, Parameters, Cookies, and Timings. One entry shows a 301 Moved Permanently response for the URL "image.png http://localhost:5000/". The Headers section of the tool shows the "Location" header set to "/png-images/image.png". The Body section shows the response content as "Status Code: ▲ 301 / Moved Permanently OK".

Name / Path	Protocol	Method	Result / Description	Headers	Body	Parameters	Cookies	Timings
image.png http://localhost:5000/	HTTP	GET	301 Moved Permanently					
image.png http://localhost:5000/png-images/	HTTP	GET	200 OK					

原始请求: /image.jpg



正则表达式示例

目标	正则表达式字符串和匹配示例	替换字符串和输出示例
将路径重写为查询字符串	<code>^path/(.*)/(.*)</code> <code>/path/abc/123</code>	<code>path?var1=\$1&var2=\$2</code> <code>/path?var1=abc&var2=123</code>
去掉尾部反斜杠	<code>(.*)/\$</code> <code>/path/</code>	<code>\$1</code> <code>/path</code>
强制添加尾部反斜杠	<code>(.*[^/])\$</code> <code>/path</code>	<code>\$1/</code> <code>/path/</code>
避免重写特定请求	<code>^(.*)(?<!\.axd)\$</code> 或 <code>^(?!.*\.axd\$)(.*)\$</code> 正确: <code>/resource.htm</code> 错误: <code>/resource.axd</code>	<code>rewritten/\$1</code> <code>/rewritten/resource.htm</code> <code>/resource.axd</code>
重新排列 URL 段	<code>path/(.*)/(.*)/(.*)</code> <code>path/1/2/3</code>	<code>path/\$3/\$2/\$1</code> <code>path/3/2/1</code>
替换 URL 段	<code>^(.*)/segment2/(.*)</code> <code>/segment1/segment2/segment3</code>	<code>\$1/replaced/\$2</code> <code>/segment1/replaced/segment3</code>

其他资源

- [应用程序启动](#)
- [中间件](#)
- [.NET 中的正则表达式](#)
- [正则表达式语言 - 快速参考](#)
- [Apache mod_rewrite](#)
- [使用 URL 重写模块 2.0\(适用于 IIS\)](#)
- [URL Rewrite Module Configuration Reference](#)(URL 重写模块配置引用)
- [IIS URL 重写模块论坛](#)
- [Keep a simple URL structure](#)(保持简单的 URL 结构)
- [10 URL Rewriting Tips and Tricks](#)(10 个 URL 重写提示和技巧)
- [To slash or not to slash](#)(用斜杠或不用斜杠)

在 ASP.NET Core 中使用多个环境

2018/5/14 • 6 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

ASP.NET Core 提供通过环境变量在运行时针对设置应用程序行为的支持。

[查看或下载示例代码\(如何下载\)](#)

环境

ASP.NET Core 在应用程序启动时读取环境变量 `ASPNETCORE_ENVIRONMENT`，并将该值存储在 `IHostingEnvironment.EnvironmentName` 中。`ASPNETCORE_ENVIRONMENT` 可设置为任意值，但框架支持三个值：[开发](#)、[暂存](#)和[生产](#)。如果未设置 `ASPNETCORE_ENVIRONMENT`，将默认为 `Production`。

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

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
    }

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

前面的代码：

- 当 `ASPNETCORE_ENVIRONMENT` 设置为 `Development` 时，调用 [UseDeveloperExceptionPage](#) 和 [UseBrowserLink](#)。
- 当 `ASPNETCORE_ENVIRONMENT` 的值设置为下列之一时，调用 [UseExceptionHandler](#)：
 - `Staging`
 - `Production`
 - `Staging_2`

[环境标记帮助程序](#)使用 `IHostingEnvironment.EnvironmentName` 的值用于包含或排除元素中的标记：

```
@page
@inject Microsoft.AspNetCore.Hosting.IHostingEnvironment hostingEnv
@model AboutModel
 @{
     ViewData["Title"] = "About";
 }
<h2>@ViewData["Title"]</h2>
<h3>@Model.Message</h3>

<p> ASPNETCORE_ENVIRONMENT = @hostingEnv.EnvironmentName</p>

<environment include="Development">
    <div>&lt;environment include="Development"&gt;</div>
</environment>
<environment exclude="Development">
    <div>&lt;environment exclude="Development"&gt;</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>
        &lt;environment include="Staging,Development,Staging_2"&gt;
        </div>
    </environment>
</environment>
```

注意：在 Windows 和 macOS 上，环境变量和值不区分大小写。默认情况下，Linux 环境变量和值要区分大小写。

开发

开发环境可以启用不应该在生产中公开的功能。例如，ASP.NET Core 模板在开发环境中启用了[开发人员异常页](#)。

本地计算机开发环境可以在项目的 Properties\launchSettings.json 文件中设置。在 launchSettings.json 中设置的环境值替代在系统环境中设置的值。

以下 JSON 显示 launchSettings.json 文件中的三个配置文件：

```
{  
  "iisSettings": {  
    "windowsAuthentication": false,  
    "anonymousAuthentication": true,  
    "iisExpress": {  
      "applicationUrl": "http://localhost:54339/",  
      "sslPort": 0  
    }  
  },  
  "profiles": {  
    "IIS Express": {  
      "commandName": "IISExpress",  
      "launchBrowser": true,  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    },  
    "WebApp1": {  
      "commandName": "Project",  
      "launchBrowser": true,  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Staging"  
      },  
      "applicationUrl": "http://localhost:54340/"  
    },  
    "Kestrel Staging": {  
      "commandName": "Project",  
      "launchBrowser": true,  
      "environmentVariables": {  
        "ASPNETCORE_My_Environment": "1",  
        "ASPNETCORE_DETAILEDERRORS": "1",  
        "ASPNETCORE_ENVIRONMENT": "Staging"  
      },  
      "applicationUrl": "http://localhost:51997/"  
    }  
  }  
}
```

注意

launchSettings.json 中的 `applicationUrl` 属性可指定服务器 URL 的列表。在列表中的 URL 之间使用分号：

```
"WebApplication1": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "applicationUrl": "https://localhost:5001;http://localhost:5000",  
  "environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
  }  
}
```

使用 `dotnet run` 启动应用程序时，将使用具有 `"commandName": "Project"` 的第一个配置文件。`commandName` 的值指定要启动的 Web 服务器。`commandName` 可以是以下之一：

- IIS Express
- IIS
- 项目(启动 Kestrel 的项目)

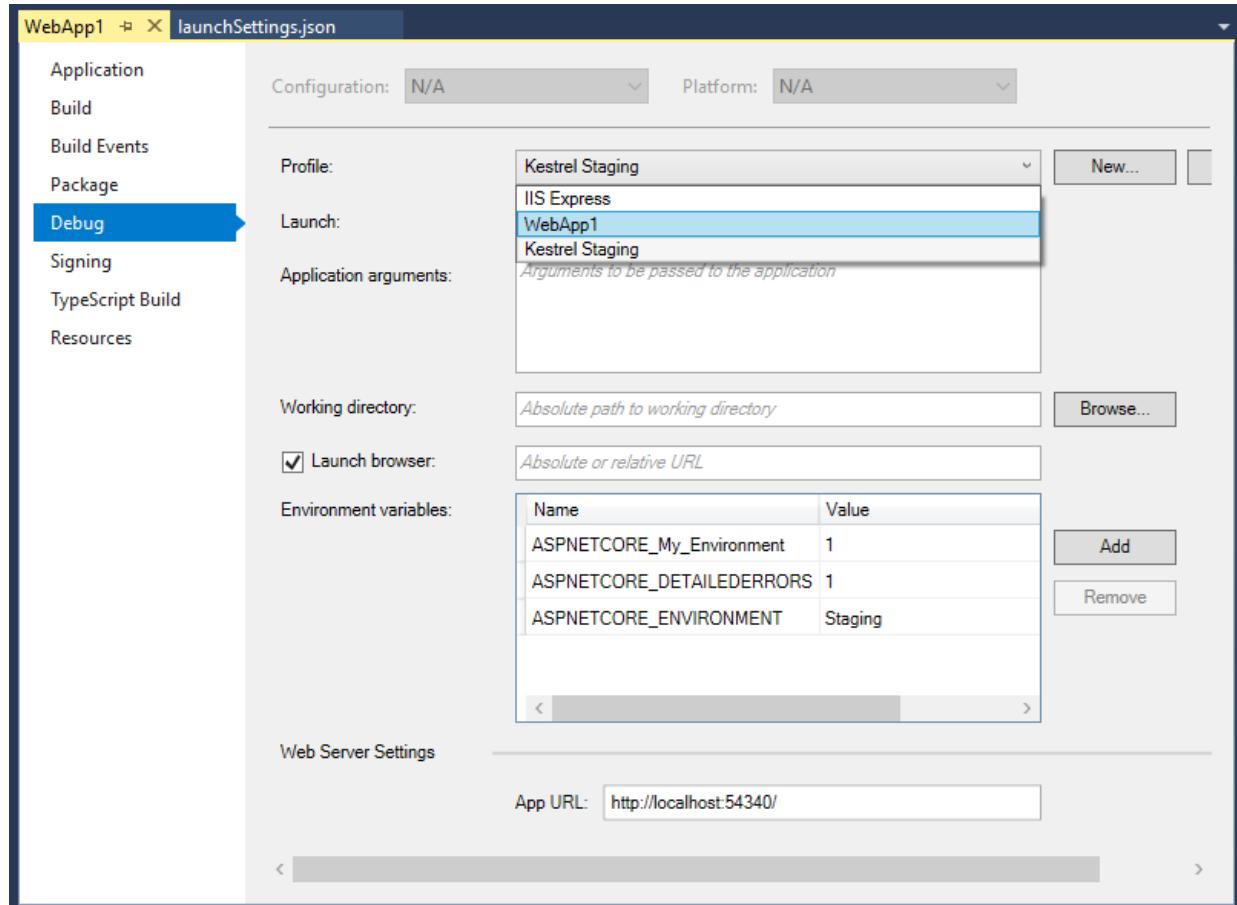
使用 `dotnet run` 启动应用时：

- 如果可用，读取 launchSettings.json。launchSettings.json 中的 `environmentVariables` 设置会替代环境变量。
- 此时显示承载环境。

以下输出显示了使用 `dotnet run` 启动的应用：

```
PS C:\Webs\WebApp1> dotnet run
Using launch settings from C:\Webs\WebApp1\Properties\launchSettings.json...
Hosting environment: Staging
Content root path: C:\Webs\WebApp1
Now listening on: http://localhost:54340
Application started. Press Ctrl+C to shut down.
```

Visual Studio“调试”选项卡提供 GUI 来编辑 `launchSettings.json` 文件：



在 Web 服务器重新启动之前，对项目配置文件所做的更改可能不会生效。必须重新启动 Kestrel 才能检测到对其环境所做的更改。

警告

`launchSettings.json` 不应存储机密。[机密管理器工具可用于存储本地开发的机密。](#)

生产

生产环境应配置为最大限度地提高安全性、性能和应用程序可靠性。不同于开发的一些通用设置包括：

- 缓存。
- 客户端资源被捆绑和缩小，并可能从 CDN 提供。
- 已禁用诊断错误页。
- 已启用友好错误页。
- 已启用生产记录和监视。例如，[Application Insights](#)。

设置环境

为测试设置特定环境通常很有用。如果未设置环境，默认值为 `Production`，这会禁用大多数调试功能。

设置环境的方法取决于操作系统。

Azure

对于 Azure 应用服务：

- 选择“应用程序设置”边栏选项卡。
- 将键和值添加到“应用设置”。

Windows

要为当前会话设置 `ASPNETCORE_ENVIRONMENT`，如果使用 `dotnet run` 启动该应用，则使用以下命令

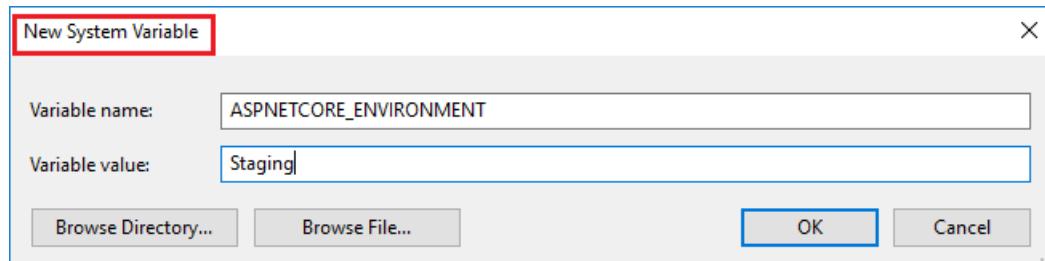
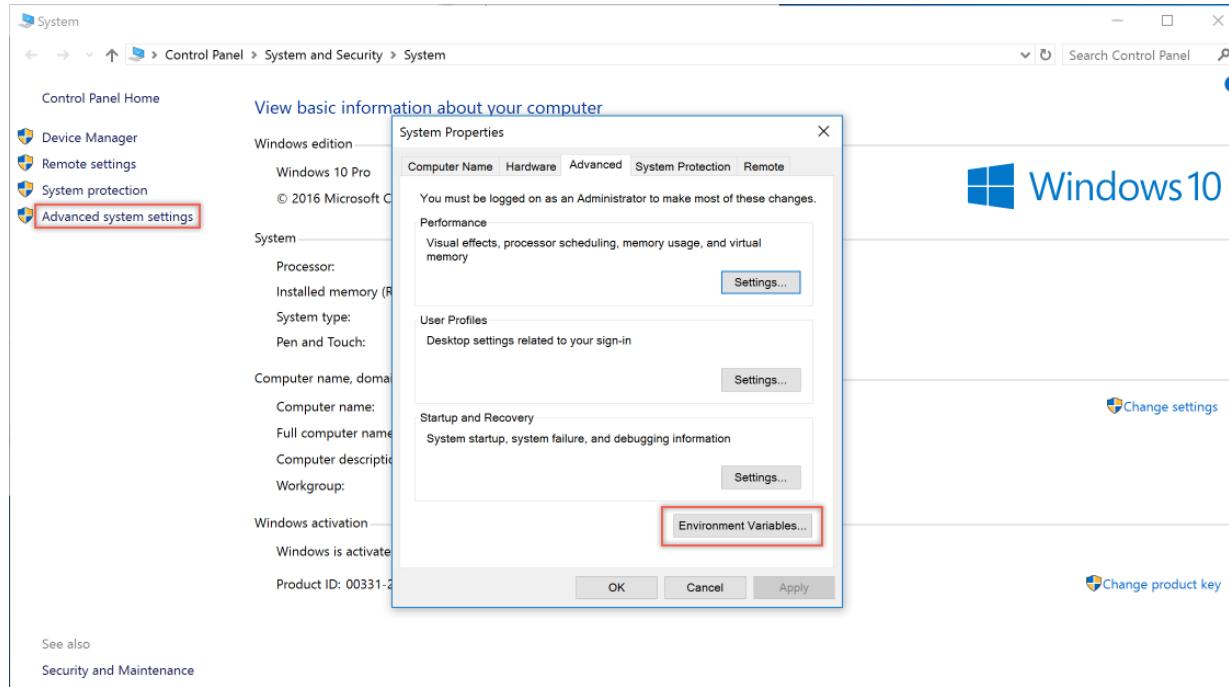
命令行

```
set ASPNETCORE_ENVIRONMENT=Development
```

PowerShell

```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

这些命令仅对当前窗口有效。窗口关闭时，`ASPNETCORE_ENVIRONMENT` 设置将恢复为默认设置或计算机值。若要在 Windows 上全局设置该值，请打开“控制面板”>“系统”>“高级系统设置”并添加或编辑 `ASPNETCORE_ENVIRONMENT` 值。



web.config

请参阅 [ASP.NET Core 模块配置参考](#) 主题的“设置环境变量”部分。

每个 IIS 应用程序池

若要为独立应用程序池中运行的单个应用设置环境变量(IIS 10.0+ 中支持此操作), 请参阅[环境变量 <environmentVariables>](#) 主题中的“AppCmd.exe 命令”部分。

macOS

设置 macOS 的当前环境可在运行应用程序时完成:

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

或在运行应用之前使用 `export` 对其进行设置。

```
export ASPNETCORE_ENVIRONMENT=Development
```

在 `.bashrc` 或 `.bash_profile` 文件中设置计算机级别环境变量。使用任何文本编辑器编辑文件并添加以下语句。

```
export ASPNETCORE_ENVIRONMENT=Development
```

Linux

对于 Linux 发行版, 请在命令行中使用 `export` 命令进行基于会话的变量设置, 并使用 `bash_profile` 文件进行计算机级别环境设置。

按环境配置

有关详细信息, 请参阅[按环境配置](#)。

基于环境的 Startup 类和方法

当 ASP.NET Core 应用启动时, [Startup](#) 类启动应用。如果类 `Startup{EnvironmentName}` 存在, 那么将为类 `EnvironmentName` 调用该类:

```
public class StartupDevelopment
{
    public StartupDevelopment(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

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

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

        if (env.IsProduction() || env.IsStaging())
        {
            throw new Exception("Not development.");
        }

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

注意: 调用 [WebHostBuilder.UseStartup](#) 替代配置节。

[Configure](#) 和 [ConfigureServices](#) 支持窗体 `Configure{EnvironmentName}` 和 `Configure{EnvironmentName}Services` 的环境特定版本:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

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

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

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

        if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
        {
            app.UseExceptionHandler("/Error");
        }

        app.UseStaticFiles();
        app.UseMvcWithDefaultRoute();
    }

    public void ConfigureStaging(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (!env.IsStaging())
        {
            throw new Exception("Not staging.");
        }

        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseMvcWithDefaultRoute();
    }
}
```

其他资源

- [应用程序启动](#)
- [配置](#)
- [IHostingEnvironment.EnvironmentName](#)

ASP.NET Core 中的配置

2018/5/17 • 18 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Mark Michaelis](#)、[Steve Smith](#)、[Daniel Roth](#) 和 [Luke Latham](#)

通过配置 API，可基于名称/值对列表来配置 ASP.NET Core Web 应用。在运行时从多个源读取配置。可将名称/值对分组到多级层次结构。

配置提供程序适用于：

- 文件格式(INI、JSON 和 XML)。
- 命令行参数。
- 环境变量。
- 内存中的 .NET 对象。
- 未加密的[机密管理器](#)存储。
- 加密的用户存储，如 [Azure Key Vault](#)。
- (已安装或已创建)自定义提供程序。

每个配置值映射到一个字符串键。可借助内置绑定支持，将设置反序列化为自定义 [POCO](#) 对象(一种具有属性的简单 .NET 类)。

选项模式使用选项类来表示相关设置的组。有关使用选项模式的详细信息，请参阅[选项](#)主题。

[查看或下载示例代码](#) ([如何下载](#))

JSON 配置

以下控制台应用使用 JSON 配置提供程序：

```

using System;
using System.IO;
// Requires NuGet package
// Microsoft.Extensions.Configuration.Json
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        Configuration = builder.Build();

        Console.WriteLine($"option1 = {Configuration["Option1"]}");
        Console.WriteLine($"option2 = {Configuration["option2"]}");
        Console.WriteLine(
            $"suboption1 = {Configuration["subsection:suboption1"]}");
        Console.WriteLine();

        Console.WriteLine("Wizards:");
        Console.Write($"{Configuration["wizards:0:Name"]}, ");
        Console.WriteLine($"{Configuration["wizards:0:Age"]}");
        Console.Write($"{Configuration["wizards:1:Name"]}, ");
        Console.WriteLine($"{Configuration["wizards:1:Age"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

该应用将读取和显示下列配置设置：

```
{
    "option1": "value1_from_json",
    "option2": 2,

    "subsection": {
        "suboption1": "subvalue1_from_json"
    },
    "wizards": [
        {
            "Name": "Gandalf",
            "Age": "1000"
        },
        {
            "Name": "Harry",
            "Age": "17"
        }
    ]
}
```

配置包含名称/值对的分层列表，其中节点由冒号 (:) 分隔。要检索某个值，请使用相应项的键访问 `Configuration` 索引器：

```
Console.WriteLine(
    $"suboption1 = {Configuration["subsection:suboption1"]});
```

要在 JSON 格式的配置源中使用数组，请在由冒号分隔的字符串中使用数组索引。以下示例获取上述 `wizards` 数组中第一个项的名称：

```
Console.WriteLine(Configuration["wizards:0:Name"]);
// Output: Gandalf
```

写入内置[配置提供程序](#)的名称/值对不是持久的。但是，可以创建一个自定义提供程序来保存值。请参阅[自定义配置提供程序](#)。

前面的示例使用配置索引器来读取值。要访问 `Startup` 外部的配置，请使用选项模式。有关详细信息，请参阅[选项主题](#)。

XML 配置

若要在 XML 格式的配置源中使用数组，请向每个元素提供一个 `name` 索引。使用该索引访问以下值：

```
<wizards>
  <wizard name="Gandalf">
    <age>1000</age>
  </wizard>
  <wizard name="Harry">
    <age>17</age>
  </wizard>
</wizards>
```

```
Console.WriteLine(Configuration["wizard:Harry:age"]);
// Output: 17
```

按环境配置

通常而言，配置设置因环境(如开发、测试和生产等)而异。ASP.NET Core 2.x 应用中的 `CreateDefaultBuilder` 扩展方法(或直接在 ASP.NET Core 1.x 应用中使用 `AddJsonFile` 和 `AddEnvironmentVariables`)添加了用于读取 JSON 文件和系统配置源的配置提供程序：

- `appsettings.json`
- `appsettings.<EnvironmentName>.json`
- 环境变量

ASP.NET Core 1.x 应用需要调用 `AddJsonFile` 和 `AddEnvironmentVariables`。

有关参数的说明，请参阅 `AddJsonFile`。仅 ASP.NET Core 1.1 及更高版本支持 `reloadOnChange`。

按指定配置源的顺序读取它们。在前面的代码中，最后才读取环境变量。在环境中设置的任意配置值将替换先前两个提供程序中设置的配置值。

请考虑使用以下 `appsettings.Staging.json` 文件：

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "System": "Information",  
            "Microsoft": "Information"  
        }  
    },  
    "MyConfig": "My Config Value for staging."  
}
```

环境设置为 `Staging` 时，以下 `Configure` 方法将读取 `MyConfig` 的值：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
    var myConfig = Configuration["MyConfig"];  
    // use myConfig  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
        app.UseBrowserLink();  
    }  
  
    if (env.IsProduction() || env.IsStaging())  
    {  
        app.UseExceptionHandler("/Error");  
    }  
  
    app.UseStaticFiles();  
    app.UseMvcWithDefaultRoute();  
}
```

环境通常设置为 `Development`、`Staging` 或 `Production`。有关详细信息，请参阅[使用多个环境](#)。

配置注意事项：

- 配置数据发生更改时，`IOptionsSnapshot` 可将其重载。
- 配置密钥不区分大小写。
- 请勿在配置提供程序代码或纯文本配置文件中存储密码或其他敏感数据。不要在开发或测试环境中使用生产机密。请在项目外部指定机密，避免将其意外提交到源代码存储库。详细了解[如何使用多个环境和在开发期间管理应用机密的安全存储](#)。
- 对于在环境变量中指定的分层配置值，冒号 (`:`) 可能不适用于所有平台。而所有平台均支持采用双下划线 (`__`)。
- 与配置 API 交互时，冒号 (`:`) 适用于所有平台。

内存中提供程序及绑定到 POCO 类

以下示例演示如何使用内存中提供程序及绑定到类：

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };
        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        var window = new MyWindow();
        // Bind requires NuGet package
        // Microsoft.Extensions.Configuration.Binder
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

public class MyWindow
{
    public int Height { get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}

```

配置值以字符串的形式返回，但绑定使对象的构造成为可能。通过绑定可检索 POCO 对象，甚至可检索整个对象图。

GetValue

以下示例演示 `GetValue<T>` 扩展方法：

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };
        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        // Show GetValue overload and set the default value to 80
        // Requires NuGet package "Microsoft.Extensions.Configuration.Binder"
        var left = Configuration.GetValue<int>("App:MainWindow:Left", 80);
        Console.WriteLine($"Left {left}");

        var window = new MyWindow();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

public class MyWindow
{
    public int Height { get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}

```

ConfigurationBinder 的 `GetValue<T>` 方法允许指定默认值(在此示例中为 80)。`GetValue<T>` 适用于简单方案，并不绑定到整个部分。`GetValue<T>` 从转换为特定类型的 `GetSection(key).Value` 中获取标量值。

绑定至对象图

可递归绑定类中的每个对象。请考虑使用以下 `AppSettings` 类：

```
public class AppSettings
{
    public Window Window { get; set; }
    public Connection Connection { get; set; }
    public Profile Profile { get; set; }
}

public class Window
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Connection
{
    public string Value { get; set; }
}

public class Profile
{
    public string Machine { get; set; }
}
```

以下示例绑定到 `AppSettings` 类：

```
using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var config = builder.Build();

        var appConfig = new AppSettings();
        config.GetSection("App").Bind(appConfig);

        Console.WriteLine($"Height {appConfig.Window.Height}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}
```

ASP.NET Core 1.1 及更高版本可使用 `Get<T>`，它适用于整个部分。使用 `Get<T>` 可能比 `Bind` 方便。以下代码演示了如何通过前述示例使用 `Get<T>`：

```
var appConfig = config.GetSection("App").Get<AppSettings>();
```

使用以下 `appsettings.json` 文件：

```
{  
    "App": {  
        "Profile": {  
            "Machine": "Rick"  
        },  
        "Connection": {  
            "Value": "connectionstring"  
        },  
        "Window": {  
            "Height": "11",  
            "Width": "11"  
        }  
    }  
}
```

该程序显示 `Height 11`。

可使用以下代码对配置进行单元测试：

```
[Fact]  
public void CanBindObjectTree()  
{  
    var dict = new Dictionary<string, string>  
    {  
        {"App:Profile:Machine", "Rick"},  
        {"App:Connection:Value", "connectionstring"},  
        {"App:Window:Height", "11"},  
        {"App:Window:Width", "11"}  
    };  
    var builder = new ConfigurationBuilder();  
    builder.AddInMemoryCollection(dict);  
    var config = builder.Build();  
  
    var settings = new AppSettings();  
    config.GetSection("App").Bind(settings);  
  
    Assert.Equal("Rick", settings.Profile.Machine);  
    Assert.Equal(11, settings.Window.Height);  
    Assert.Equal(11, settings.Window.Width);  
    Assert.Equal("connectionstring", settings.Connection.Value);  
}
```

创建 Entity Framework 自定义提供程序

本部分将创建一个使用 EF 从数据库读取名称/值对的基本配置提供程序。

定义 `ConfigurationValue` 实体，用于在数据库中存储配置值：

```
public class ConfigurationValue  
{  
    public string Id { get; set; }  
    public string Value { get; set; }  
}
```

添加 `ConfigurationContext` 以便存储和访问配置值：

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}
```

创建实现 [IConfigurationSource](#) 的类：

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigSource : IConfigurationSource
    {
        private readonly Action<DbContextOptionsBuilder> _optionsAction;

        public EFConfigSource(Action<DbContextOptionsBuilder> optionsAction)
        {
            _optionsAction = optionsAction;
        }

        public IConfigurationProvider Build(IConfigurationBuilder builder)
        {
            return new EFConfigProvider(_optionsAction);
        }
    }
}
```

通过从 [ConfigurationProvider](#) 继承来创建自定义配置提供程序。当数据库为空时，配置提供程序将对其进行初始化：

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigProvider : ConfigurationProvider
    {
        public EFConfigProvider(Action<DbContextOptionsBuilder> optionsAction)
        {
            OptionsAction = optionsAction;
        }

        Action<DbContextOptionsBuilder> OptionsAction { get; }

        // Load config data from EF DB.
        public override void Load()
        {
            var builder = new DbContextOptionsBuilder<ConfigurationContext>();
            OptionsAction(builder);

            using (var dbContext = new ConfigurationContext(builder.Options))
            {
                dbContext.Database.EnsureCreated();
                Data = !dbContext.Values.Any()
                    ? CreateAndSaveDefaultValues(dbContext)
                    : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
            }
        }

        private static IDictionary<string, string> CreateAndSaveDefaultValues(
            ConfigurationContext dbContext)
        {
            var configValues = new Dictionary<string, string>
            {
                { "key1", "value_from_ef_1" },
                { "key2", "value_from_ef_2" }
            };
            dbContext.Values.AddRange(configValues
                .Select(kvp => new ConfigurationValue { Id = kvp.Key, Value = kvp.Value })
                .ToArray());
            dbContext.SaveChanges();
            return configValues;
        }
    }
}

```

运行示例时将显示数据库中突出显示的值("value_from_ef_1"和"value_from_ef_2")。

可使用 `EFConfigSource` 扩展方法添加配置源：

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class EntityFrameworkExtensions
    {
        public static IConfigurationBuilder AddEntityFrameworkConfig(
            this IConfigurationBuilder builder, Action<DbContextOptionsBuilder> setup)
        {
            return builder.Add(new EFConfigSource(setup));
        }
    }
}

```

下面的代码演示如何使用自定义 `EFConfigProvider`：

```

using System;
using System.IO;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using CustomConfigurationProvider;

public static class Program
{
    public static void Main()
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var connectionStringConfig = builder.Build();

        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // Add "appsettings.json" to bootstrap EF config.
            .AddJsonFile("appsettings.json")
            // Add the EF configuration provider, which will override any
            // config made with the JSON provider.
            .AddEntityFrameworkConfig(options =>
                options.UseSqlServer(connectionStringConfig.GetConnectionString(
                    "DefaultConnection"))
            )
            .Build();

        Console.WriteLine("key1={0}", config["key1"]);
        Console.WriteLine("key2={0}", config["key2"]);
        Console.WriteLine("key3={0}", config["key3"]);
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

请注意，示例在 JSON 提供程序之后添加自定义 `EFConfigProvider`，因此数据库中的任何设置都将替代 appsettings.json 文件中的设置。

使用以下 appsettings.json 文件：

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Server=  
        (localdb)\\mssqllocaldb;Database=CustomConfigurationProvider;Trusted_Connection=True;MultipleActiveResultSets=t  
        rue"  
    },  
    "key1": "value_from_json_1",  
    "key2": "value_from_json_2",  
    "key3": "value_from_json_3"  
}
```

显示以下输出：

```
key1=value_from_ef_1  
key2=value_from_ef_2  
key3=value_from_json_3
```

CommandLine 配置提供程序

CommandLine 配置提供程序在运行时接收用于配置的命令行参数键值对。

[查看或下载命令行配置示例](#)

设置和使用 **CommandLine** 配置提供程序

- [基本配置](#)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

要激活命令行配置，请在 [ConfigurationBuilder](#) 的实例上调用 `AddCommandLine` 扩展方法：

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "MairaPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args);

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

运行代码将显示以下输出：

```

MachineName: MairaPC
Left: 1980

```

在命令行上传递参数键值对将更改 `Profile:MachineName` 和 `App:MainWindow:Left` 的值：

```
dotnet run Profile:MachineName=BartPC App:MainWindow:Left=1979
```

控制台窗口将显示：

```

MachineName: BartPC
Left: 1979

```

要使用命令行配置替代由其他配置提供程序提供的配置，请在 `ConfigurationBuilder` 上最后调用 `AddCommandLine`：

```

var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables()
    .AddCommandLine(args)
    .Build();

```

自变量

在命令行上传递的参数必须符合下表所示的两种格式之一：

参数格式	示例
单一参数:由等于号 (=) 分隔的键值对	key1=value
两个参数的序列:由空格分隔的键值对	/key1 value1

单一参数

值必须跟在等于号 (=) 之后。其值可以为 NULL(例如 mykey=)。

键可以具有前缀。

键前缀	示例
无前缀	key1=value1
单划线 (-)+	-key2=value2
双划线 (--)	--key3=value3
正斜杠 (/)	/key4=value4

+[交换映射](#) 中必须提供带有单划线前缀 (-) 的键, 如下所示。

示例命令:

```
dotnet run key1=value1 -key2=value2 --key3=value3 /key4=value4
```

注意:如果向配置提供程序提供的[交换映射](#)中没有 -key2, 则将引发 `FormatException`。

两个参数的序列

该值不可为 NULL, 且必须跟在由空格分隔的键之后。

键必须具有前缀。

键前缀	示例
单划线 (-)+	-key1 value1
双划线 (--)	--key2 value2
正斜杠 (/)	/key3 value3

+[交换映射](#) 中必须提供带有单划线前缀 (-) 的键, 如下所示。

示例命令:

```
dotnet run -key1 value1 --key2 value2 /key3 value3
```

注意:如果向配置提供程序提供的[交换映射](#)中没有 -key1, 则将引发 `FormatException`。

重复键

如果提供了重复的键, 将使用最后一个键值对。

交换映射

使用 `ConfigurationBuilder` 手动生成配置时，可将交换映射字典添加到 `AddCommandLine` 方法。交换映射支持键名替换逻辑。

当使用交换映射字典时，会检查字典中是否有与命令行参数提供的键匹配的键。如果在字典中找到命令行键，则传回字典值（替换键）以设置配置。对任何具有单划线（`-`）前缀的命令行键而言，交换映射都是必需的。

交换映射字典键规则：

- 交换必须以单划线（`-`）或双划线（`--`）开头。
- 交换映射字典不得包含重复键。

在以下示例中，`GetSwitchMappings` 方法允许命令行参数使用单划线（`-`）键前缀，并避免使用前导子键前缀。

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static Dictionary<string, string> GetSwitchMappings(
        IReadOnlyDictionary<string, string> configurationStrings)
    {
        return configurationStrings.Select(item =>
            new KeyValuePair<string, string>(
                "-" + item.Key.Substring(item.Key.LastIndexOf(':') + 1),
                item.Key))
            .ToDictionary(
                item => item.Key, item => item.Value);
    }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "RickPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args, GetSwitchMappings(dict));

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}
```

如果不提供命令行参数，则由提供给 `AddInMemoryCollection` 的字典来设置配置值。使用以下命令运行应用：

```
dotnet run
```

控制台窗口将显示：

```
MachineName: RickPC
Left: 1980
```

使用以下命令传递配置设置：

```
dotnet run /Profile:MachineName=DahliaPC /App:MainWindow:Left=1984
```

控制台窗口将显示：

```
MachineName: DahliaPC
Left: 1984
```

创建交换映射字典后，它将包含下表所示的数据：

键	“值”
-MachineName	Profile:MachineName
-Left	App:MainWindow:Left

要使用字典演示键交换，请运行下列命令：

```
dotnet run -MachineName=ChadPC -Left=1988
```

将交换命令行键。控制台窗口将显示 `Profile:MachineName` 和 `App:MainWindow:Left` 的配置值：

```
MachineName: ChadPC
Left: 1988
```

web.config 文件

在 IIS 或 IIS Express 中托管应用时，需要 `web.config` 文件。通过 `web.config` 中的设置，[ASP.NET Core Module](#) 可以启动应用并配置其他 IIS 设置和模块。如果 `web.config` 文件不存在，并且项目文件中包含 `<Project Sdk="Microsoft.NET.Sdk.Web">`，则发布项目时会在发布的输出（“发布”文件夹）中创建一个 `web.config` 文件。有关详细信息，请参阅 [使用 IIS 在 Windows 上托管 ASP.NET Core](#)。

在启动期间访问配置

若要在启动时访问 `ConfigureServices` 或 `Configure` 中的配置，请参阅[应用程序启动](#)主题中的示例。

从外部程序集添加配置

通过 [IHostingStartup](#) 实现，可在启动时从应用 `Startup` 类之外的外部程序集向应用添加增强功能。有关详细信息，请参阅[从外部程序集增强应用](#)。

在 Razor 页面或 MVC 视图中访问配置

若要访问 Razor 页面页或 MVC 视图中的配置设置，请为 [Microsoft.Extensions.Configuration 命名空间](#) 添加 `using` 指令（[C# 参考:using 指令](#)）并将 `IConfiguration` 注入页面或视图。

在 Razor 页面中：

```
@page
@model IndexModel

@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Access configuration in a Razor Pages page</h1>
    <p>Configuration["key"]:&nbsp;&nbsp;@Configuration["key"]</p>
</body>
</html>
```

在 MVC 视图中：

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index View</title>
</head>
<body>
    <h1>Access configuration in an MVC view</h1>
    <p>Configuration["key"]:&nbsp;&nbsp;@Configuration["key"]</p>
</body>
</html>
```

附加说明

- 调用 `ConfigureServices` 后才会设置依赖项注入 (DI)。
- 配置系统无法感知 DI。
- `IConfiguration` 具有两项专用化：
 - `IConfigurationRoot` 用于根节点。可以触发重载。
 - `IConfigurationSection` 表示配置值的一节。`GetSection` 和 `GetChildren` 方法返回 `IConfigurationSection`。
- 重新加载配置或要访问每个提供程序时，请使用 `IConfigurationRoot`。这两种情况都不常见。

其他资源

- [选项](#)
- [使用多个环境](#)
- [在开发期间安全存储应用机密](#)
- [ASP.NET Core 中的托管](#)
- [依赖关系注入](#)
- [Azure Key Vault 配置提供程序](#)

ASP.NET Core 中的配置

2018/5/17 • 18 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Mark Michaelis](#)、[Steve Smith](#)、[Daniel Roth](#) 和 [Luke Latham](#)

通过配置 API，可基于名称/值对列表来配置 ASP.NET Core Web 应用。在运行时从多个源读取配置。可将名称/值对分组到多级层次结构。

配置提供程序适用于：

- 文件格式 (INI、JSON 和 XML)。
- 命令行参数。
- 环境变量。
- 内存中的 .NET 对象。
- 未加密的[机密管理器](#)存储。
- 加密的用户存储，如 [Azure Key Vault](#)。
- (已安装或已创建的)自定义提供程序。

每个配置值映射到一个字符串键。可借助内置绑定支持，将设置反序列化为自定义 [POCO](#) 对象(一种具有属性的简单 .NET 类)。

选项模式使用选项类来表示相关设置的组。有关使用选项模式的详细信息，请参阅[选项](#)主题。

[查看或下载示例代码 \(如何下载\)](#)

JSON 配置

以下控制台应用使用 JSON 配置提供程序：

```

using System;
using System.IO;
// Requires NuGet package
// Microsoft.Extensions.Configuration.Json
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        Configuration = builder.Build();

        Console.WriteLine($"option1 = {Configuration["Option1"]}");
        Console.WriteLine($"option2 = {Configuration["option2"]}");
        Console.WriteLine(
            $"suboption1 = {Configuration["subsection:suboption1"]}");
        Console.WriteLine();

        Console.WriteLine("Wizards:");
        Console.Write($"{Configuration["wizards:0:Name"]}, ");
        Console.WriteLine($"{Configuration["wizards:0:Age"]}");
        Console.Write($"{Configuration["wizards:1:Name"]}, ");
        Console.WriteLine($"{Configuration["wizards:1:Age"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

该应用将读取和显示下列配置设置：

```
{
    "option1": "value1_from_json",
    "option2": 2,

    "subsection": {
        "suboption1": "subvalue1_from_json"
    },
    "wizards": [
        {
            "Name": "Gandalf",
            "Age": "1000"
        },
        {
            "Name": "Harry",
            "Age": "17"
        }
    ]
}
```

配置包含名称/值对的分层列表，其中节点由冒号 (:) 分隔。要检索某个值，请使用相应项的键访问 `Configuration` 索引器：

```
Console.WriteLine(
    $"suboption1 = {Configuration["subsection:suboption1"]});
```

要在 JSON 格式的配置源中使用数组，请在由冒号分隔的字符串中使用数组索引。以下示例获取上述 `wizards` 数组中第一个项的名称：

```
Console.WriteLine(Configuration["wizards:0:Name"]);
// Output: Gandalf
```

写入内置[配置提供程序](#)的名称/值对不是持久的。但是，可以创建一个自定义提供程序来保存值。请参阅[自定义配置提供程序](#)。

前面的示例使用配置索引器来读取值。要访问 `Startup` 外部的配置，请使用选项模式。有关详细信息，请参阅[选项](#)主题。

XML 配置

若要在 XML 格式的配置源中使用数组，请向每个元素提供一个 `name` 索引。使用该索引访问以下值：

```
<wizards>
  <wizard name="Gandalf">
    <age>1000</age>
  </wizard>
  <wizard name="Harry">
    <age>17</age>
  </wizard>
</wizards>
```

```
Console.WriteLine(Configuration["wizard:Harry:age"]);
// Output: 17
```

按环境配置

通常而言，配置设置因环境（如开发、测试和生产等）而异。ASP.NET Core 2.x 应用中的 `CreateDefaultBuilder` 扩展方法（或直接在 ASP.NET Core 1.x 应用中使用 `AddJsonFile` 和 `AddEnvironmentVariables`）添加了用于读取 JSON 文件和系统配置源的配置提供程序：

- `appsettings.json`
- `appsettings.<EnvironmentName>.json`
- 环境变量

ASP.NET Core 1.x 应用需要调用 `AddJsonFile` 和 `AddEnvironmentVariables`。

有关参数的说明，请参阅 [AddJsonFile](#)。仅 ASP.NET Core 1.1 及更高版本支持 `reloadOnChange`。

按指定配置源的顺序读取它们。在前面的代码中，最后才读取环境变量。在环境中设置的任意配置值将替换先前两个提供程序中设置的配置值。

请考虑使用以下 `appsettings.Staging.json` 文件：

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "System": "Information",  
            "Microsoft": "Information"  
        }  
    },  
    "MyConfig": "My Config Value for staging."  
}
```

环境设置为 `Staging` 时，以下 `Configure` 方法将读取 `MyConfig` 的值：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
    var myConfig = Configuration["MyConfig"];  
    // use myConfig  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
        app.UseBrowserLink();  
    }  
  
    if (env.IsProduction() || env.IsStaging())  
    {  
        app.UseExceptionHandler("/Error");  
    }  
  
    app.UseStaticFiles();  
    app.UseMvcWithDefaultRoute();  
}
```

环境通常设置为 `Development`、`Staging` 或 `Production`。有关详细信息，请参阅[使用多个环境](#)。

配置注意事项：

- 配置数据发生更改时，`IOptionsSnapshot` 可将其重载。
- 配置密钥不区分大小写。
- 请勿在配置提供程序代码或纯文本配置文件中存储密码或其他敏感数据。不要在开发或测试环境中使用生产机密。请在项目外部指定机密，避免将其意外提交到源代码存储库。详细了解[如何使用多个环境和在开发期间管理应用机密的安全存储](#)。
- 对于在环境变量中指定的分层配置值，冒号 (`:`) 可能不适用于所有平台。而所有平台均支持采用双下划线 (`__`)。
- 与配置 API 交互时，冒号 (`:`) 适用于所有平台。

内存中提供程序及绑定到 POCO 类

以下示例演示如何使用内存中提供程序及绑定到类：

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };
        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict);

        Configuration = builder.Build();

        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        var window = new MyWindow();
        // Bind requires NuGet package
        // Microsoft.Extensions.Configuration.Binder
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
        Console.WriteLine();

        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

public class MyWindow
{
    public int Height { get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}

```

配置值以字符串的形式返回，但绑定使对象的构造成为可能。通过绑定可检索 POCO 对象，甚至可检索整个对象图。

GetValue

以下示例演示 [GetValue<T>](#) 扩展方法：

```

using System;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "Rick"},
            {"App:MainWindow:Height", "11"},
            {"App:MainWindow:Width", "11"},
            {"App:MainWindow:Top", "11"},
            {"App:MainWindow:Left", "11"}
        };
    }

    var builder = new ConfigurationBuilder();
    builder.AddInMemoryCollection(dict);

    Configuration = builder.Build();

    Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

    // Show GetValue overload and set the default value to 80
    // Requires NuGet package "Microsoft.Extensions.Configuration.Binder"
    var left = Configuration.GetValue<int>("App:MainWindow:Left", 80);
    Console.WriteLine($"Left {left}");

    var window = new MyWindow();
    Configuration.GetSection("App:MainWindow").Bind(window);
    Console.WriteLine($"Left {window.Left}");
    Console.WriteLine();

    Console.WriteLine("Press a key...");
    Console.ReadKey();
}
}

public class MyWindow
{
    public int Height { get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}

```

ConfigurationBinder 的 `GetValue<T>` 方法允许指定默认值(在此示例中为 80)。`GetValue<T>` 适用于简单方案，并不绑定到整个部分。`GetValue<T>` 从转换为特定类型的 `GetSection(key).Value` 中获取标量值。

绑定至对象图

可递归绑定类中的每个对象。请考虑使用以下 `AppSettings` 类：

```

public class AppSettings
{
    public Window Window { get; set; }
    public Connection Connection { get; set; }
    public Profile Profile { get; set; }
}

public class Window
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Connection
{
    public string Value { get; set; }
}

public class Profile
{
    public string Machine { get; set; }
}

```

以下示例绑定到 `AppSettings` 类：

```

using System;
using System.IO;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static void Main(string[] args = null)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var config = builder.Build();

        var appConfig = new AppSettings();
        config.GetSection("App").Bind(appConfig);

        Console.WriteLine($"Height {appConfig.Window.Height}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

ASP.NET Core 1.1 及更高版本可使用 `Get<T>`，它适用于整个部分。使用 `Get<T>` 可能比 `Bind` 方便。以下代码演示了如何通过前述示例使用 `Get<T>`：

```
var appConfig = config.GetSection("App").Get<AppSettings>();
```

使用以下 `appsettings.json` 文件：

```
{  
    "App": {  
        "Profile": {  
            "Machine": "Rick"  
        },  
        "Connection": {  
            "Value": "connectionstring"  
        },  
        "Window": {  
            "Height": "11",  
            "Width": "11"  
        }  
    }  
}
```

该程序显示 `Height 11`。

可使用以下代码对配置进行单元测试：

```
[Fact]  
public void CanBindObjectTree()  
{  
    var dict = new Dictionary<string, string>  
    {  
        {"App:Profile:Machine", "Rick"},  
        {"App:Connection:Value", "connectionstring"},  
        {"App:Window:Height", "11"},  
        {"App:Window:Width", "11"}  
    };  
    var builder = new ConfigurationBuilder();  
    builder.AddInMemoryCollection(dict);  
    var config = builder.Build();  
  
    var settings = new AppSettings();  
    config.GetSection("App").Bind(settings);  
  
    Assert.Equal("Rick", settings.Profile.Machine);  
    Assert.Equal(11, settings.Window.Height);  
    Assert.Equal(11, settings.Window.Width);  
    Assert.Equal("connectionstring", settings.Connection.Value);  
}
```

创建 Entity Framework 自定义提供程序

本部分将创建一个使用 EF 从数据库读取名称/值对的基本配置提供程序。

定义 `ConfigurationValue` 实体，用于在数据库中存储配置值：

```
public class ConfigurationValue  
{  
    public string Id { get; set; }  
    public string Value { get; set; }  
}
```

添加 `ConfigurationContext` 以便存储和访问配置值：

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}
```

创建实现 [IConfigurationSource](#) 的类：

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigSource : IConfigurationSource
    {
        private readonly Action<DbContextOptionsBuilder> _optionsAction;

        public EFConfigSource(Action<DbContextOptionsBuilder> optionsAction)
        {
            _optionsAction = optionsAction;
        }

        public IConfigurationProvider Build(IConfigurationBuilder builder)
        {
            return new EFConfigProvider(_optionsAction);
        }
    }
}
```

通过从 [ConfigurationProvider](#) 继承来创建自定义配置提供程序。当数据库为空时，配置提供程序将对其进行初始化：

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public class EFConfigProvider : ConfigurationProvider
    {
        public EFConfigProvider(Action<DbContextOptionsBuilder> optionsAction)
        {
            OptionsAction = optionsAction;
        }

        Action<DbContextOptionsBuilder> OptionsAction { get; }

        // Load config data from EF DB.
        public override void Load()
        {
            var builder = new DbContextOptionsBuilder<ConfigurationContext>();
            OptionsAction(builder);

            using (var dbContext = new ConfigurationContext(builder.Options))
            {
                dbContext.Database.EnsureCreated();
                Data = !dbContext.Values.Any()
                    ? CreateAndSaveDefaultValues(dbContext)
                    : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
            }
        }

        private static IDictionary<string, string> CreateAndSaveDefaultValues(
            ConfigurationContext dbContext)
        {
            var configValues = new Dictionary<string, string>
            {
                { "key1", "value_from_ef_1" },
                { "key2", "value_from_ef_2" }
            };
            dbContext.Values.AddRange(configValues
                .Select(kvp => new ConfigurationValue { Id = kvp.Key, Value = kvp.Value })
                .ToArray());
            dbContext.SaveChanges();
            return configValues;
        }
    }
}

```

运行示例时将显示数据库中突出显示的值("value_from_ef_1"和"value_from_ef_2")。

可使用 `EFConfigSource` 扩展方法添加配置源：

```

using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;

namespace CustomConfigurationProvider
{
    public static class EntityFrameworkExtensions
    {
        public static IConfigurationBuilder AddEntityFrameworkConfig(
            this IConfigurationBuilder builder, Action<DbContextOptionsBuilder> setup)
        {
            return builder.Add(new EFConfigSource(setup));
        }
    }
}

```

下面的代码演示如何使用自定义 `EFConfigProvider`：

```

using System;
using System.IO;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using CustomConfigurationProvider;

public static class Program
{
    public static void Main()
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("appsettings.json");

        var connectionStringConfig = builder.Build();

        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // Add "appsettings.json" to bootstrap EF config.
            .AddJsonFile("appsettings.json")
            // Add the EF configuration provider, which will override any
            // config made with the JSON provider.
            .AddEntityFrameworkConfig(options =>
                options.UseSqlServer(connectionStringConfig.GetConnectionString(
                    "DefaultConnection"))
            )
            .Build();

        Console.WriteLine("key1={0}", config["key1"]);
        Console.WriteLine("key2={0}", config["key2"]);
        Console.WriteLine("key3={0}", config["key3"]);
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

请注意，示例在 JSON 提供程序之后添加自定义 `EFConfigProvider`，因此数据库中的任何设置都将替代 `appsettings.json` 文件中的设置。

使用以下 `appsettings.json` 文件：

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=CustomConfigurationProvider;Trusted_Connection=True;MultipleActiveResultSets=true"  
  },  
  "key1": "value_from_json_1",  
  "key2": "value_from_json_2",  
  "key3": "value_from_json_3"  
}
```

显示以下输出：

```
key1=value_from_ef_1  
key2=value_from_ef_2  
key3=value_from_json_3
```

CommandLine 配置提供程序

CommandLine 配置提供程序在运行时接收用于配置的命令行参数键值对。

[查看或下载命令行配置示例](#)

设置和使用 CommandLine 配置提供程序

- [基本配置](#)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

要激活命令行配置，请在 `ConfigurationBuilder` 的实例上调用 `AddCommandLine` 扩展方法：

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "MairaPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args);

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}

```

运行代码将显示以下输出：

```

MachineName: MairaPC
Left: 1980

```

在命令行上传递参数键值对将更改 `Profile:MachineName` 和 `App:MainWindow:Left` 的值：

```
dotnet run Profile:MachineName=BartPC App:MainWindow:Left=1979
```

控制台窗口将显示：

```

MachineName: BartPC
Left: 1979

```

要使用命令行配置替代由其他配置提供程序提供的配置，请在 `ConfigurationBuilder` 上最后调用 `AddCommandLine`：

```

var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables()
    .AddCommandLine(args)
    .Build();

```

自变量

在命令行上传递的参数必须符合下表所示的两种格式之一：

参数格式	示例
单一参数：由等于号 (=) 分隔的键值对	key1=value
两个参数的序列：由空格分隔的键值对	/key1 value1

单一参数

值必须跟在等于号 (=) 之后。其值可以为 NULL (例如 mykey=)。

键可以具有前缀。

键前缀	示例
无前缀	key1=value1
单划线 (-)+	-key2=value2
双划线 (--)	--key3=value3
正斜杠 (/)	/key4=value4

[†]交换映射中必须提供带有单划线前缀 (-) 的键，如下所示。

示例命令：

```
dotnet run key1=value1 -key2=value2 --key3=value3 /key4=value4
```

注意：如果向配置提供程序提供的交换映射中没有 -key2，则将引发 FormatException。

两个参数的序列

该值不可为 NULL，且必须跟在由空格分隔的键之后。

键必须具有前缀。

键前缀	示例
单划线 (-)+	-key1 value1
双划线 (--)	--key2 value2
正斜杠 (/)	/key3 value3

[†]交换映射中必须提供带有单划线前缀 (-) 的键，如下所示。

示例命令：

```
dotnet run -key1 value1 --key2 value2 /key3 value3
```

注意：如果向配置提供程序提供的交换映射中没有 -key1，则将引发 FormatException。

重复键

如果提供了重复的键，将使用最后一个键值对。

交换映射

使用 `ConfigurationBuilder` 手动生成配置时，可将交换映射字典添加到 `AddCommandLine` 方法。交换映射支持键名替换逻辑。

当使用交换映射字典时，会检查字典中是否有与命令行参数提供的键匹配的键。如果在字典中找到命令行键，则传回字典值（替换键）以设置配置。对任何具有单划线（-）前缀的命令行键而言，交换映射都是必需的。

交换映射字典键规则：

- 交换必须以单划线（-）或双划线（--）开头。
- 交换映射字典不得包含重复键。

在以下示例中，`GetSwitchMappings` 方法允许命令行参数使用单划线（-）键前缀，并避免使用前导子键前缀。

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static IConfiguration Configuration { get; set; }

    public static Dictionary<string, string> GetSwitchMappings(
        IReadOnlyDictionary<string, string> configurationStrings)
    {
        return configurationStrings.Select(item =>
            new KeyValuePair<string, string>(
                "-" + item.Key.Substring(item.Key.LastIndexOf(':') + 1),
                item.Key))
            .ToDictionary(
                item => item.Key, item => item.Value);
    }

    public static void Main(string[] args = null)
    {
        var dict = new Dictionary<string, string>
        {
            {"Profile:MachineName", "RickPC"},
            {"App:MainWindow:Left", "1980"}
        };

        var builder = new ConfigurationBuilder();

        builder.AddInMemoryCollection(dict)
            .AddCommandLine(args, GetSwitchMappings(dict));

        Configuration = builder.Build();

        Console.WriteLine($"MachineName: {Configuration["Profile:MachineName"]}");
        Console.WriteLine($"Left: {Configuration["App:MainWindow:Left"]}");
        Console.WriteLine();

        Console.WriteLine("Press a key...");
        Console.ReadKey();
    }
}
```

如果不提供命令行参数，则由提供给 `AddInMemoryCollection` 的字典来设置配置值。使用以下命令运行应

用:

```
dotnet run
```

控制台窗口将显示:

```
MachineName: RickPC  
Left: 1980
```

使用以下命令传递配置设置:

```
dotnet run /Profile:MachineName=DahliaPC /App:MainWindow:Left=1984
```

控制台窗口将显示:

```
MachineName: DahliaPC  
Left: 1984
```

创建交换映射字典后, 它将包含下表所示的数据:

键	“值”
-MachineName	Profile:MachineName
-Left	App:MainWindow:Left

要使用字典演示键交换, 请运行下列命令:

```
dotnet run -MachineName=ChadPC -Left=1988
```

将交换命令行键。控制台窗口将显示 `Profile:MachineName` 和 `App:MainWindow:Left` 的配置值:

```
MachineName: ChadPC  
Left: 1988
```

web.config 文件

在 IIS 或 IIS Express 中托管应用时, 需要 web.config 文件。通过 web.config 中的设置, [ASP.NET Core Module](#) 可以启动应用并配置其他 IIS 设置和模块。如果 `web.config` 文件不存在, 并且项目文件中包含 `<Project Sdk="Microsoft.NET.Sdk.Web">`, 则发布项目时会在发布的输出("发布"文件夹)中创建一个 `web.config` 文件。有关详细信息, 请参阅 [使用 IIS 在 Windows 上托管 ASP.NET Core](#)。

在启动期间访问配置

若要在启动时访问 `ConfigureServices` 或 `Configure` 中的配置, 请参阅[应用程序启动](#)主题中的示例。

从外部程序集添加配置

通过 [IHostingStartup](#) 实现, 可在启动时从应用 `Startup` 类之外的外部程序集向应用添加增强功能。有关详细信息, 请参阅[从外部程序集增强应用](#)。

在 Razor 页面或 MVC 视图中访问配置

若要访问 Razor 页面页或 MVC 视图中的配置设置, 请为 [Microsoft.Extensions.Configuration 命名空间](#)添加 [using 指令\(C# 参考:using 指令\)](#) 并将 [IConfiguration](#) 注入页面或视图。

在 Razor 页面页中:

```
@page
@model IndexModel

@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Access configuration in a Razor Pages page</h1>
    <p>Configuration["key"]:<code> @Configuration["key"]</code></p>
</body>
</html>
```

在 MVC 视图中:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index View</title>
</head>
<body>
    <h1>Access configuration in an MVC view</h1>
    <p>Configuration["key"]:<code> @Configuration["key"]</code></p>
</body>
</html>
```

附加说明

- 调用 `ConfigureServices` 后才会设置依赖项注入 (DI)。
- 配置系统无法感知 DI。
- `IConfiguration` 具有两项专用化:
 - `IConfigurationRoot` 用于根节点。可以触发重载。
 - `IConfigurationSection` 表示配置值的一节。`GetSection` 和 `GetChildren` 方法返回 `IConfigurationSection`。
 - 重新加载配置或要访问每个提供程序时, 请使用 `IConfigurationRoot`。这两种情况都不常见。

其他资源

- [选项](#)
- [使用多个环境](#)
- [在开发期间安全存储应用机密](#)
- [ASP.NET Core 中的托管](#)
- [依赖关系注入](#)

- Azure Key Vault 配置提供程序

ASP.NET Core 中的选项模式

2018/5/14 • 11 min to read • [Edit Online](#)

作者: [Luke Latham](#)

选项模式使用类来表示相关设置的组。当配置设置由功能隔离到单独的类时，应用遵循两个重要软件工程原则：

- **接口分离原则 (ISP)**: 依赖于配置设置的功能(类)仅依赖于其使用的配置设置。
- **关注点分离**: 应用的不同部件的设置不彼此依赖或相互耦合。

[查看或下载示例代码 \(如何下载\)](#) 跟随示例应用可更轻松地理解本文。

基本选项配置

基本选项配置已作为示例 #1 在[示例应用](#)中进行了演示。

选项类必须为包含公共无参数构造函数的非抽象类。以下类 `MyOptions` 具有两种属性：`Option1` 和 `Option2`。设置默认值为可选，但以下示例中的类构造函数设置了 `Option1` 的默认值。`Option2` 具有通过直接初始化属性设置的默认值 (`Models/MyOptions.cs`)：

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

`MyOptions` 类已通过 `IConfigureOptions<TOptions>` 添加到服务容器并绑定到配置：

```
// Example #1: Basic options
// Register the Configuration instance which MyOptions binds against.
services.Configure<MyOptions>(Configuration);
```

以下页上的模型通过 `IOptions<TOptions>` 使用[构造函数依赖关系注入](#)来访问设置 (`Pages/Index.cshtml.cs`)：

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #1: Simple options
var option1 = _options.Option1;
var option2 = _options.Option2;
SimpleOptions = $"option1 = {option1}, option2 = {option2}";
```

示例的 appsettings.json 文件指定 `option1` 和 `option2` 的值：

```
{
    "option1": "value1_from_json",
    "option2": -1,
    "subsection": {
        "suboption1": "subvalue1_from_json",
        "suboption2": 200
    }
}
```

运行应用时，页模型的 `OnGet` 方法返回显示选项类值的字符串：

```
option1 = value1_from_json, option2 = -1
```

通过委托配置简单选项

通过委托配置简单选项已作为示例 #2 在[示例应用](#)中进行了演示。

使用委托设置选项值。此示例应用使用 `MyOptionsWithDelegateConfig` 类 (Models/MyOptionsWithDelegateConfig.cs)：

```
public class MyOptionsWithDelegateConfig
{
    public MyOptionsWithDelegateConfig()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

在以下代码中，已向服务容器添加第二个 `IConfigureOptions<TOptions>` 服务。它通过 `MyOptionsWithDelegateConfig` 使用委托来配置绑定：

```
// Example #2: Options bound and configured by a delegate
services.Configure<MyOptionsWithDelegateConfig>(myOptions =>
{
    myOptions.Option1 = "value1_configured_by_delegate";
    myOptions.Option2 = 500;
});
```

Index.cshtml.cs:

```
private readonly MyOptionsWithDelegateConfig _optionsWithDelegateConfig;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #2: Options configured by delegate
var delegate_config_option1 = _optionsWithDelegateConfig.Option1;
var delegate_config_option2 = _optionsWithDelegateConfig.Option2;
SimpleOptionsWithDelegateConfig =
    $"delegate_option1 = {delegate_config_option1}, " +
    $"delegate_option2 = {delegate_config_option2}";
```

可添加多个配置提供程序。配置提供程序在 NuGet 包中可用。应用此提供程序，以便将其注册。

每次调用 `Configure<TOptions>` 都将添加 `IConfigureOptions<TOptions>` 服务到服务容器。在前面的示例中，`Option1` 和 `Option2` 的值同时在 `appsettings.json` 中指定，但 `Option1` 和 `Option2` 的值被配置的委托替代。

当启用多个配置服务时，指定的最后一个配置源优于其他源，由其设置配置值。运行应用时，页模型的 `OnGet` 方法返回显示选项类值的字符串：

```
delegate_option1 = value1_configured_by_delegate, delegate_option2 = 500
```

子选项配置

子选项配置已作为示例 #3 在[示例应用](#)中进行了演示。

应用应创建适用于应用中特定功能组(类)的选项类。需要配置值的部分应用应仅有权访问其使用的配置值。

将选项绑定到配置时，选项类型中的每个属性都将绑定到窗体 `property[:sub-property:]` 的配置键。例如，`MyOptions.Option1` 属性将绑定到从 `appsettings.json` 中的 `option1` 属性读取的键 `Option1`。

在以下代码中，已向服务容器添加第三个 `IConfigureOptions<TOptions>` 服务。它将 `MySubOptions` 绑定到 `appsettings.json` 文件的 `subsection` 部分：

```
// Example #3: Sub-options
// Bind options using a sub-section of the appsettings.json file.
services.Configure<MySubOptions>(Configuration.GetSection("subsection"));
```

`GetSection` 扩展方法需要 [Microsoft.Extensions.Options.ConfigurationExtensions](#) NuGet 包。如果应用使用 [Microsoft.AspNetCore.All](#) 元包，将自动包含此包。

示例的 `appsettings.json` 文件定义具有 `suboption1` 和 `suboption2` 的键的 `subsection` 成员：

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  }
}
```

`MySubOptions` 类定义属性 `SubOption1` 和 `SubOption2`，以保留子选项值 (`Models/MySubOptions.cs`)：

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }

    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

页模型的 `OnGet` 方法返回包含子选项值的字符串 (`Pages/Index.cshtml.cs`)：

```
private readonly MySubOptions _subOptions;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #3: Sub-options
var subOption1 = _subOptions.SubOption1;
var subOption2 = _subOptions.SubOption2;
SubOptions = $"subOption1 = {subOption1}, subOption2 = {subOption2}";
```

运行应用时，`OnGet` 方法返回显示子选项类值的字符串：

```
subOption1 = subvalue1_from_json, subOption2 = 200
```

视图模型或通过直接视图注入提供的选项

视图模型或通过直接视图注入提供的选项已作为示例 #4 在[示例应用](#)中进行了演示。

可在视图模型中或通过将 `IOptions<TOptions>` 直接注入到视图 (`Pages/Index.cshtml.cs`) 来提供选项：

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #4: Bind options directly to the page
MyOptions = _options;
```

对于直接注入，通过 `@inject` 指令注入 `IOptions<MyOptions>`：

```
@page
@model IndexModel
@using Microsoft.Extensions.Options
@using UsingOptionsSample.Models
@inject IOptions<MyOptions> OptionsAccessor
 @{
    ViewData["Title"] = "Using Options Sample";
}
<h1>@ViewData["Title"]</h1>
```

运行应用时，选项值显示在已呈现的页中：

Example #4: Model and injected options

Options provided by the model

Options provided by the model: `@Model.MyOptions.Option1` and `@Model.MyOptions.Option2`

Option1: value1_from_json

Option2: -1

Options injected into the page

Options injected into the page: `@inject IOptions<MyOptions> OptionsAccessor` with `@OptionsAccessor.Value.Option1` and `@OptionsAccessor.Value.Option2`

Option1: value1_from_json

Option2: -1

通过 `IOptionsSnapshot` 重新加载配置数据

通过 `IOptionsSnapshot` 重新加载配置数据已作为示例 #5 在[示例应用](#)中进行了演示。

需要 ASP.NET Core 1.1 或更高版本。

`IOptionsSnapshot` 支持包含最小处理开销的重新加载选项。在 ASP.NET Core 1.1 中，`IOptionsSnapshot` 是 `IOptionsMonitor<TOptions>` 的快照，且在每次监视器基于数据源更改触发更改时自动更新。在 ASP.NET Core 2.0 及更高版本中，在针对请求的生存期访问和缓存选项时，将针对每个请求计算一次选项。

以下示例演示如何在更改 `appsettings.json` (Pages/Index.cshtml.cs) 后创建新的 `IOptionsSnapshot`。在更改文件和重新加载配置之前，针对服务器的多个请求返回 `appsettings.json` 文件提供的常数值。

```
private readonly MyOptions _snapshotOptions;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #5: Snapshot options
var snapshotOption1 = _snapshotOptions.Option1;
var snapshotOption2 = _snapshotOptions.Option2;
SnapshotOptions =
    $"snapshot option1 = {snapshotOption1}, " +
    $"snapshot option2 = {snapshotOption2}";
```

下图显示从 `appsettings.json` 文件加载的初始 `option1` 和 `option2` 值：

```
snapshot option1 = value1_from_json, snapshot option2 = -1
```

将 appsettings.json 文件中的值更改为 `value1_from_json UPDATED` 和 `200`。保存 appsettings.json 文件。刷新浏览器，查看更新的选项值：

```
snapshot option1 = value1_from_json UPDATED, snapshot option2 = 200
```

包含 `IConfigureNamedOptions` 的命名选项支持

包含 `IConfigureNamedOptions` 的命名选项支持已作为示例 #6 在[示例应用](#)中进行了演示。

需要 ASP.NET Core 2.0 或更高版本。

命名选项支持允许应用在命名选项配置之间进行区分。在示例应用中，命名选项通过 `ConfigureNamedOptions<TOptions>.Configure` 方法声明：

```
// Example #6: Named options (named_options_1)
// Register the ConfigurationBuilder instance which MyOptions binds against.
// Specify that the options loaded from configuration are named
// "named_options_1".
services.Configure<MyOptions>("named_options_1", Configuration);

// Example #6: Named options (named_options_2)
// Specify that the options loaded from the MyOptions class are named
// "named_options_2".
// Use a delegate to configure option values.
services.Configure<MyOptions>("named_options_2", myOptions =>
{
    myOptions.Option1 = "named_options_2_value1_from_action";
});
```

示例应用通过 `IOptionsSnapshot<TOptions>.Get` (`Pages/Index.cshtml.cs`) 访问命名选项：

```
private readonly MyOptions _named_options_1;
private readonly MyOptions _named_options_2;
```

```
public IndexModel(
    IOptions<MyOptions> optionsAccessor,
    IOptions<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptions<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.Value;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.Value;
    _subOptions = subOptionsAccessor.Value;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #6: Named options
var named_options_1 =
    $"named_options_1: option1 = {_named_options_1.Option1}, " +
    $"option2 = {_named_options_1.Option2}";
var named_options_2 =
    $"named_options_2: option1 = {_named_options_2.Option1}, " +
    $"option2 = {_named_options_2.Option2}";
NamedOptions = $"{named_options_1} {named_options_2}";
```

运行示例应用，将返回命名选项：

```
named_options_1: option1 = value1_from_json, option2 = -1
named_options_2: option1 = named_options_2_value1_from_action, option2 = 5
```

从配置中提供从 appsettings.json 文件中加载的 `named_options_1` 值。通过以下内容提供 `named_options_2` 值：

- 针对 `Option1` 的 `ConfigureServices` 中的 `named_options_2` 委托。
- `MyOptions` 类提供的 `Option2` 的默认值。

通过 `OptionsServiceCollectionExtensions.ConfigureAll` 方法配置所有命名选项实例。以下代码将针对包含公共值的所有命名配置实例配置 `Option1`。将以下代码手动添加到 `Configure` 方法：

```
services.ConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "ConfigureAll replacement value";
});
```

添加代码后运行示例应用将产生以下结果：

```
named_options_1: option1 = ConfigureAll replacement value, option2 = -1
named_options_2: option1 = ConfigureAll replacement value, option2 = 5
```

注意

在 ASP.NET Core 2.0 及更高版本中，所有选项都为命名实例。现有 `IConfigureOption` 实例将被视为面向为 `string.Empty` 的 `Options.DefaultName` 实例。`IConfigureNamedOptions` 还可实现 `IConfigureOptions`。`IOptionsFactory<TOptions>`（引用源）的默认实现具有适当使用每个选项的逻辑。`null` 命名选项用于面向所有命名实例而不是某一特定命名实例（`ConfigureAll` 和 `PostConfigureAll` 使用此约定）。

IPostConfigureOptions

需要 ASP.NET Core 2.0 或更高版本。

通过 `IPostConfigureOptions<TOptions>` 设置后期配置。在发生所有 `IConfigureOptions<TOptions>` 配置后运行后期配置：

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

`PostConfigure<TOptions>` 可用于对命名选项进行后期配置：

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

使用 `PostConfigureAll<TOptions>` 对所有命名配置实例进行后期配置：

```
services.PostConfigureAll<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

选项工厂、监视和缓存

`IOptionsMonitor` 用于在 `TOptions` 实例更改时进行通知。`IOptionsMonitor` 支持可重载选项、更改通知和 `IPostConfigureOptions`。

`IOptionsFactory<TOptions>` (ASP.NET Core 2.0 或更高版本) 负责创建新选项实例。它具有单个 [创建方法](#)。默认实现采用所有已注册 `IConfigureOptions` 和 `IPostConfigureOptions` 并首先运行所有配置，然后才进行后期配置。它区分 `IConfigureNamedOptions` 和 `IConfigureOptions` 且仅调用适当的接口。

`IOptionsMonitorCache<TOptions>` (ASP.NET Core 2.0 或更高版本) 由 `IOptionsMonitor` 用于缓存 `TOptions` 实例。`IOptionsMonitorCache` 可使监视器中的选项实例无效，以便重新计算值 ([TryRemove](#))。还可通过 `TryAdd` 手动引入值。在应按需重新创建所有命名实例时使用 [清除方法](#)。

在启动期间访问选项

`IOptions` 可用于 `Configure` 中，因为在 `Configure` 方法执行之前已生成服务。如果在 `ConfigureServices` 中生成服务提供程序以访问选项，则它不应包含生成服务提供程序后所提供的任何选项配置。因此，由于服务注册的顺序，可能存在不一致的选项状态。

由于选项通常从配置中加载，因此，可在 `Configure` 和 `ConfigureServices` 中的启动中使用配置。有关在启动期间使用配置的示例，请参阅[应用程序启动主题](#)。

请参阅

- [配置](#)

ASP.NET Core 中的日志记录

2018/5/17 • 26 min to read • [Edit Online](#)

作者: [Steve Smith](#) 和 [Tom Dykstra](#)

ASP.NET Core 支持适用于各种日志记录提供程序的日志记录 API。通过内置提供程序，可向一个或多个目标发送日志，还可插入第三方记录框架。本文介绍如何在代码中使用内置日志记录 API 和提供程序。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

[查看或下载示例代码\(如何下载\)](#)

如何创建日志

要创建日志，请先从[依赖关系注入](#)容器获取 `ILogger` 对象：

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
}
```

然后在该记录器对象上调用日志记录方法：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

此示例使用 `TodoController` 类创建日志作为类别。[本文的稍后部分](#)对这些类别进行了介绍。

ASP.NET Core 不提供异步记录器方法，因为日志记录的速度应非常快，使用异步的代价是不值得的。如果发现自己的实际情况与上述不同，请考虑更改记录方式。如果数据存储速度较慢，请先将日志消息写入快速存储，稍后再将其转移至低速存储。例如，记录到由另一进程读取和暂留以减缓存储的消息队列。

如何添加提供程序

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

日志记录提供程序通过 `ILogger` 对象获取所创建的消息，并显示或存储它们。例如，控制台提供程序在控制台上显示消息，Azure App Service 提供程序可将消息存储在 Azure blob 存储中。

要使用提供程序，请在 `Program.cs` 中调用提供程序的 `Add<ProviderName>` 扩展方法：

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
    {
        var env = hostingContext.HostingEnvironment;
        config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange:
true);
        config.AddEnvironmentVariables();
    })
        .ConfigureLogging((hostingContext, logging) =>
    {
        logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
    })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

通过默认项目模板可以使用 `CreateDefaultBuilder` 方法登录：

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

本文稍后部分介绍了每个[内置日志记录提供程序](#)，并提供了[第三方日志记录提供程序](#)的链接。

日志记录输出示例

从命令行运行上一部分所示的示例代码时，将在控制台中看到日志。以下是控制台输出示例：

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - 
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

转到 `http://localhost:5000/api/todo/0`，触发上一部分所示的两个 `ILogger` 调用的执行，创建了以上日志。

在 Visual Studio 中运行示例应用程序时，“调试”窗口中将显示如下日志：

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item 0
TodoApi.Controllers.TodoController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting HTTP status
code 404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TodoController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

由上一部分所示的 `ILogger` 调用创建的日志以“`TodoApi.Controllers.TodoController`”开头的。以“`Microsoft`”类别开头的日志来自 ASP.NET Core。ASP.NET Core 本身和应用程序代码使用相同的日子记录 API 和日子记录提供程序。

本文余下部分将介绍有关日子记录的某些详细信息及选项。

NuGet 包

`ILogger` 和 `ILoggerFactory` 接口位于 [Microsoft.Extensions.Logging.Abstractions](#) 中，其默认实现位于 [Microsoft.Extensions.Logging](#) 中。

日子类别

所创建的每个日子都包含一个类别。在创建 `ILogger` 对象时指定类别。类别可以是任意字符串，但约定使用写入日子的类的完全限定名称。例如“`TodoApi.Controllers.TodoController`”。

可以将类别指定为字符串，或使用从类型派生类别的扩展方法。要将类别指定为字符串，请在 `ILoggerFactory` 实例上调用 `CreateLogger`，如下所示。

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApi.Controllers.TodoController");
    }
}
```

如下方示例所示，在大多数情况下使用 `ILogger<T>` 更简单。

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
}
```

这相当于使用 `T` 的完全限定类型名称来调用 `CreateLogger`。

日志级别

每次写入日志时都需指定其 `LogLevel`。日志级别指示严重性或重要程度。例如，如果方法正常结束则写入 `Information` 日志，如果方法返回 404 返回代码则写入 `Warning` 日志，如果捕获未知异常则写入 `Error` 日志。

在下列代码示例中，由方法的名称（如 `LogWarning`）指定日志级别。第一个参数是 **日志事件 ID**。第二个形参是 **消息模板**，包含由剩余方法形参提供的实参值的占位符。本文稍后部分更详细地介绍了方法参数。

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, ".GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

在方法名称中包含级别的日志方法是 `ILogger` 的扩展方法。在后台，这些方法调用可接受 `LogLevel` 参数的 `Log` 方法。可直接调用 `Log` 方法而不调用其中某个扩展方法，但语法相对复杂。有关详细信息，请参阅 [ILogger 接口和记录器扩展源代码](#)。

ASP.NET Core 定义了以下 **日志级别**（按严重性从低到高排列）。

- 跟踪 = 0

表示仅对于开发人员调试问题有价值的信息。这些消息可能包含敏感应用程序数据，因此不得在生产环境中启用它们。默认情况下禁用。示例：`Credentials: {"User": "someuser", "Password": "P@ssword"}`

- 调试 = 1

表示在开发和调试过程中短期有用的信息。示例: Entering method Configure with flag set to true. 除非要排查问题, 否则通常不会在生产中启用 Debug 级别日志, 因为日志数量过多。

- 信息 = 2

用于跟踪应用程序的常规流。这些日志通常有长期价值。示例: Request received for path /api/todo

- 警告 = 3

表示应用程序流中的异常或意外事件。可能包括不会中断应用程序运行但仍需调查的错误或其他条件。

Warning 日志级别常用于已处理的异常。示例: FileNotFoundException for file quotes.txt.

- 错误 = 4

表示无法处理的错误和异常。这些消息指示的是当前活动或操作(如当前 HTTP 请求)中的失败, 而不是应用程序范围的失败。日志消息示例: Cannot insert record due to duplicate key violation.

- 严重 = 5

需要立即关注的失败。例如数据丢失、磁盘空间不足。

可以使用日志级别控制写入到特定存储介质或显示窗口的日志输出量。例如在生产中, 可将所有 Information 及以下级别的日志放在卷数据存储, 将所有 Warning 及以上级别的日志放在值数据存储。在开发期间, 通常要将严重性为 Warning 或以上级别的日志发送到控制台。需要进行故障排除时, 可添加 Debug 级别。本文稍后的[日志筛选](#)部分介绍如何控制提供程序处理的日志级别。

ASP.NET Core 框架为框架事件写入 Debug 级别日志。本文前面部分提供的日志示例排除了低于 Information 级别的日志, 因此未显示 Debug 级别日志。如果运行配置为显示控制台提供程序 Debug 及以上级别的日志的示例应用程序, 则控制台日志示例如下所示。

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
dbug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/Todo/{id}'.
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Update (TodoApi)' with id '089d59b6-92ec-472d-b552-cc613df625d' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Delete (TodoApi)' with id 'f3476abe-4bd9-4ad3-9261-3ead09607366' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action TodoApi.Controllers.TodoController.GetById (TodoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method TodoApi.Controllers.TodoController.GetById (TodoApi), returned result
Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing StatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 0.8788ms
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404
```

日志事件 ID

每次写入日志时都可指定一个事件 ID。该示例应用通过使用本地定义的 `LoggingEvents` 类来执行此操作：

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, ".GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

事件 ID 是一个整数值，可用于关联多组已记录事件。例如，向购物车添加项的日志可以是事件 ID 1000，完成购买的日志可以是事件 ID 1001。

在日志记录输出中，事件 ID 既可存储在字段里，也可包含在文本消息内，这由提供程序来决定。调试提供程序不显示事件 ID，但控制台提供程序会将其显示在类别后的括号里：

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

日志消息模板

每次写入日志消息都会提供一个消息模板。消息模板可以是字符串，也可以包含放置参数值的命名占位符。该模板不是格式字符串，应对占位符进行命名而不是编号。

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, ".GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

占位符的顺序(而非其名称)决定了为其提供值的参数。如果你有以下代码：

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

生成的日志消息如下所示：

```
Parameter values: parm1, parm2
```

记录框架以这种方式对消息进行格式化，以便日志记录提供程序能实现[语义日志记录（又称结构化日志记录）](#)。由于传递到日志记录系统的是参数本身（而不仅仅是格式化的消息模板），因此除消息模板外，日志记录提供程序还可将参数值存储为字段。如果将日志输出定向到 Azure 表存储，则记录器方法调用将如下所示：

```
_logger.LogInformation("Getting item {ID} at {RequestTime}", id, DateTime.Now);
```

每个 Azure 表实体都具有 `ID` 和 `RequestTime` 属性，简化了对日志数据的查询。无需分析文本消息的超时，即可找到特定 `RequestTime` 范围内的全部日志。

日志记录异常

记录器方法有可传入异常的重载，如下方示例所示：

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

不同的提供程序处理异常信息的方式不同。以下是上示代码的调试提供程序输出示例。

```
TodoApi.Controllers.TodoController:Warning: GetById(036dd898-fb01-47e8-9a65-f92eb73cf924) NOT FOUND
System.Exception: Item not found exception.
at TodoApi.Controllers.TodoController.GetById(String id) in
C:\logging\sample\src\TodoApi\Controllers\TodoController.cs:line 226
```

日志筛选

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

可为特定或所有提供程序和类别指定最低日志级别。最低级别以下的日志不会传递给该提供程序，因此不会显示或存储它们。

要禁止显示所有日志，可将 `LogLevel.None` 指定为最低日志级别。`LogLevel.None` 的整数值为 6，它大于 `LogLevel.Critical` (5)。

在配置中创建筛选规则

项目模板创建调用 `CreateDefaultBuilder` 的代码，以便为控制台和调试提供程序设置日志记录。

`CreateDefaultBuilder` 方法还使用如下所示的代码，设置日志记录以查找 `Logging` 部分的配置：

```

public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
    {
        var env = hostingContext.HostingEnvironment;
        config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange:
true);
        config.AddEnvironmentVariables();
    })
    .ConfigureLogging((hostingContext, logging) =>
    {
        logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
    })
    .UseStartup<Startup>()
    .Build();

    webHost.Run();
}

```

配置数据按提供程序和类别指定最低日志级别，如下方示例所示：

```
{
    "Logging": {
        "IncludeScopes": false,
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "LogLevel": {
                "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
                "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
                "Microsoft.AspNetCore.Mvc.Razor": "Error",
                "Default": "Information"
            }
        },
        "LogLevel": {
            "Default": "Debug"
        }
    }
}
```

此 JSON 将创建六条筛选规则，一条用于调试提供程序，四条用于控制台提供程序，一条用于所有提供程序。稍后你将了解在创建 `ILogger` 对象后，如何为每个提供程序从上述规则中选择其一。

代码中的筛选规则

可在代码中注册筛选规则，如下方示例所示：

```

WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
            .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace)
    .Build();

```

第二个 `AddFilter` 使用类型名称来指定调试提供程序。第一个 `AddFilter` 应用于全部提供程序，因为它未指定提供程序类型。

如何筛选已应用的规则

先前示例中显示的配置数据和 `AddFilter` 代码会创建下表所示的规则。前六条由配置示例创建，后两条由代码示例创建。

数字	提供程序	类别的开头为...	最低日志级别
1	调试	全部类别	信息
2	控制台	Microsoft.AspNetCore.Mvc. Razor.Internal	警告
3	控制台	Microsoft.AspNetCore.Mvc. Razor.Razor	调试
4	控制台	Microsoft.AspNetCore.Mvc. Razor	Error
5	控制台	全部类别	信息
6	全部提供程序	全部类别	调试
7	全部提供程序	系统	调试
8	调试	Microsoft	跟踪

创建 `ILogger` 对象来写入日志时，`ILoggerFactory` 对象将从根据提供程序选择一条规则，将其应用于该记录器。将按所选规则筛选该 `ILogger` 对象写入的所有消息。从可用规则中为每个提供程序和类别对选择最具体的规则。

在为给定的类别创建 `ILogger` 时，以下算法将用于每个提供程序：

- 选择匹配提供程序或其别名的所有规则。如果找不到，则选择具有空提供程序的所有规则。
- 根据上一步的结果，选择具有最长匹配类别前缀的规则。如果找不到，则选择未指定类别的所有规则。
- 如果选择了多条规则，则采用最后一条。
- 如果未选择任何规则，则使用 `MinimumLevel`。

例如，假设你拥有上述规则列表，并为类别“`Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine`”创建了 `ILogger` 对象：

- 对于调试提供程序，规则 1、6 和 8 适用。规则 8 最为具体，因此选择了它。
- 对于控制台提供程序，符合的有规则 3、规则 4、规则 5 和规则 6。规则 3 最为具体。

在使用 `ILogger` 为类别“`Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine`”创建日志时，`Trace` 及以上级别的日志将发送到调试提供程序，`Debug` 及以上级别的日志将发送到控制台提供程序。

提供程序别名

虽然可以使用类型名称在配置中指定提供程序，但每个提供程序都定义了更短且更易于使用的别名。对于内置提供程序，请使用以下别名：

- 控制台
- 调试
- EventLog
- AzureAppServices
- TraceSource
- EventSource

默认最低级别

仅当配置或代码中的规则对给定提供程序和类别都不适用时，最低级别设置才会生效。下面的示例演示如何设置最低级别：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
    .Build();
```

如果没有明确设置最低级别，则默认值为 `Information`，它表示 `Trace` 和 `Debug` 日志将被忽略。

筛选器函数

可向筛选器函数写入代码以应用筛选规则。对配置或代码没有向其分配规则的所有提供程序和类别调用筛选器函数。函数中的代码有权访问提供程序类型、类别和日志级别，以决定是否记录某条消息。例如：

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
{
    logBuilder.AddFilter((provider, category, logLevel) =>
    {
        if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
            category == "TodoApi.Controllers.TodoController")
        {
            return false;
        }
        return true;
    });
})
.Build();
```

日志作用域

可以将逻辑操作集划入范围，从而将相同的数据附加到在此集中创建的每个日志。例如，可让处理事务时创建的每个日志都包含事务 ID。

范围是由 `ILogger.BeginScope<TState>` 方法返回的 `IDisposable` 类型，持续至释放为止。要使用作用域，请在 `using` 块中包装记录器调用，如下所示：

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {ID}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({ID}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

下列代码为控制台提供程序启用作用域：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

在 Program.cs 中：

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

注意

要启用基于作用域的日志记录，必须先配置 `IncludeScopes` 控制台记录器选项。ASP.NET Core 2.1 发布后，将支持使用 appsettings 配置文件来配置 `IncludeScopes`。

每条日志消息都包含作用域内的信息：

```
info: TodoApi.Controllers.TodoController[1002]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApi.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      GetById(0) NOT FOUND
```

内置日志记录提供程序

ASP.NET Core 提供以下提供程序：

- [控制台](#)
- [调试](#)
- [EventSource](#)
- [EventLog](#)
- [TraceSource](#)
- [Azure 应用服务](#)

控制台提供程序

Microsoft.Extensions.Logging.Console 提供程序包向控制台发送日志输出。

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
logging.AddConsole()
```

调试提供程序

Microsoft.Extensions.Logging.Debug 提供程序包使用 System.Diagnostics.Debug 类(`Debug.WriteLine` 方法调用)来写入日志输出。

在 Linux 中, 此提供程序将日志写入 /var/log/message。

- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
logging.AddDebug()
```

EventSource 提供程序

对于面向 ASP.NET Core 1.1.0 或更高版本的应用, Microsoft.Extensions.Logging.EventSource 提供程序包可实现事件跟踪。在 Windows 中, 它使用 ETW。提供程序可跨平台使用, 但尚无支持 Linux 或 macOS 的事件集合和显示工具。

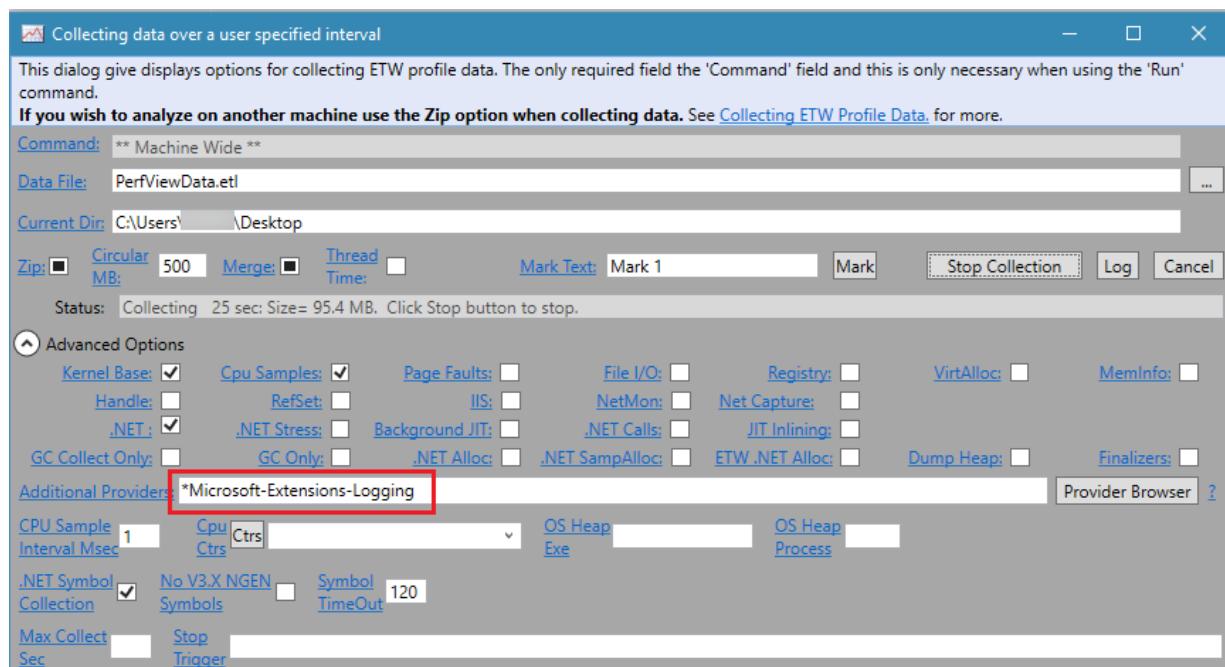
- ASP.NET Core 2.x
- ASP.NET Core 1.x

```
logging.AddEventSourceLogger()
```

可使用 [PerfView 实用工具](#)收集和查看日志。虽然其他工具也可以查看 ETW 日志, 但在处理由 ASP.NET 发出的 ETW 事件时, 使用 PerfView 能获得最佳体验。

要将 PerfView 配置为收集此提供程序记录的事件, 请向 Additional Providers 列表添加字符串

```
*Microsoft-Extensions-Logging
```



Windows EventLog 提供程序

[Microsoft.Extensions.Logging.EventLog](#) 提供程序包向 Windows 事件日志发送日志输出。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddEventLog()
```

TraceSource 提供程序

[Microsoft.Extensions.Logging.TraceSource](#) 提供程序包使用 [System.Diagnostics.TraceSource](#) 库和提供程序。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
logging.AddTraceSource(sourceSwitchName);
```

[AddTraceSource 重载](#) 允许传入资源开关和跟踪侦听器。

要使用此提供程序，应用程序必须在 .NET Framework(而非 .NET Core) 上运行。使用提供程序，可以将消息路由到多个 [侦听器](#)，例如示例应用程序中使用的 [TextWriterTraceListener](#)。

以下示例配置了 [TraceSource](#) 提供程序，用于向控制台窗口记录 [Warning](#) 和更高级别的消息。

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddDebug();

    // add Trace Source logging
    var testSwitch = new SourceSwitch("sourceSwitch", "Logging Sample");
    testSwitch.Level = SourceLevels.Warning;
    loggerFactory.AddTraceSource(testSwitch,
        new TextWriterTraceListener(writer: Console.Out));
}
```

Azure App Service 提供程序

[Microsoft.Extensions.Logging.AzureAppServices](#) 提供程序包将日志写入 Azure App Service 应用的文件系统，以及 Azure 存储帐户中的 [blob 存储](#)。该提供程序仅适用于面向 ASP.NET Core 1.1.0 或更高版本的应用。

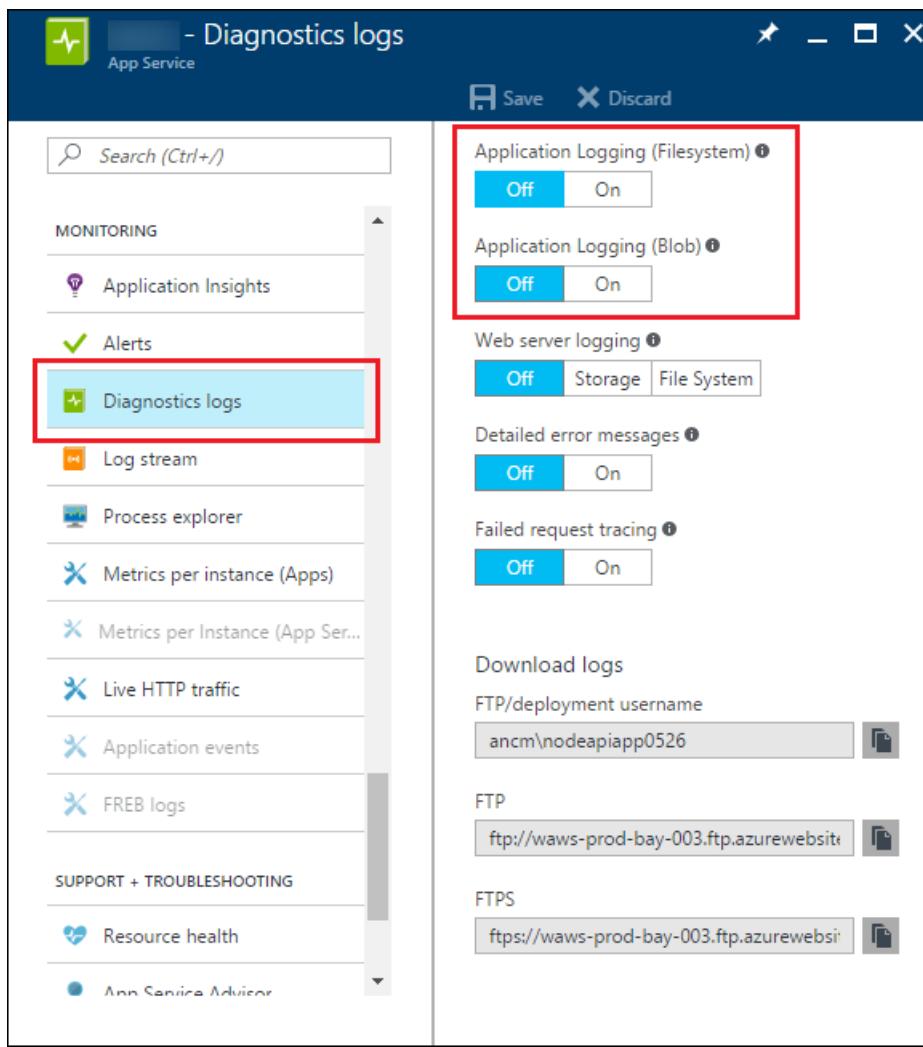
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果面向 .NET Core，无需安装提供程序包或显式调用 [AddAzureWebAppDiagnostics](#)。将应用部署到 Azure App Service 时，提供程序对应用自动可用。

如果面向 .NET Framework，需要向项目添加提供程序包并调用 [AddAzureWebAppDiagnostics](#)：

```
logging.AddAzureWebAppDiagnostics();
```

在部署 App Service 应用时，应用程序将遵循 Azure 门户中 App Service 页下 [诊断日志](#) 部分的设置。这些设置将在更改后立即生效，无需重启应用或重新向其部署代码。



日志文件的默认位置是 D:\home\LogFiles\Application 文件夹， 默认文件名为 diagnostics-yyyy-mm-dd.txt。默认文件大小上限为 10 MB, 默认最大保留文件数为 2。默认 blob 名为 {app-name}-{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt。有关默认行为的详细信息，请参阅 [AzureAppServicesDiagnosticsSettings](#)。

该提供程序仅当项目在 Azure 环境中运行时有效。它在本地运行时无效。也就是说，不会写入本地文件或 blob 的本地开发存储。

第三方日志记录提供程序

以下第三方日志记录框架适用于 ASP.NET Core:

- [elmah.io](#) - Elmah.io 服务的提供程序
- [JSONLog](#) - 在服务器端日志中记录 JavaScript 异常和其他客户端事件。
- [Loggr](#) - Loggr 服务的提供程序
- [NLog](#) - NLog 库的提供程序
- [Serilog](#) - Serilog 库的提供程序

某些第三方框架可以执行语义日志记录(又称结构化日志记录)。

使用第三方框架与使用内置提供程序类似:向项目添加 NuGet 包并在 `I LoggerFactory` 上调用扩展方法即可。有关详细信息，请参阅各框架的相关文档。

Azure 日志流式处理

通过 Azure 日志流式处理，可从以下位置实时查看日志活动：

- 应用程序服务器
- Web 服务器
- 请求跟踪失败

要配置 Azure 日志流式处理，请执行以下操作：

- 从应用程序的门户页导航到“诊断日志”页
- 将“应用程序日志记录 (Filesystem)”设置为启用。

The screenshot shows the 'Diagnostics logs' configuration page for an Azure App Service. The left sidebar lists various monitoring and troubleshooting tools, with 'Diagnostics logs' highlighted by a red box. The main area contains settings for Application Logging (Filesystem), which is currently set to 'On'. Other settings shown include Application Logging (Blob) set to 'Off', Web server logging set to 'Storage' (with 'File System' as an option), Detailed error messages set to 'On', and Failed request tracing set to 'Off'. Below these are sections for Download logs, FTP deployment username (set to 'azure-raffle\azure-ftp-deployment-user'), and FTP and FTPS connection details (both set to 'ftp://waws-prod-am2-135.ftp.azurewebsites.windows.net').

导航到“日志流式处理”页，查看应用程序消息。它们由应用程序通过 `ILogger` 接口记录。

The screenshot shows the Azure Log Stream interface. On the left, there's a sidebar with various navigation options: Performance test, Resource explorer, Testing in production, Extensions, MOBILE (Easy tables, Easy APIs, Data connections), API (API definition, CORS), MONITORING (Application Insights, Alerts, Diagnostics logs, Log stream, Process explorer), SUPPORT + TROUBLESHOOTING (Resource health, App Service Advisor). The 'Log stream' option is highlighted with a red box. The main pane displays log entries from November 16, 2017, at 18:39:47. The logs include details about browser versions (e.g., Chrome 62.0.3202.94, Safari 537.36) and IP addresses (e.g., 73.130.153.74, 73.130.153.153).

Date	Time	Message
2017-11-16	18:39:47	Welcome, you are now connected to log-streaming service.
2017-11-16	18:39:04 ~1J	GET /api/vfs/site/wwwroot/_-15108573274048X-ARR-LOG-ID=36b4e37b-e31e-433d-82dd-(Windows+NT+10.0;+Win64;+x64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 - https://wecontent.WebsitesIndex?cacheability=3®ion=eastus&cacheVersion=0&defaultCloudName=azure&extensionName=WebsitesExtensionsAuthority-portal.azure.com .scm.azurewebsites.net 200 0 0 966 1697 156
2017-11-16	18:39:06	GET / X-ARR-LOG-ID=784009e2-cecb-4815-968a-f63b9e8879a6 80 - 73.130.153.74 Mozilla/5.0+(Windows+NT+10.0;+Win64;+x64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145 websites.net 403 14 0 370 943 296
2017-11-16	18:39:11 ~1	GET /api/siteextensions X-ARR-LOG-ID=7850d378-909d-4b1d-a425-9408dab70910 443 - Mozilla/5.0+(Windows+NT+10.0;+Win64;+x64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 0 499 1190 703
2017-11-16	18:39:39	GET /ticket.html X-ARR-LOG-ID=b7e33a93-f31e-436f-8214-672dfe954c74 80 - 73.130.153+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145 file.azurewebsites.net 304 0 0 314 1087 696
2017-11-16	18:39:40	AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=70f2946b-4139-4f65-a936-ba437411cdf9 80 - 73.130.153+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145 file.azurewebsites.net 304 0 0 314 1087 0
2017-11-16	18:39:40	AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=c5f3593f-4d4b-4811-b150-500d58341622 80 - 73.130.153+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145 file.azurewebsites.net 304 0 0 314 1087 218
2017-11-16	18:39:54	AZURE-RAFFLE GET /ticket.html X-ARR-LOG-ID=ae4526f8-c8c7-4399-9f19-9c3fdac7cdcb 80 - 73.130.153+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/62.0.3202.94+Safari/537.36 ARRAffinity=146838c347f645f6238d55649b145 file.azurewebsites.net 304 0 0 314 1087 15

请参阅

[使用 LoggerMessage 的高性能日志记录](#)

在 ASP.NET Core 中使用 LoggerMessage 的高性能日志记录

2018/5/14 • 8 min to read • [Edit Online](#)

作者: [Luke Latham](#)

`LoggerMessage` 功能创建可缓存的委托, 该功能比记录器扩展方法(例如 `LogInformation`、`LogDebug` 和 `.LogError`)需要的对象分配和计算开销少。对于高性能日志记录方案, 请使用 `LoggerMessage` 模式。

与记录器扩展方法相比, `LoggerMessage` 具有以下性能优势:

- 记录器扩展方法需要将值类型(例如 `int`)“装箱”(转换)到 `object` 中。`LoggerMessage` 模式使用带强类型参数的静态 `Action` 字段和扩展方法来避免装箱。
- 记录器扩展方法每次写入日志消息时必须分析消息模板(命名的格式字符串)。如果已定义消息, 那么 `LoggerMessage` 只需分析一次模板即可。

[查看或下载示例代码\(如何下载\)](#)

此示例应用通过基本引号跟踪系统演示 `LoggerMessage` 功能。此应用使用内存中数据库添加和删除引号。发生这些操作时, 通过 `LoggerMessage` 模式生成日志消息。

LoggerMessage.Define

`Define(LogLevel, EventId, 字符串)` 创建用于记录消息的 `Action` 委托。`Define` 重载允许向命名的格式字符串(模板)传递最多六个类型参数。

提供给 `Define` 方法的字符串是一个模板, 而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称, 建议使用[帕斯卡拼写法](#)。例如: `{Count}`、`{FirstName}`。

每条日志消息都是一个 `Action`, 保存在由 `LoggerMessage.Define` 创建的静态字段中。例如, 示例应用创建一个字段来为索引页 (`Internal/LoggerExtensions.cs`) 描述 GET 请求的日志消息:

```
private static readonly Action<ILogger, Exception> _indexPageRequested;
```

对于 `Action`, 指定:

- 日志级别。
- 具有静态扩展方法名称的唯一事件标识符 (`EventId`)。
- 消息模板(命名的格式字符串)。

对示例应用的索引页的请求设置:

- 将日志级别设置为 `Information`。
- 将事件 ID 设置为具有 `IndexPageRequested` 方法名称的 `1`。
- 将消息模板(命名的格式字符串)设置为字符串。

```
_indexPageRequested = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(1, nameof(IndexPageRequested)),
    "GET request for Index page");
```

结构化日志记录存储可以使用事件名称(当它获得事件 ID 时)来丰富日志记录。例如, [Serilog](#) 使用该事件名称。

通过强类型扩展方法调用 `Action`。`IndexPageRequested` 方法在示例应用中记录索引页 GET 请求的消息:

```
public static void IndexPageRequested(this ILogger logger)
{
    _indexPageRequested(logger, null);
}
```

在 Pages/Index.cshtml.cs 的 `OnGetAsync` 方法中, 在记录器上调用 `IndexPageRequested`:

```
public async Task OnGetAsync()
{
    _logger.IndexPageRequested();

    Quotes = await _db.Quotes.AsNoTracking().ToListAsync();
}
```

检查应用的控制台输出:

```
info: LoggerMessageSample.Pages.IndexModel[1]
=> RequestId:0HL90M6E7PHK4:00000001 RequestPath:/ => /Index
GET request for Index page
```

要将参数传递给日志消息, 创建静态字段时最多定义六种类型。通过为 `Action` 字段定义 `string` 类型来添加引号时, 示例应用会记录一个字符串:

```
private static readonly Action<ILogger, string, Exception> _quoteAdded;
```

委托的日志消息模板从提供的类型接收其占位符值。示例应用定义一个委托, 用于在 quote 参数是 `string` 的位置添加引号:

```
_quoteAdded = LoggerMessage.Define<string>(
    LogLevel.Information,
    new EventId(2, nameof(QuoteAdded)),
    "Quote added (Quote = '{Quote}')");
```

用于添加引号的静态扩展方法 `QuoteAdded` 接收 quote 参数值并将其传递给 `Action` 委托:

```
public static void QuoteAdded(this ILogger logger, string quote)
{
    _quoteAdded(logger, quote, null);
}
```

在索引页的页面模型 (Pages/Index.cshtml.cs) 中, 调用 `QuoteAdded` 来记录消息:

```

public async Task<IActionResult> OnPostAddQuoteAsync()
{
    _db.Quotes.Add(Quote);
    await _db.SaveChangesAsync();

    _logger.QuoteAdded(Quote.Text);

    return RedirectToAction();
}

```

检查应用的控制台输出：

```

info: LoggerMessageSample.Pages.IndexModel[2]
      => RequestId:0HL90M6E7PHK5:0000000A RequestPath:/ => /Index
      Quote added (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding reality.
      - Ayn Rand')

```

本示例应用实现用于删除引号的 `try - catch` 模式。为成功的删除操作记录提示性信息。引发异常时，为删除操作记录错误消息。针对未成功的删除操作，日志消息包括异常堆栈跟踪 (Internal/LoggerExtensions.cs)：

```

private static readonly Action<ILogger, string, int, Exception> _quoteDeleted;
private static readonly Action<ILogger, int, Exception> _quoteDeleteFailed;

_quoteDeleted = LoggerMessage.Define<string, int>(
    LogLevel.Information,
    new EventId(4, nameof(QuoteDeleted)),
    "Quote deleted (Quote = '{Quote}' Id = {Id})");

_quoteDeleteFailed = LoggerMessage.Define<int>(
    LogLevel.Error,
    new EventId(5, nameof(QuoteDeleteFailed)),
    "Quote delete failed (Id = {Id})");

```

请注意异常如何传递到 `QuoteDeleteFailed` 中的委托：

```

public static void QuoteDeleted(this ILogger logger, string quote, int id)
{
    _quoteDeleted(logger, quote, id, null);
}

public static void QuoteDeleteFailed(this ILogger logger, int id, Exception ex)
{
    _quoteDeleteFailed(logger, id, ex);
}

```

在索引页的页面模型中，成功删除引号时会在记录器上调用 `QuoteDeleted` 方法。如果找不到要删除的引号，则会引发 `ArgumentNullException`。通过 `try - catch` 语句捕获异常，并在 `catch` 块 (Pages/Index.cshtml.cs) 中调用记录器上的 `QuoteDeleteFailed` 方法来记录异常：

```

public async Task<IActionResult> OnPostDeleteQuoteAsync(int id)
{
    var quote = await _db.Quotes.FindAsync(id);

    // DO NOT use this approach in production code!
    // You should check quote to see if it's null before removing
    // it and saving changes to the database. A try-catch is used
    // here for demonstration purposes of LoggerMessage features.
    try
    {
        _db.Quotes.Remove(quote);
        await _db.SaveChangesAsync();

        _logger.QuoteDeleted(quote.Text, id);
    }
    catch (ArgumentNullException ex)
    {
        _logger.QuoteDeleteFailed(id, ex);
    }

    return RedirectToAction();
}

```

成功删除引号时，检查应用的控制台输出：

```

info: LoggerMessageSample.Pages.IndexModel[4]
=> RequestId:0HL90M6E7PHK5:00000016 RequestPath:/ => /Index
    Quote deleted (Quote = 'You can avoid reality, but you cannot avoid the consequences of avoiding
    reality. - Ayn Rand' Id = 1)

```

引号删除失败时，检查应用的控制台输出。请注意，异常包括在日志消息中：

```

fail: LoggerMessageSample.Pages.IndexModel[5]
=> RequestId:0HL90M6E7PHK5:00000010 RequestPath:/ => /Index
    Quote delete failed (Id = 999)
System.ArgumentNullException: Value cannot be null.
Parameter name: entity
    at Microsoft.EntityFrameworkCore.Utilities.Check.NotNull[T](T value, String parameterName)
    at Microsoft.EntityFrameworkCore.DbContext.Remove[TEntity](TEntity entity)
    at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.Remove(TEntity entity)
    at LoggerMessageSample.Pages.IndexModel.<OnPostDeleteQuoteAsync>d__14.MoveNext() in
        <PATH>\sample\Pages\Index.cshtml.cs:line 87

```

LoggerMessage.DefineScope

[DefineScope\(字符串\)](#) 创建一个用于定义 [日志作用域](#) 的 `Func` 委托。`DefineScope` 重载允许向命名的格式字符串（模板）传递最多三个类型参数。

`Define` 方法也一样，提供给 `DefineScope` 方法的字符串是一个模板，而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称，建议使用[帕斯卡拼写法](#)。例如：`{Count}`、`{FirstName}`。

使用 `DefineScope(字符串)` 方法定义一个 [日志作用域](#)，以应用到一系列日志消息中。

示例应用含有一个“全部清除”按钮，用于删除数据库中的所有引号。通过一次删除一个引号来将其删除。每当删除一个引号时，都会在记录器上调用 `QuoteDeleted` 方法。在这些日志消息中会添加一个日志作用域。

在控制台记录器选项中启用 `IncludeScopes`：

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging((hostingContext, logging) =>
    {
        // Setting options.IncludeScopes is required in ASP.NET Core 2.0
        // apps. Setting IncludeScopes via appsettings configuration files
        // is a feature that's planned for the ASP.NET Core 2.1 release.
        // See: https://github.com/aspnet/logging/pull/706
        logging.AddConsole(options => options.IncludeScopes = true);
    })
    .Build();
}
```

在 ASP.NET Core 2.0 应用中需要设置 `IncludeScopes` 来启用日志作用域。通过 `appsettings` 配置文件来设置 `IncludeScopes` 是针对 ASP.NET Core 2.1 版本计划的一项功能。

示例应用清除其他提供程序并添加筛选器来减少日志记录输出。这样便可更加轻松地查看演示 `LoggerMessage` 功能的示例的日志消息。

要创建日志作用域，请添加一个字段来保存该作用域的 `Func` 委托。示例应用创建一个名为 `_allQuotesDeletedScope` (`Internal/LoggerExtensions.cs`) 的字段：

```
private static Func<ILogger, int, IDisposable> _allQuotesDeletedScope;
```

使用 `DefineScope` 来创建委托。调用委托时最多可以指定三种类型作为模板参数使用。示例应用使用包含删除的引号数量的消息模板 (`int` 类型)：

```
_allQuotesDeletedScope = LoggerMessage.DefineScope<int>("All quotes deleted (Count = {Count})");
```

为日志消息提供一种静态扩展方法。包含已命名属性的任何类型参数 (这些参数出现在消息模板中)。示例应用采用引号的 `count`，以删除并返回 `_allQuotesDeletedScope`：

```
public static IDisposable AllQuotesDeletedScope(this ILogger logger, int count)
{
    return _allQuotesDeletedScope(logger, count);
}
```

该作用域将日志记录扩展调用包装在 `using` 块中：

```
public async Task<IActionResult> OnPostDeleteAllQuotesAsync()
{
    var quoteCount = await _db.Quote.CountAsync();

    using (_logger.AllQuotesDeletedScope(quoteCount))
    {
        foreach (Quote quote in _db.Quote)
        {
            _db.Quote.Remove(quote);

            _logger.QuoteDeleted(quote.Text, quote.Id);
        }
        await _db.SaveChangesAsync();
    }

    return RedirectToAction();
}
```

检查应用控制台输出中的日志消息。以下结果显示删除的三个引号，以及包含的日志作用域消息：

```
info: LoggerMessageSample.Pages.IndexModel[4]
=> RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
Quote deleted (Quote = 'Quote 1' Id = 2)
info: LoggerMessageSample.Pages.IndexModel[4]
=> RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
Quote deleted (Quote = 'Quote 2' Id = 3)
info: LoggerMessageSample.Pages.IndexModel[4]
=> RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index => All quotes deleted (Count = 3)
Quote deleted (Quote = 'Quote 3' Id = 4)
```

请参阅

- [日志记录](#)

处理 ASP.NET Core 中的错误

2018/5/14 • 5 min to read • [Edit Online](#)

作者: Steve Smith 和 Tom Dykstra

本文介绍了处理 ASP.NET Core 应用中常见错误的一些方法。

[查看或下载示例代码\(如何下载\)](#)

开发人员异常页

如需在应用中配置可以显示异常详细信息的页面, 请先安装 `Microsoft.AspNetCore.Diagnostics` NuGet 包并以下代码行添加到 Startup 类 [Startup 类配置方法](#):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

`UseDeveloperExceptionPage` 需要放在所有你想要捕获异常的中间件声明之前, 如 `app.UseMvc`。

警告

仅当应用程序在开发环境中运行时才启用开发人员异常页。否则当应用程序在生产环境中运行时, 详细的异常信息会向公众泄露。[了解环境配置](#)。

若要查看开发人员异常页, 可使用 `Development` 环境来运行应用程序, 并在基础 URL 后添加 `?throw=true`。该页面包括几个选项卡, 这些选项卡中包含关于异常和请求的信息。第一个选项卡包括堆栈跟踪。

An unhandled exception occurred while processing the request.

Exception: Exception triggered!

```
ErrorHandlingSample.Startup+<>c+<<Configure>b_1_4>d.MoveNext() in Startup.cs, line 67
```

Stack **Query** Cookies Headers

Exception: Exception triggered!

```
ErrorHandlingSample.Startup+<>c+<<Configure>b_1_4>d.MoveNext() in Startup.cs
+    67.           var builder = new StringBuilder();
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Threading.Tasks.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
```

```
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware+<Invoke>d_7.MoveNext()
```

[Show raw exception details](#)

下一个 **Query** 选项卡将显示查询字符串参数信息(如有)。

An unhandled exception occurred while processing the request.

Exception: Exception triggered!

```
ErrorHandlingSample.Startup+<>c+<<Configure>b_1_4>d.MoveNext() in Startup.cs, line 67
```

Stack **Query** Cookies Headers

Variable	Value
throw	true

示例请求没有任何 cookie, 但如果有的话, 它们将出现在 **Cookies** 选项卡。最后一个选项卡显示请求的标头。

An unhandled exception occurred while processing the request.

Exception: Exception triggered!

ErrorHandlingSample.Startup+<>c+<<Configure>b_1_4>d.MoveNext() in Startup.cs, line 67

Stack	Query	Cookies	Headers
Variable	Value		
Accept	text/html, application/xhtml+xml, image/jxr, */*		
Accept-Encoding	gzip, deflate		
Accept-Language	en-US		
Connection	Keep-Alive		
Host	localhost:13930		
MS-ASPARTCORE-TOKEN	55ddb2cf-3cf5-4734-afa9-7abcf38839f		
Referer	http://localhost:13930/		
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.79 Safari/537.36 Edge/14.14393		
X-Original-For	127.0.0.1:20566		
X-Original-Proto	http		

配置自定义异常处理页

最好配置一个当应用在非 `Development` 环境中运行时可以使用的异常处理页。

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

在 MVC 应用程序中，不要使用 HTTP 谓词 Attribute(如 `HttpGet`)来显式修饰错误处理程序操作方法。使用显式谓词会阻止某些请求访问方法。

```
[Route("/Error")]
public IActionResult Index()
{
    // Handle error here
}
```

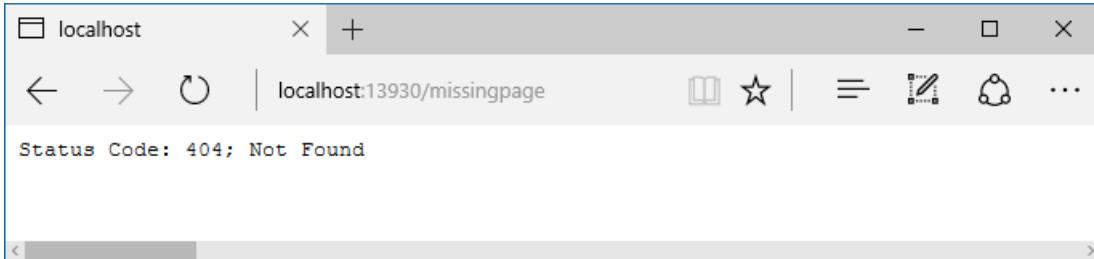
配置状态代码页

默认情况下，应用不会为 HTTP 状态代码提供丰富状态代码页，例如 404 未找到。要提供状态代码页，请通过向

`Startup.Configure` 方法添加行来配置状态代码页中间件：

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

默认情况下，状态代码页中间件为常见状态代码(如 404)添加简单的纯文本处理程序：



该中间件支持多种扩展方法。一种方法采用 Lambda 表达式：

```
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";
    await context.HttpContext.Response.WriteAsync(
        "Status code page, status code: " +
        context.HttpContext.Response.StatusCode);
});
```

另一种方法采用内容类型和格式字符串：

```
app.UseStatusCodePages("text/plain", "Status code page, status code: {0}");
```

也可以使用重定向和重新执行扩展方法。重定向方法向客户端发送 302 状态代码：

```
app.UseStatusCodePagesWithRedirects("/error/{0}");
```

重新执行方法将原始状态代码返回到客户端，但同时还会执行重定向 URL 的处理程序：

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

可禁用 Razor 页处理程序方法或 MVC 控制器中的特定请求的状态代码页。要禁用状态代码页，请尝试从请求的 `HttpContext.Features` 集合中检索 `IStatusCodePagesFeature`，并在功能可用时禁用该功能：

```
var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();

if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

异常处理代码

异常处理页中的代码可能会引发异常。建议在生产错误页面中包含纯静态内容。

此外还要注意，一旦发送响应的标头，则无法更改响应的状态代码，也不能运行任何异常页或处理程序。必须完成响应或中止连接。

服务器异常处理

除应用程序中的异常处理逻辑外，托管应用程序的服务器[服务器](#)也会执行一些异常处理。如果服务器在发送标头之前捕获异常，服务器将发送没有正文的 500 内部服务器错误响应。如果服务器在已发送标头后捕获异常，服务器会关闭连接。应用程序无法处理的请求将由服务器进行处理。发生的任何异常都将由服务器进行处理。任何配置的自定义错误页面或异常处理中间件或筛选器都不会影响此行为。

启动异常处理

应用程序启动期间发生的异常仅可在承载层进行处理。可以使用 `captureStartupErrors` 和 `detailedErrors` 键配置主机在响应启动期间的错误时的行为方式。

仅当捕获的启动错误发生在主机地址/端口绑定之后，承载层才会为该错误显示错误页。如果绑定因任何原因而失败，则承载层会记录关键异常，dotnet 进程崩溃，且在 Kestrel 服务器上运行应用时，不会显示任何错误页。

在 IIS 或 IIS Express 上运行应用时，如果无法启动进程，ASP.NET Core 模块将返回 502.5 进程失败。请按照[对 IIS 上的 ASP.NET Core 进行故障排除](#)主题中的故障排除建议进行操作。

ASP.NET MVC 错误处理

MVC 应用还有一些其他的错误处理选项，例如配置异常筛选器和执行模型验证。

异常筛选器

在 MVC 应用中，异常筛选器可以进行全局配置，也可以为每个控制器或每个操作单独配置。这些筛选器处理在执行控制器操作或其他筛选器时出现的任何未处理的异常，且不会以其他方式调用。通过[筛选器](#)详细了解异常筛选器。

提示

异常筛选器非常适用于捕获 MVC 操作内出现的异常，但灵活性不如错误处理中间件。一般情况下建议使用中间件；仅在需要根据所选 MVC 操作以不同方式执行错误处理时，才使用筛选器。

处理模型状态错误

[模型验证](#) 在每个控制器操作被调用之前发生，操作方法负责检查 `ModelState.IsValid` 并相应地作出反应。

某些应用会使用标准约定来处理模型验证错误，在这种情况下，使用[筛选器](#)可以更好地实施这种策略。你需要使用无效模型状态测试操作的行为。查看[测试控制器逻辑](#)了解详情。

ASP.NET Core 中的文件提供程序

2018/5/8 • 7 min to read • [Edit Online](#)

作者: [Steve Smith](#)

ASP.NET Core 通过文件提供程序来抽象化文件系统访问。

[查看或下载示例代码\(如何下载\)](#)

文件提供程序抽象

文件提供程序是对文件系统的抽象。主接口是 `IFileProvider`。`IFileProvider` 公开一些方法以获取文件信息 (`IFileInfo`)、目录信息 (`IDirectoryContents`)，以及设置更改通知(使用 `IChangeToken`)。

`IFileInfo` 提供有关单个文件或目录的方法和属性。它有两个布尔属性 (`Exists` 和 `IsDirectory`)，以及描述文件 `Name``Length` (以字节为单位)、`LastModified` 日期的属性。可使用其 `CreateReadStream` 方法从文件中读取。

文件提供程序实现

`IFileProvider` 的三种实现可用:物理、嵌入和复合。物理提供程序用于访问实际系统的文件。嵌入式提供程序用于访问嵌入在程序集中的文件。复合式提供程序提供对一个或多个其他提供程序中的文件和目录的合并访问。

PhysicalFileProvider

`PhysicalFileProvider` 提供对物理文件系统的访问。它包装 `System.IO.File` 类型(针对物理提供程序)，将所有路径范围限定在一个目录及其子目录中。此范围限制对特定目录及其子目录的访问，从而阻止对该边界外的文件系统的访问。实例化此提供程序时，用户必须向其提供一个目录路径，它可作为对该提供程序的所有请求的基本路径 (并且限制此路径之外的访问)。在 ASP.NET Core 应用中，可以直接实例化 `PhysicalFileProvider` 提供程序，也可以通过 [依赖关系注入](#)在控制器或服务的构造函数中请求 `IFileProvider`。后一种方法生成的解决方案通常更具灵活性和可测试性。

以下示例演示如何创建 `PhysicalFileProvider`。

```
IFileProvider provider = new PhysicalFileProvider(applicationRoot);
IDirectoryContents contents = provider.GetDirectoryContents(""); // the applicationRoot contents
IFileInfo fileInfo = provider.GetFileInfo("wwwroot/js/site.js"); // a file under applicationRoot
```

你可以循环访问其目录内容，或通过提供子路径来获取特定文件的信息。

若要从控制器请求提供程序，请在控制器的构造函数中指定该提供程序，并将其分配给本地字段。使用操作方法中的本地实例：

```

public class HomeController : Controller
{
    private readonly IFileProvider _fileProvider;

    public HomeController(IFileProvider fileProvider)
    {
        _fileProvider = fileProvider;
    }

    public IActionResult Index()
    {
        var contents = _fileProvider.GetDirectoryContents("");
        return View(contents);
    }
}

```

然后, 在应用的 `Startup` 类中创建提供程序:

```

using System.Linq;
using System.Reflection;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.FileProviders;
using Microsoft.Extensions.Logging;

namespace FileProviderSample
{
    public class Startup
    {
        private IHostingEnvironment _hostingEnvironment;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();
            Configuration = builder.Build();

            _hostingEnvironment = env;
        }

        public IConfigurationRoot Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            // Add framework services.
            services.AddMvc();

            var physicalProvider = _hostingEnvironment.ContentRootFileProvider;
            var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
            var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);

            // choose one provider to use for the app and register it
            //services.AddSingleton<IFileProvider>(physicalProvider);
            //services.AddSingleton<IFileProvider>(embeddedProvider);
            services.AddSingleton<IFileProvider>(compositeProvider);
        }
}

```

在 `Index.cshtml` 视图中, 循环访问提供的 `IDirectoryContents`:

```

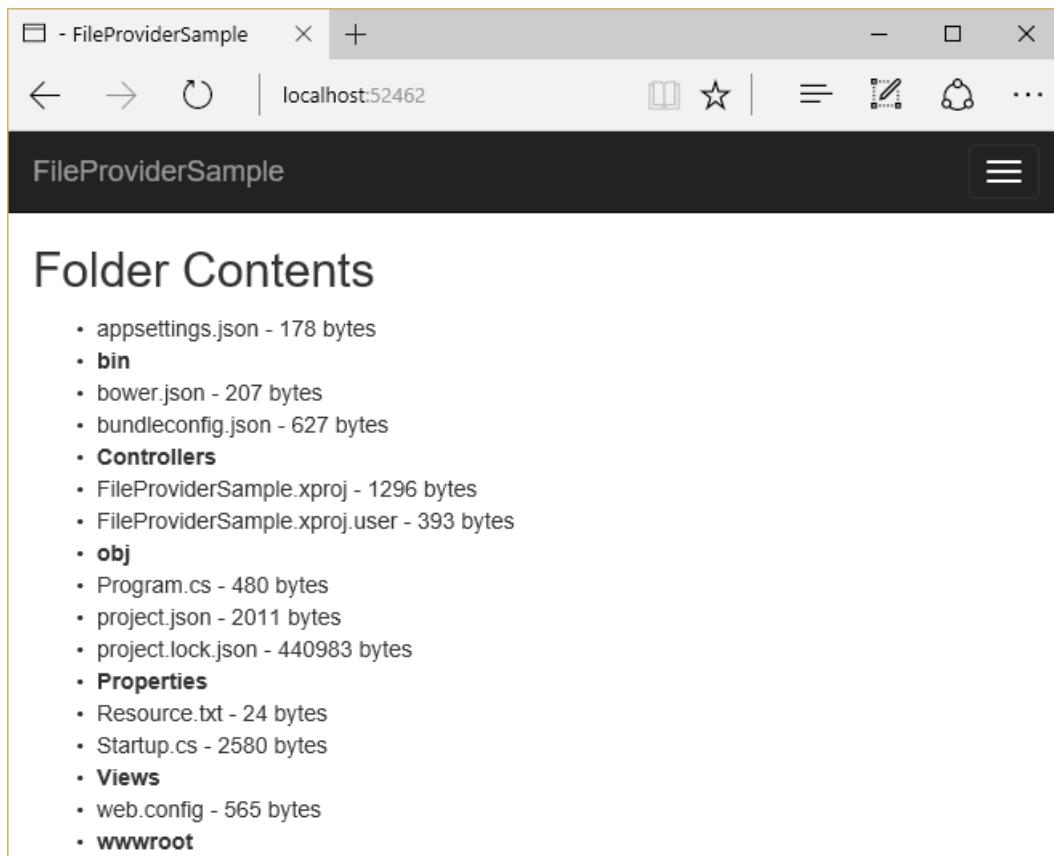
@using Microsoft.Extensions.FileProviders
@model IDirectoryContents

<h2>Folder Contents</h2>

<ul>
    @foreach (IFileInfo item in Model)
    {
        if (item.IsDirectory)
        {
            <li><strong>@item.Name</strong></li>
        }
        else
        {
            <li>@item.Name - @item.Length bytes</li>
        }
    }
</ul>

```

结果：



EmbeddedFileProvider

`EmbeddedFileProvider` 用于访问嵌入在程序集中的文件。在 .NET Core 中，通过 `.csproj` 文件中的 `<EmbeddedResource>` 元素将各个文件嵌入程序集中：

```

<ItemGroup>
    <EmbeddedResource Include="Resource.txt;**\*.js"
        Exclude="bin\**;obj\**;**\*.xproj;packages\**;@(EmbeddedResource)" />
    <Content Update="wwwroot\**\*;Views\**\*;Areas\**\Views;appsettings.json;web.config">
        <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </Content>
</ItemGroup>

```

在指定要嵌入到程序集中的文件时，你可以使用通配模式。这些模式可用于匹配一个或多个文件。

注意

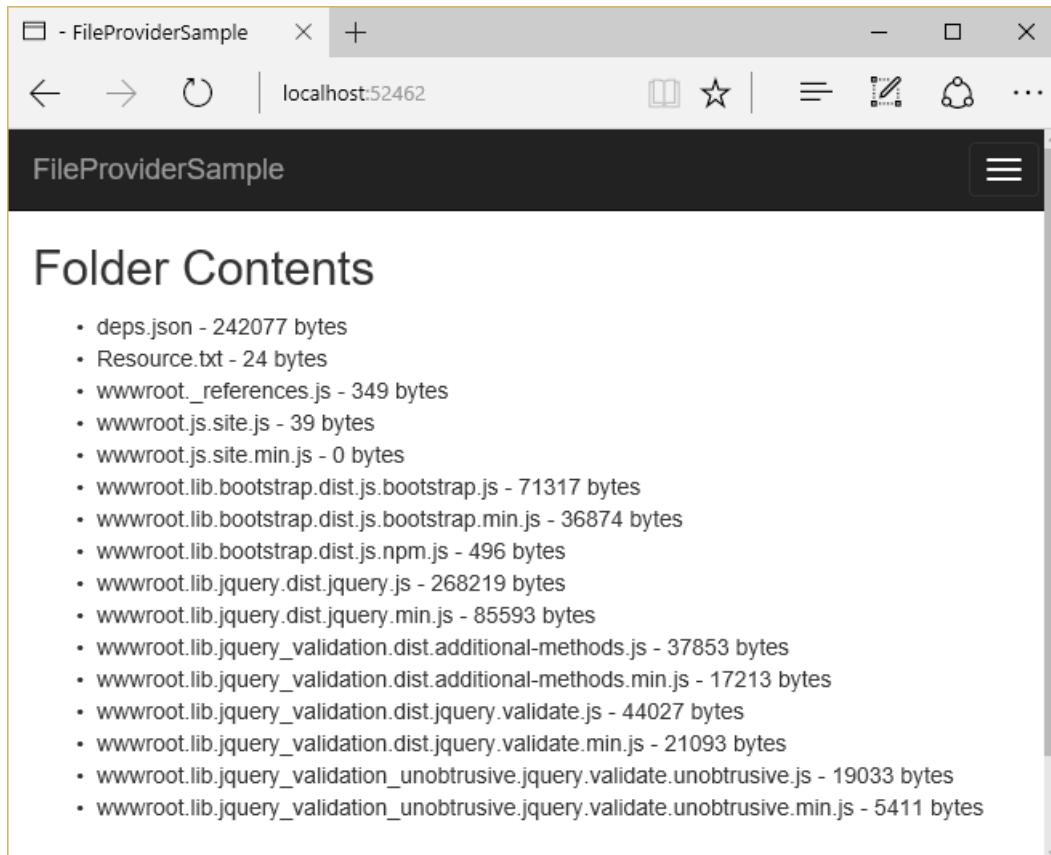
实际上，用户不太可能会将项目中所有的 .js 文件都嵌入其程序集中；以上示例仅供演示。

创建 `EmbeddedFileProvider` 时，将它要读取的程序集传递给其构造函数。

```
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
```

以上片段演示如何创建可访问当前执行的程序集的 `EmbeddedFileProvider`。

更新要使用 `EmbeddedFileProvider` 的示例应用会导致以下输出：



注意

嵌入的资源不会公开目录。而是使用 `.` 分隔符将指向资源的路径（通过其命名空间）嵌入其文件名中。

提示

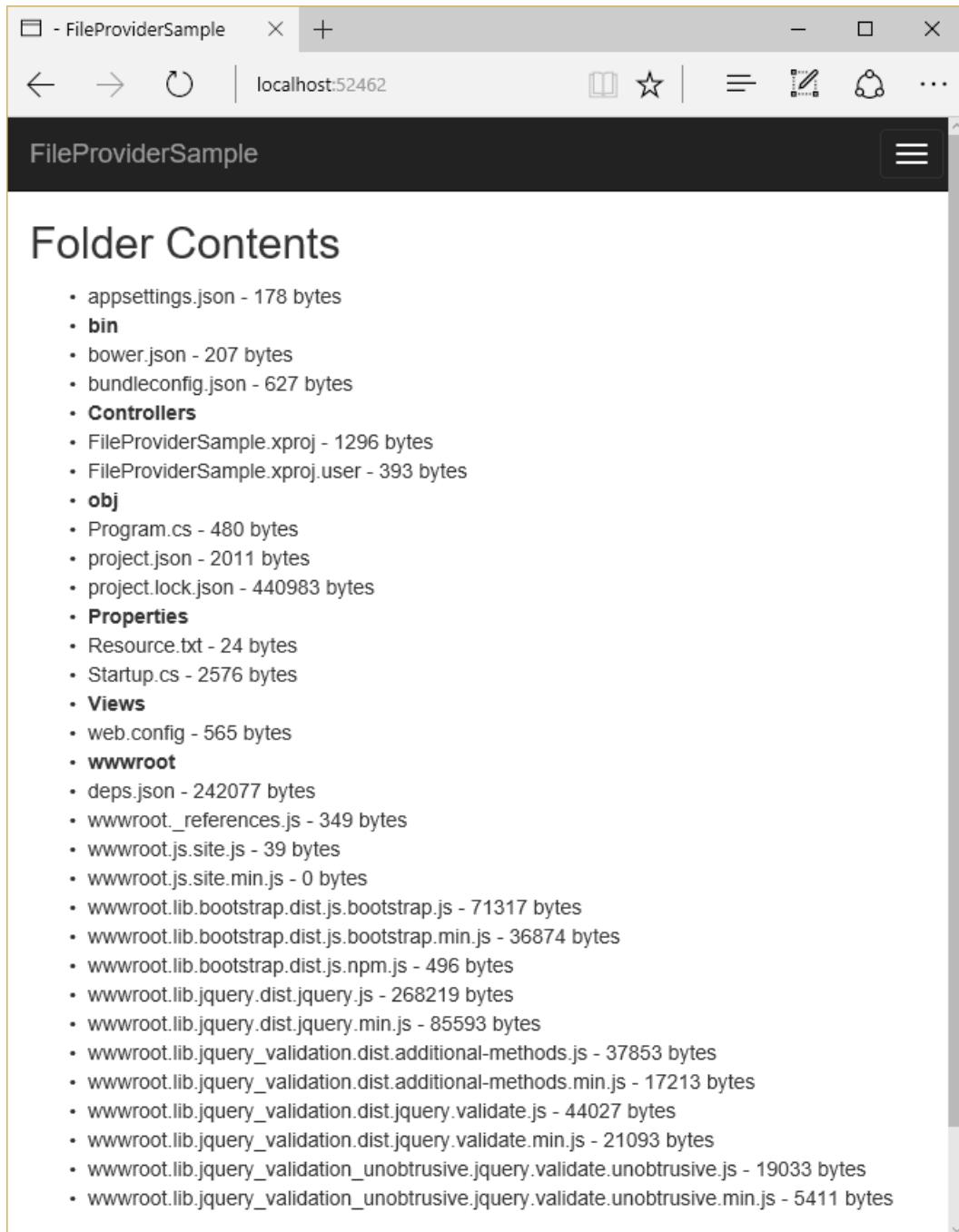
`EmbeddedFileProvider` 构造函数接受一个可选的 `baseNamespace` 参数。指定此操作会将到 `GetDirectoryContents` 的调用范围限制为提供的名称空间下的那些资源。

CompositeFileProvider

`CompositeFileProvider` 合并 `IFileProvider` 实例，以便公开一个接口来处理多个提供程序中的文件。创建 `CompositeFileProvider` 时，将一个或多个 `IFileProvider` 实例传递给其构造函数：

```
var physicalProvider = _hostingEnvironment.ContentRootFileProvider;
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);
```

更新要使用 `CompositeFileProvider` 的示例应用(该应用包括之前配置的物理和嵌入式提供程序)会导致以下输出:



监视更改

`IFileProvider` `Watch` 方法提供一种方法来监视一个或多个文件或目录的更改。此方法接受一个路径字符串，该字符串可使用[通配模式](#)来指定多个文件，并返回一个 `IChangeToken`。此令牌公开可被检查的 `HasChanged` 属性，以及在检测到对指定路径字符串进行更改时调用的 `RegisterChangeCallback` 方法。请注意，每个更改令牌仅调用其关联的回调来响应单个更改。若要启用持续监视，可以使用如下所示的 `TaskCompletionSource`，或者重新创建 `IChangeToken` 实例来响应更改。

在本文的示例中，控制台应用程序被配置为每当在修改文本文件时显示消息：

```

private static PhysicalFileProvider _fileProvider =
    new PhysicalFileProvider(Directory.GetCurrentDirectory());

public static void Main(string[] args)
{
    Console.WriteLine("Monitoring quotes.txt for changes (Ctrl-c to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    IChangeToken token = _fileProvider.Watch("quotes.txt");
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("quotes.txt changed");
}

```

多次保存文件之后的结果：

```

C:\Windows\System32\cmd.exe - dotnet run

>dotnet run
Project WatchConsole (.NETCoreApp,Version=v1.0) will be compiled because inputs were modified
Compiling WatchConsole for .NETCoreApp,Version=v1.0

Compilation succeeded.
  0 Warning(s)
  0 Error(s)

Time elapsed 00:00:01.6533960

Monitoring quotes.txt for changes (ctrl-c to quit)...
quotes.txt changed
quotes.txt changed
quotes.txt changed
quotes.txt changed
quotes.txt changed

```

注意

某些文件系统(例如 Docker 容器和网络共享)可能无法可靠地发送更改通知。将 `DOTNET_USE_POLLINGFILEWATCHER` 环境变量设置为 `1` 或 `true`，以便每 4 秒对文件系统的更改进行轮询。

通配模式

文件系统路径使用称作“通配模式”的通配符模式。这些简单的模式可以用来指定文件组。这两个通配符为 `*` 和 `**`。

`*`

匹配位于当前文件夹级别的任何内容、任何文件名或任何文件扩展名。匹配由文件路径中的 `/` 和 `.` 字符终止。

`**`

匹配多个目录级别的任何内容。可用于以递归方式匹配目录层次结构中的许多文件。

通配模式示例

`directory/file.txt`

匹配特定目录中的特定文件。

`directory/*.txt`

匹配特定目录中带有 `.txt` 扩展名的所有文件。

`directory/*/bower.json`

匹配 `directory` 目录的下一级目录中的所有 `bower.json` 文件。

`directory/**/*.txt`

匹配在 `directory` 目录下带有 `.txt` 扩展名所有文件。

ASP.NET Core 中的文件提供程序使用情况

ASP.NET Core 的多个部分使用文件提供程序。`IHostingEnvironment` 将应用的内容根和 Web 根作为 `IFileProvider` 类型公开。静态文件中间件使用文件提供程序来查找静态文件。Razor 在查找视图时大量使用 `IFileProvider`。Dotnet 的发布功能使用文件提供程序和通配模式来指定应该发布哪些文件。

在应用中使用的建议

如果 ASP.NET Core 应用需要文件系统访问权限，那么你可以通过依赖关系注入请求 `IFileProvider` 的实例，然后使用其方法执行访问操作（如此示例中所示）。这样在应用启动时配置一次提供程序即可，而且可以减少应用实例化的实现类型数量。

4 min to read •

ASP.NET Core 中的会话和应用程序状态

2018/5/14 • 15 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Steve Smith](#) 和 [Diana LaRose](#)

HTTP 是无状态的协议。Web 服务器将每个 HTTP 请求视为独立的请求，并不会保留以前请求中的用户值。本文讨论保留请求之间的应用程序和会话状态的不同方式。

会话状态

会话状态是 ASP.NET Core 中的一项功能，可用于在用户浏览 Web 应用时保存和存储用户数据。会话状态由服务器上的字典或哈希表组成，在浏览器的请求中保持数据。会话数据由缓存支持。

ASP.NET Core 通过向客户端提供包含会话 ID 的 Cookie 来维护会话状态，该会话 ID 与每个请求一起发送到服务器。服务器使用会话 ID 来获取会话数据。因为会话 Cookie 是特定于浏览器的，所以不能跨浏览器中共享会话。仅当浏览器会话结束时才能删除会话 Cookie。如果收到过期的会话 Cookie，则创建使用相同会话 Cookie 的新会话。

服务器在上次请求后保留会话的时间有限。设置会话超时，或者使用 20 分钟的默认值。会话状态非常适合用于存储特定于特定会话但并不需要永久保留的用户数据。在调用 `Session.Clear` 时或数据存储中会话到期时将删除后备存储中的数据。服务器不知道关闭浏览器或删除会话 Cookie 的时间。

警告

请勿将敏感数据存储在会话中。客户端可能不会关闭浏览器并清除会话 Cookie（某些浏览器在多个窗口中保持会话 Cookie）。另外，会话可能不限于单个用户；下一个用户可能继续同一个会话。

内存中会话提供程序将会话数据存储在本地服务器上。如果计划在服务器场上运行 Web 应用，则必须使用粘性会话将每个会话绑定到特定的服务器上。Microsoft Azure 网站平台默认设置为粘性会话（应用程序请求路由或 ARR）。然而，粘性会话可能会影响可伸缩性，并使 Web 应用更新变得复杂。更好的选择是使用 Redis 或 SQL Server 分布式缓存，它们不需要粘性会话。有关详细信息，请参阅[使用分布式缓存](#)。有关服务提供程序设置的详细信息，请参阅本文后续部分中的[配置会话](#)。

TempData

ASP.NET Core MVC 在[控制器](#)上公开了 `TempData` 属性。此属性存储未读取的数据。`Keep` 和 `.Peek` 方法可用于检查数据，而不执行删除。多个请求需要数据时，`TempData` 非常有助于进行重定向。通过 `TempData` 提供程序实现 `TempData`，例如，使用 Cookie 或会话状态。

TempData 提供程序

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

ASP.NET Core 2.0 及更高版本中，基于 Cookie 的 `TempData` 提供程序在默认情况下使用，将 `TempData` 存储在 Cookie 中。

使用 `Base64UrlTextEncoder` 对 Cookie 数据进行编码。因为 Cookie 被加密并分块，所以 ASP.NET Core 1.x 中的单个 Cookie 大小限制不适用。未压缩 Cookie 数据，因为压缩加密的数据会导致安全问题，如 `CRIME` 和 `BREACH` 攻击。有关基于 Cookie 的 `TempData` 提供程序的详细信息，请参阅 [CookieTempDataProvider](#)。

选择 TempData 提供程序

选择 TempData 提供程序涉及几个注意事项，例如：

1. 应用程序是否已经将会话状态用于其他目的？如果是，使用会话状态 TempData 提供程序对应用程序没有额外的成本（除了数据的大小）。
2. 应用程序是否只对相对较小的数据量（最多 500 个字节）使用 TempData？如果是，Cookie TempData 提供程序将为每个携带 TempData 的请求增加较小的成本。如果不是，会话状态 TempData 提供程序有助于在使用 TempData 前，避免在每个请求中来回切换大量数据。
3. 应用程序是否在 Web 场（多个服务器）中运行？如果是，则使用 Cookie TempData 提供程序无需额外配置。

注意

大多数 Web 客户端（如 Web 浏览器）针对每个 Cookie 的最大大小和/或 Cookie 总数强制实施限制。因此，使用 Cookie TempData 提供程序时，请验证应用程序未超过这些限制。请考虑数据的总大小，将加密和分块的开销考虑在内。

配置 TempData 提供程序

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

默认情况下启用基于 Cookie 的 TempData 提供程序。以下 `Startup` 类代码配置基于会话的 TempData 提供程序：

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

    services.AddSession();
}

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

排序对于中间件组件至关重要。在前面的示例中，在 `UseMvcWithDefaultRoute` 之后调用 `UseSession` 时会发生 `InvalidOperationException` 类型的异常。有关详细信息，请参阅[中间件排序](#)。

重要事项

如果面向 .NET Framework 和使用基于会话的提供程序，将 [Microsoft.AspNetCore.Session](#) NuGet 包添加到项目。

查询字符串

可以将有限的数据从一个请求传递到另一个请求，方法是将其添加到新请求的查询字符串中。这有利于以一种持久的方式捕获状态，这种方式允许通过电子邮件或社交网络共享嵌入式状态的链接。但是，为此，不可将查询字符串用于敏感数据。在查询字符串中包含数据除了易于共享，还为[跨站点请求伪造 \(CSRF\)](#) 攻击创造了机会，可以欺骗用户在通过身份验证时访问恶意网站。然后，攻击者可以从应用程序中窃取用户数据，或者代表用户采取恶意操作。任何保留的应用程序或会话状态必须防止 CSRF 攻击。有关 CSRF 攻击的详细信息，请参阅[预防跨网站请求伪造 \(XSRF/CSRF\) 攻击](#)。

后期数据和隐藏字段

数据可以保存在隐藏的表单域中，并在下一个请求上回发。这在多页窗体中很常见。但是，由于客户端可能会篡

改数据，因此服务器必须始终重新验证数据。

Cookie

Cookie 提供了一种在 Web 应用程序中存储用户特定数据的方法。因为 Cookie 是随每个请求发送的，所以它们的大小应该保持在最低限度。理想情况下，仅标识符应存储在 Cookie 中，而实际数据存储在服务器上。大多数浏览器将 Cookie 限制为 4096 个字节。此外，每个域仅有有限数量的 Cookie 可用。

因为 Cookie 易被篡改，所以它们必须在服务器上进行验证。尽管在客户端的 Cookie 的持续性是受用户干预并到期，但它们通常是客户端上最持久的数据持久形式。

Cookie 通常用于个性化设置，其中的内容是为已知用户定制的。因为在大多数情况下，用户仅被标识且未经过身份验证，所以通常可以通过在 Cookie 中存储用户名、帐户名或唯一用户 ID（例如 GUID）来保护 Cookie。然后可以使用 Cookie 来访问站点的用户个性化设置基础结构。

HttpContext.Items

`Items` 集合是存储仅在处理一个特定请求时才需要的数据的理想位置。集合的内容在每次请求后被放弃。`Items` 集合最好用作组件或中间件在请求期间在不同时间点操作且没有直接传递参数的方法时进行通信的方式。有关详细信息，请参阅本文后面的[使用 HttpContext.Items](#)。

缓存

缓存是存储和检索数据的有效方法。可以根据时间和其他注意事项控制缓存项的生存期。了解有关[如何缓存](#)的详细信息。

使用会话状态

配置会话

`Microsoft.AspNetCore.Session` 包提供用于管理会话状态的中间件。若要启用会话中间件，`Startup` 必须包含：

- 任一 `IDistributedCache` 内存缓存。`IDistributedCache` 实现用作会话后备存储。
- `AddSession` 调用，要求 NuGet 包“`Microsoft.AspNetCore.Session`”。
- `UseSession` 调用。

下面的代码演示如何设置内存中的会话提供程序。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

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

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

        // Adds a default in-memory implementation of IDistributedCache.
        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            // Set a short timeout for easy testing.
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.Cookie.HttpOnly = true;
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseSession();
        app.UseMvcWithDefaultRoute();
    }
}

```

一旦 `HttpContext` 安装并配置，可以从它引用会话。

如果在 `UseSession` 被调用前尝试访问 `Session`，将引发异常

`InvalidOperationException: Session has not been configured for this application or request。`

如果在已经开始写入 `Response` 流后尝试创建一个新 `Session`（即，未创建会话 Cookie），将引发异常

`InvalidOperationException: The session cannot be established after the response has started。` 此异常可在 Web 服务器日志中找到；它将不会在浏览器中显示。

以异步方式加载会话

只有在 `TryGetValue`、`Set` 或 `Remove` 方法之前显式调用 `ISession.LoadAsync` 方法，ASP.NET Core 中的默认会话提供程序才会从基础 `IDistributedCache` 存储以异步方式加载会话记录。如果不首先调用 `LoadAsync`，基础会话记录以同步方式加载，这可能影响应用的扩展能力。

若要让应用程序强制实施此模式，如果未在 `TryGetValue`、`Set` 或 `Remove` 之前调用 `LoadAsync` 方法，将 `DistributedSessionStore` 和 `DistributedSession` 实现包装在引起异常的版本。在服务容器中注册的已包装的版本。

实现的详细信息

会话使用 Cookie 跟踪和标识来自单个浏览器的请求。默认情况下，此 Cookie 名为“.AspNet.Session”，并使用路径“/”。因为 Cookie 默认值不指定域，所以它不提供页上的客户端脚本（因为 `CookieHttpOnly` 默认为 `true`）。

若要重写会话默认值，使用 `SessionOptions`：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

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

    // Adds a default in-memory implementation of IDistributedCache.
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.Cookie.Name = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
    });
}
```

服务器使用 `IdleTimeout` 属性来确定在放弃会话内容之前可以保持空闲的时间长短。此属性独立于 Cookie 到期时间。通过会话中间件(从会话中间件读取或写入)传递每个请求都会重置超时。

因为 `Session` 是非锁定的, 如果两个请求都试图修改会话内容, 最后一个内容会重写第一个内容。`Session` 是作为一个连贯会话实现的, 这意味着所有内容都存储在一起。正在修改会话的不同部分(不同键)的两个请求仍可能会相互影响。

设置和获取会话值

通过 `HttpContext` 的 `Session` 属性访问会话。此属性是 `ISession` 实现。

下面的示例演示了设置和获取 int 和字符串:

```
public class HomeController : Controller
{
    const string SessionKeyName = "_Name";
    const string SessionKeyYearsMember = "_YearsMember";
    const string SessionKeyDate = "_Date";

    public IActionResult Index()
    {
        // Requires using Microsoft.AspNetCore.Http;
        HttpContext.Session.SetString(SessionKeyName, "Rick");
        HttpContext.Session.SetInt32(SessionKeyYearsMember, 3);
        return RedirectToAction("SessionNameYears");
    }

    public IActionResult SessionNameYears()
    {
        var name = HttpContext.Session.GetString(SessionKeyName);
        var yearsMember = HttpContext.Session.GetInt32(SessionKeyYearsMember);

        return Content($"Name: \'{name}\', Membership years: \'{yearsMember}\'");
    }
}
```

如果添加以下扩展方法, 可以设置并获取可序列化的对象到会话:

```

using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}

```

下面的示例演示如何设置和获取可序列化的对象：

```

public IActionResult setDate()
{
    // Requires you add the Set extension method mentioned in the article.
    HttpContext.Session.Set<DateTime>(SessionKeyDate, DateTime.Now);
    return RedirectToAction("getDate");
}

public IActionResult getDate()
{
    // Requires you add the Get extension method mentioned in the article.
    var date = HttpContext.Session.Get<DateTime>(SessionKeyDate);
    var sessionTime = date.ToString();
    var currentTime = DateTime.Now.ToString();

    return Content($"Current time: {currentTime} - "
                  + $"session time: {sessionTime}");
}

```

使用 HttpContext.Items

`HttpContext` 抽象为名为 `Items` 的 `IDictionary<object, object>` 类型字典集合提供支持。此集合在 `HttpRequest` 开始时可用并在每个请求的末尾被放弃。可以通过给键控的项分配值或为特定键请求值来访问它。

在下面示例中，[中间件](#)将 `isVerified` 添加到 `Items` 集合。

```

app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});

```

在更高版本的管道中，另一个中间件无法访问它：

```

app.Run(async (context) =>
{
    await context.Response.WriteAsync("Verified request? " +
        context.Items["isVerified"]);
});

```

对于只供单个应用程序使用的中间件，`string` 键是可以接受的。但是，应用程序之间共享的中间件应使用唯一的对象键以避免键冲突的可能性。如果正在开发必须跨多个应用程序工作的中间件，使用中间件类中定义的唯一对象键，如下所示：

```
public class SampleMiddleware
{
    public static readonly object SampleKey = new Object();

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[SampleKey] = "some value";
        // additional code omitted
    }
}
```

其他代码可以使用通过中间件类公开的键访问存储在 `HttpContext.Items` 中的值：

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        string value = HttpContext.Items[SampleMiddleware.SampleKey];
    }
}
```

这种方法还具有消除代码中多个位置重复使用“神奇字符串”的优点。

应用程序状态数据

使用[依赖关系注入](#)可向所有用户提供数据：

1. 定义包含数据的服务(例如，一个名为 `MyAppData` 的类)。

```
public class MyAppData
{
    // Declare properties/methods/etc.
}
```

2. 添加服务类到 `ConfigureServices` (例如 `services.AddSingleton<MyAppData>();`)。

3. 使用每个控制器中的数据服务类：

```
public class MyController : Controller
{
    public MyController(MyAppData myService)
    {
        // Do something with the service (read some data from it,
        // store it in a private field/property, etc.)
    }
}
```

使用会话时的常见错误

- “在尝试激活‘Microsoft.AspNetCore.Session.DistributedSessionStore’时无法为类型‘Microsoft.Extensions.Caching.Distributed.IDistributedCache’解析服务。”

这通常是由于不能配置至少一个 `IDistributedCache` 实现而造成的。有关详细信息，请参阅[使用分布式缓存和内存缓存中](#)。

- 如果会话中间件无法保留会话(例如:如果数据库不可用), 它将记录并吞并异常。然后, 请求将继续正常运行, 这会导致非常难以预料的行为。

典型示例:

有人将购物篮存储在会话中。用户添加项但提交失败。应用不知道失败, 因此报告消息“已添加项”, 然而并不是如此。

检查是否存在此类错误的建议方法是完成写入到该会话后从应用代码调用 `await feature.Session.CommitAsync();`。然后就可以随意处理错误。调用 `LoadAsync` 时同样适用。

其他资源

- [ASP.NET Core 1.x:本文档中使用的代码示例](#)
- [ASP.NET Core 2.x:本文档中使用的代码示例](#)

ASP.NET Core 中的 Web 服务器实现

2018/5/17 • 6 min to read • [Edit Online](#)

作者: Tom Dykstra、Steve Smith、Stephen Halter 和 Chris Ross

ASP.NET Core 应用与进程中 HTTP 服务器实现一起运行。服务器实现侦听 HTTP 请求，并在一系列[请求功能](#)被撰写到 `HttpContext` 时将这些请求展现到应用中。

ASP.NET Core 交付两种服务器实现：

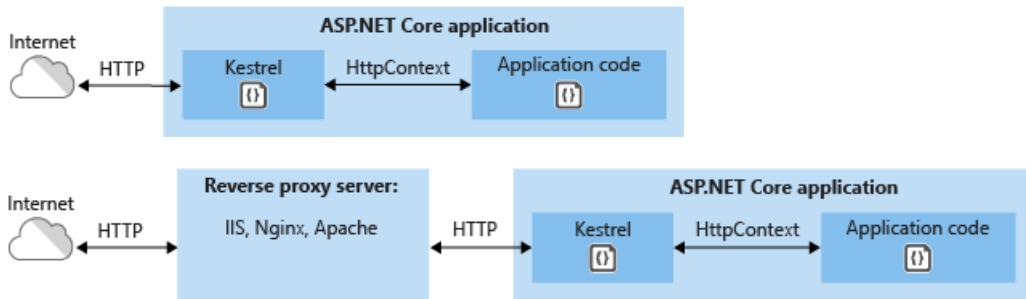
- [Kestrel](#) 是适用于 ASP.NET Core 的默认跨平台 HTTP 服务器。
- [HTTP.sys](#) 是仅适用于 Windows 的 HTTP 服务器，它基于[核心驱动程序和 HTTP 服务器 API](#)。(HTTP.sys 在 ASP.NET 1.x 中被命名为 [WebListener](#)。)

Kestrel

Kestrel 是 ASP.NET Core 项目模板中包括的默认 Web 服务器。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Kestrel 可以单独使用，也可以与反向代理服务器(如 IIS、Nginx 或 Apache)一起使用。反向代理服务器接收到来自 Internet 的 HTTP 请求，并在进行一些初步处理后将这些请求转发到 Kestrel。



如果仅向内部网络公开 Kestrel，可以使用任一配置(无需考虑是否使用反向代理服务器)。

有关详细信息，请参阅[何时结合使用 Kestrel 和反向代理](#)。

在没有 Kestrel 或[自定义服务器](#)实现的情况下，不能使用 IIS、Nginx 和 Apache。ASP.NET Core 设计为在其自己的进程中运行，以实现跨平台统一操作。IIS、Nginx 和 Apache 规定自己的启动过程和环境。若要直接使用这些服务器技术，ASP.NET Core 必须满足每个服务器的需求。使用 Kestrel 等 Web 服务器实现时，ASP.NET Core 可以控制托管在不同服务器技术上的启动过程和环境。

IIS 与 Kestrel

将 [IIS](#) 或 [IIS Express](#) 用作 ASP.NET Core 的反向代理时，ASP.NET Core 应用在独立于 IIS 工作进程的某个进程中运行。在 IIS 进程中，[ASP.NET Core 模块](#)协调反向代理关系。ASP.NET Core 模块的主要功能是启动 ASP.NET Core 应用，在其出现故障时重启应用，并向应用转发 HTTP 流量。有关详细信息，请参阅[ASP.NET Core 模块](#)。

Nginx 与 Kestrel

若要了解如何在 Linux 上使用 Nginx 作为 Kestrel 的反向代理服务器，请参阅[在 Linux 上使用 Nginx 进行托管](#)。

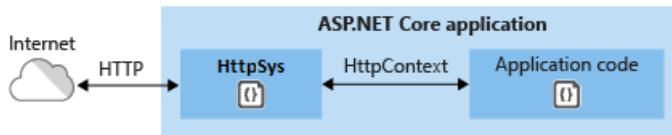
Apache 与 Kestrel

若要了解如何在 Linux 上使用 Apache 作为 Kestrel 的反向代理服务器，请参阅[在 Linux 上使用 Apache 进行托管](#)。

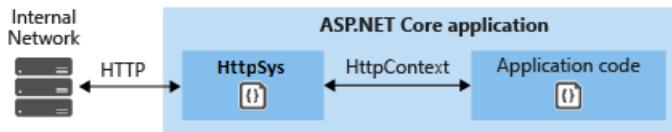
HTTP.sys

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果 ASP.NET Core 应用在 Windows 上运行，则 HTTP.sys 是 Kestrel 的替代选项。为了获得最佳性能，通常建议使用 Kestrel。在向 Internet 公开应用且所需功能受 HTTP.sys 支持（而不是 Kestrel）的方案中，可以使用 HTTP.sys。有关 HTTP.sys 功能的信息，请参阅 [HTTP.sys](#)。



对于仅向内部网络公开的应用，HTTP.sys 同样适用。



ASP.NET Core 服务器基础结构

`Startup.Configure` 方法中提供的 [IApplicationBuilder](#) 公开了类型 [IFeatureCollection](#) 的 `ServerFeatures` 属性。Kestrel 和 HTTP.sys（在 ASP.NET Core 1.x 中为 `WebListener`）各自仅公开单个功能，即 [IServerAddressesFeature](#)，但是不同的服务器实现可能公开其他功能。

[IServerAddressesFeature](#) 可用于查找服务器实现在运行时绑定的端口。

自定义服务器

如果内置服务器无法满足应用需求，可以创建一个自定义服务器实现。[.NET 的开放 Web 接口 \(OWIN\) 指南](#) 演示了如何编写基于 `Nowin` 的 [IServer](#) 实现。只有应用使用的功能接口需要实现，但至少必须支持 [IHttpRequestFeature](#) 和 [IHttpResponseFeature](#)。

服务器启动

如果使用 [Visual Studio](#)、[Visual Studio for Mac](#) 或 [Visual Studio Code](#)，则服务器将在集成开发环境 (IDE) 启动应用时启动。在 Windows 上的 Visual Studio 中，可使用启动配置文件通过 [IIS Express/ASP.NET Core 模块](#) 或控制台来启动应用和服务器。在 Visual Studio Code 中，应用和服务器由 [Omnisharp](#) 启动，这会激活 CoreCLR 调试程序。使用 Visual Studio for Mac 时，应用和服务器由 [Mono Soft-Mode 调试程序](#) 启动。

从项目文件夹中的命令提示符启动应用时，`dotnet run` 会启动该应用和服务器（仅 Kestrel 和 HTTP.sys）。可通过 `-c|--configuration` 选项指定此配置，该选项设置为 `Debug`（默认值）或 `Release`。如果启动配置文件位于 `launchSettings.json` 文件中，请使用 `--launch-profile <NAME>` 选项设置启动配置文件（例如 `Development` 或 `Production`）。有关详细信息，请参阅 [dotnet run](#) 和 [.NET Core 分发打包](#) 主题。

其他资源

- [Kestrel](#)
- [Kestrel 与 IIS](#)
- [在 Linux 上使用 Nginx 进行托管](#)
- [在 Linux 上使用 Apache 进行托管](#)
- [HTTP.sys](#)（对于 ASP.NET Core 1.x，请参阅 [WebListener](#)）

ASP.NET Core 中的 Kestrel Web 服务器实现

2018/5/14 • 25 min to read • [Edit Online](#)

作者: [Tom Dykstra](#)、[Chris Ross](#) 和 [Stephen Halter](#)

Kestrel 是跨平台 [ASP.NET Core Web 服务器](#), 它基于 [libuv](#)(一个跨平台异步 I/O 库)。Kestrel 是 Web 服务器, 默认包括在 ASP.NET Core 项目模板中。

Kestrel 支持以下功能:

- HTTPS
- 用于启用 [WebSocket](#) 的不透明升级
- 用于获得 Nginx 高性能的 Unix 套接字

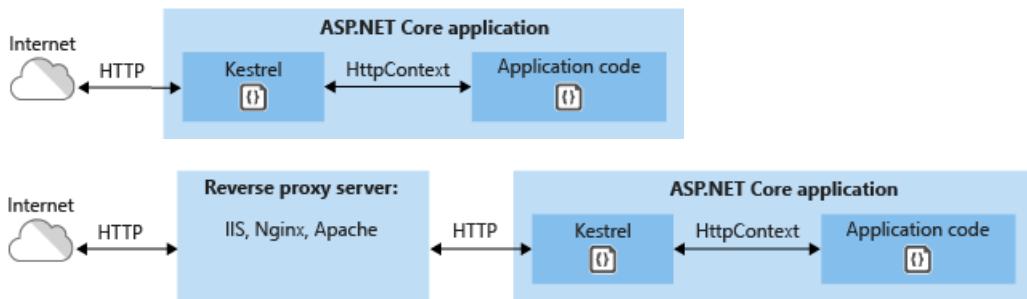
.NET Core 支持的所有平台和版本均支持 Kestrel。

[查看或下载示例代码\(如何下载\)](#)

何时结合使用 Kestrel 和反向代理

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

可以单独使用 Kestrel, 也可以将其与反向代理服务器(如 IIS、Nginx 或 Apache)结合使用。反向代理服务器接收到来自 Internet 的 HTTP 请求, 并在进行一些初步处理后将这些请求转发到 Kestrel。



建议通过反向代理服务器使用 Kestrel, 除非 Kestrel 只对内部网络公开。

具有共享在单个服务器上运行的相同 IP 和端口的多个应用时, 需要一个反向代理方案。Kestrel 不支持此方案, 因为 Kestrel 不支持在多个进程之间共享相同的 IP 和端口。如果将 Kestrel 配置为侦听某个端口, Kestrel 会处理该端口的所有流量(无视请求的主机标头)。可以共享端口的反向代理能在唯一的 IP 和端口上将请求转发至 Kestrel。

即使不需要反向代理服务器, 使用反向代理服务器可能也是个不错的选择:

- 它可以限制所承载的应用中的公开的公共外围应用。
- 它可以提供额外的配置和防护层。
- 它可以更好地与现有基础结构集成。
- 它可以简化负载均衡和 SSL 配置。仅反向代理服务器需要 SSL 证书, 并且该服务器可使用普通 HTTP 在内部网络上与应用服务器通信。

警告

如果在启用了主机筛选的情况下不使用反向代理，则必须启用“主机筛选”。

如何在 ASP.NET Core 应用中使用 Kestrel

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

`Microsoft.AspNetCore.All` 元包中包括 `Microsoft.AspNetCore.Server.Kestrel` 包。

默认情况下，ASP.NET Core 项目模板使用 Kestrel。在 `Program.cs` 中，模板代码调用 `CreateDefaultBuilder`，后者在后台调用 `UseKestrel`。

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

Kestrel 选项

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

Kestrel Web 服务器具有约束配置选项，这些选项在面向 Internet 的部署中尤其有用。可以自定义的一些重要限制：

- 客户端最大连接数
- 请求正文最大大小
- 请求正文最小数据速率

可在 `KestrelServerOptions` 类的 `Limits` 属性上设置这些约束和其他约束。`Limits` 属性包含 `KestrelServerLimits` 类的实例。

客户端最大连接数

`MaxConcurrentConnections`

`MaxConcurrentUpgradedConnections`

可使用以下代码为整个应用设置并发打开的最大 TCP 连接数：

```
.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
})
```

对于已从 HTTP 或 HTTPS 升级到另一个协议(例如, Websocket 请求)的连接, 有一个单独的限制。连接升级后, 不会计入 `MaxConcurrentConnections` 限制。

默认情况下, 最大连接数不受限制 (NULL)。

请求正文最大大小

MaxRequestBodySize

默认的请求正文最大大小为 30,000,000 字节, 大约 28.6 MB。

在 ASP.NET Core MVC 应用中替代限制的推荐方法是在操作方法上使用 `RequestSizeLimit` 属性:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

以下示例演示如何为每个请求上的应用配置约束:

```
.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
})
```

可在中间件中替代特定请求的设置:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;
    context.Features.Get<IHttpMinRequestBodyDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    context.Features.Get<IHttpMinResponseDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
});
```

如果在应用开始读取请求后尝试配置请求限制，则会引发异常。`IsReadOnly` 属性指示 `MaxRequestBodySize` 属性处于只读状态，意味已经无法再配置限制。

请求正文最小数据速率

[MinRequestBodyDataRate](#)

[MinResponseDataRate](#)

Kestrel 每秒检查一次数据是否以指定的速率(字节/秒)传入。如果速率低于最小值，则连接超时。宽限期是 Kestrel 提供给客户端用于将其发送速率提升到最小值的时间量；在此期间不会检查速率。宽限期有助于避免最初由于 TCP 慢启动而以较慢速率发送数据的连接中断。

默认的最小速率为 240 字节/秒，包含 5 秒的宽限期。

最小速率也适用于响应。除了属性和接口名称中具有 `RequestBody` 或 `Response` 以外，用于设置请求限制和响应限制的代码相同。

以下示例演示如何在 `Program.cs` 中配置最小数据速率：

```
.UseKestrel(options =>
{
    options.Limits.MaxConcurrentConnections = 100;
    options.Limits.MaxConcurrentUpgradedConnections = 100;
    options.Limits.MaxRequestBodySize = 10 * 1024;
    options.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    options.Listen(IPAddress.Loopback, 5000);
    options.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
})
```

可在中间件中配置每个请求的速率：

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;
    context.Features.Get<IHttpMinRequestBodyDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    context.Features.Get<IHttpMinResponseDataRateFeature>()
        .MinDataRate = new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
});
```

有关其他 Kestrel 选项和限制的信息，请参阅：

- [KestrelServerOptions](#)
- [KestrelServerLimits](#)
- [ListenOptions](#)

终结点配置

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

默认情况下，ASP.NET Core 绑定到 `http://localhost:5000`。在 [KestrelServerOptions](#) 上调用 [Listen](#) 或 [ListenUnixSocket](#) 方法，从而配置 Kestrel 的 URL 前缀和端口。`UseUrls`、`--urls` 命令行参数、`urls` 主机配置键以及 `ASPNETCORE_URLS` 环境变量也有用，但具有本节后面注明的限制。

`urls` 主机配置键必须来自主机配置，而不是应用配置。将 `urls` 键和值添加到 `appsettings.json` 不影响主机配置，因为是在从配置文件读取配置时对主机进行完全初始化的。但是，在主机生成器上可以使用 `appsettings.json` 中的 `urls` 键以及 [UseConfiguration](#) 来配置主机：

```
var config = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appSettings.json", optional: true, reloadOnChange: true)
    .Build();

var host = new WebHostBuilder()
    .UseKestrel()
    .UseConfiguration(config)
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseStartup<Startup>()
    .Build();
```

注意

ASP.NET Core 2.1 is in preview and not recommended for production use.

默认情况下，ASP.NET Core 绑定到：

- `http://localhost:5000`
- `https://localhost:5001` (存在本地开发证书时)

开发证书会创建于以下情况：

- 安装了 [.NET Core SDK](#) 时。
- [dev-certs tool](#) 用于创建证书。

部分浏览器需要获取信任本地开发证书的显示权限。

ASP.NET Core 2.1 及更高版本的项目模板将应用配置为默认情况下在 HTTPS 上运行，并包括 [HTTPS 重定向](#) 和 [HSTS 支持](#)。

在 [KestrelServerOptions](#) 上调用 [Listen](#) 或 [ListenUnixSocket](#) 方法，从而配置 Kestrel 的 URL 前缀和端口。

`UseUrls`、`--urls` 命令行参数、`urls` 主机配置键以及 `ASPNETCORE_URLS` 环境变量也有用，但具有本节后面注明的限制(必须要有可用于 HTTPS 终结点配置的默认证书)。

ASP.NET Core 2.1 `KestrelServerOptions` 配置：

ConfigureEndpointDefaults(Action)

指定一个为每个指定的终结点运行的配置 `Action`。多次调用 `ConfigureEndpointDefaults`，用最新指定的 `Action` 替换之前的 `Action`。

ConfigureHttpsDefaults(Action)

指定一个为每个 HTTPS 终结点运行的配置 `Action`。多次调用 `ConfigureHttpsDefaults`，用最新指定的 `Action` 替换之前的 `Action`。

Configure(IConfiguration)

创建配置加载程序，用于设置将 `IConfiguration` 作为输入的 Kestrel。配置必须针对 Kestrel 的配置节。

ListenOptions.UseHttps

将 Kestrel 配置为使用 HTTPS。

`ListenOptions.UseHttps` 扩展：

- `UseHttps` – 将 Kestrel 配置为使用 HTTPS，采用默认证书。如果没有配置默认证书，则会引发异常。
- `UseHttps(string fileName)`
- `UseHttps(string fileName, string password)`
- `UseHttps(string fileName, string password, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(StoreName storeName, string subject)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location)`
- `Action<HttpsConnectionAdapterOptions> configureOptions`
- `UseHttps(X509Certificate2 serverCertificate)`
- `UseHttps(X509Certificate2 serverCertificate, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(Action<HttpsConnectionAdapterOptions> configureOptions)`

`ListenOptions.UseHttps` 参数：

- `filename` 是证书文件的路径和文件名，关联包含应用内容文件的目录。
- `password` 是访问 X.509 证书数据所需的密码。
- `configureOptions` 是配置 `HttpsConnectionAdapterOptions` 的 `Action`。返回 `ListenOptions`。
- `storeName` 是从中加载证书的证书存储。
- `subject` 是证书的主题名称。
- `allowInvalid` 指示是否存在需要留意的无效证书，例如自签名证书。
- `location` 是从中加载证书的存储位置。
- `serverCertificate` 是 X.509 证书。

在生产中，必须显式配置 HTTPS。至少必须提供默认证书。

下面要描述的支持的配置：

- 无配置
- 从配置中替换默认证书
- 更改代码中的默认值

无配置

Kestrel 在 `http://localhost:5000` 和 `https://localhost:5001` 上进行侦听（如果默认证书可用）。

使用以下内容指定 URL：

- `ASPNETCORE_URLS` 环境变量。
- `--urls` 命令行参数。
- `urls` 主机配置键。
- `UseUrls` 扩展方法。

有关详细信息, 请参阅[服务器 URL 和重写配置](#)。

采用这些方法提供的值可以是一个或多个 HTTP 和 HTTPS 终结点(如果默认证书可用, 则为 HTTPS)。将值配置为以分号分隔的列表(例如 `"Urls": "http://localhost:8000;http://localhost:8001"`)。

[从配置中替换默认证书](#)

[WebHost.CreateDefaultBuilder](#) 在默认情况下调用

`serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` 来加载 Kestrel 配置。Kestrel 可以使用默认 HTTPS 应用设置配置架构。从磁盘上的文件或从证书存储中配置多个终结点, 包括要使用的 URL 和证书。

在以下 appsettings.json 示例中:

- 将 `AllowInvalid` 设置为 `true`, 从而允许使用无效证书(例如自签名证书)。
- 任何未指定证书的 HTTPS 终结点(下例中的 `HttpsDefaultCert`)会回退至在 Certificates > Default 下定义的证书或开发证书。

```
{  
  "Kestrel": {  
    "EndPoints": {  
      "Http": {  
        "Url": "http://localhost:5000"  
      },  
  
      "HttpsInlineCertFile": {  
        "Url": "https://localhost:5001",  
        "Certificate": {  
          "Path": "<path to .pfx file>",  
          "Password": "<certificate password>"  
        }  
      },  
  
      "HttpsInlineCertStore": {  
        "Url": "https://localhost:5002",  
        "Certificate": {  
          "Subject": "<subject; required>",  
          "Store": "<certificate store; defaults to My>",  
          "Location": "<location; defaults to CurrentUser>",  
          "AllowInvalid": "<true or false; defaults to false>"  
        }  
      },  
  
      "HttpsDefaultCert": {  
        "Url": "https://localhost:5003"  
      },  
  
      "Https": {  
        "Url": "https://*:5004",  
        "Certificate": {  
          "Path": "<path to .pfx file>",  
          "Password": "<certificate password>"  
        }  
      },  
    },  
    "Certificates": {  
      "Default": {  
        "Path": "<path to .pfx file>",  
        "Password": "<certificate password>"  
      }  
    }  
  }  
}
```

此外还可以使用任何证书节点的 Path 和 Password，采用证书存储字段指定证书。例如，可将 Certificates > Default 证书指定为：

```
"Default": {  
    "Subject": "<subject; required>",  
    "Store": "<cert store; defaults to My>",  
    "Location": "<location; defaults to CurrentUser>",  
    "AllowInvalid": "<true or false; defaults to false>"  
}
```

架构的注意事项：

- 终结点的名称不区分大小写。例如，`HTTPS` 和 `Https` 都是有效的。
- 每个终结点都要具备 `Url` 参数。此参数的格式和顶层 `Urls` 配置参数一样，只不过它只能有单个值。
- 这些终结点不会添加进顶层 `Urls` 配置中定义的终结点，而是替换它们。通过 `Listen` 在代码中定义的终结点与在配置节中定义的终结点相累积。
- `Certificate` 部分是可选的。如果未指定 `Certificate` 部分，则使用在之前的方案中定义的默认值。如果没有可用的默认值，服务器会引发异常且无法启动。
- `Certificate` 支持 Path-Password 和 Subject-Store 证书。
- 只要不会导致端口冲突，就能以这种方式定义任何数量的终结点。
- `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` 通过 `.Endpoint(string name, options => {})` 方法返回 `KestrelConfigurationLoader`，可用于补充已配置的终结点设置：

```
serverOptions.Configure(context.Configuration.GetSection("Kestrel"))  
.Endpoint("HTTPS", opt =>  
{  
    opt.HttpsOptions.SslProtocols = SslProtocols.Tls12;  
});
```

也可以直接访问 `KestrelServerOptions.ConfigurationLoader` 来保持现有加载程序上的迭代，例如由 `WebHost.CreateDefaultBuilder` 提供的加载程序。

- 每个终结点的配置节都可用于 `Endpoint` 方法中的选项，以便读取自定义设置。
- 通过另一节再次调用 `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` 可能加载多个配置。只使用最新配置，除非之前的实例上显式调用了 `Load`。元包不会调用 `Load`，所以可能会替换它的默认配置节。
- `KestrelConfigurationLoader` 从 `KestrelServerOptions` 将 API 的 `Listen` 簇反射为 `Endpoint` 重载，因此可在同样的位置配置代码和配置终结点。这些重载不使用名称，且只使用配置中的默认设置。

[更改代码中的默认值](#)

可以使用 `ConfigureEndpointDefaults` 和 `ConfigureHttpsDefaults` 更改 `ListenOptions` 和 `HttpsConnectionAdapterOptions` 的默认设置，包括重写之前的方案指定的默认证书。需要在配置任何终结点之前调用 `ConfigureEndpointDefaults` 和 `ConfigureHttpsDefaults`。

```
options.ConfigureEndpointDefaults(opt =>
{
    opt.NoDelay = true;
});

options.ConfigureHttpsDefaults(httpsOptions =>
{
    httpsOptions.SslProtocols = SslProtocols.Tls12;
});
```

SNI 的 Kestrel 支持

服务器名称指示 (SNI) 可用于承载相同 IP 地址和端口上的多个域。为了运行 SNI，客户端在 TLS 握手过程中将进行安全会话的主机名发送至服务器，从而让服务器可以提供正确的证书。在 TLS 握手后的安全会话期间，客户端将服务器提供的证书用于与服务器进行加密通信。

Kestrel 通过 `ServerCertificateSelector` 回调支持 SNI。每次连接调用一次回调，从而允许应用检查主机名并选择合适的证书。

SNI 支持需要在目标框架 `netcoreapp2.1` 上允许。在 `netcoreapp2.0` 和 `net461` 上，回调也会调用，但是 `name` 始终为 `null`。如果客户端未在 TLS 握手过程中提供主机名参数，则 `name` 也为 `null`。

```
WebHost.CreateDefaultBuilder()
    .UseKestrel((context, options) =>
{
    options.ListenAnyIP(5005, listenOptions =>
{
    listenOptions.UseHttps(httpsOptions =>
{
    var localhostCert = CertificateLoader.LoadFromStoreCert(
        "localhost", "My", StoreLocation.CurrentUser,
        allowInvalid: true);
    var exampleCert = CertificateLoader.LoadFromStoreCert(
        "example.com", "My", StoreLocation.CurrentUser,
        allowInvalid: true);
    var subExampleCert = CertificateLoader.LoadFromStoreCert(
        "sub.example.com", "My", StoreLocation.CurrentUser,
        allowInvalid: true);
    var certs = new Dictionary(StringComparer.OrdinalIgnoreCase);
    certs["localhost"] = localhostCert;
    certs["example.com"] = exampleCert;
    certs["sub.example.com"] = subExampleCert;

    httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
{
    if (name != null && certs.TryGetValue(name, out var cert))
    {
        return cert;
    }

    return exampleCert;
};
});
});
});
});
```

绑定到 TCP 套接字

`Listen` 方法绑定至 TCP 套接字，并可通过选项 `lambda` 配置 SSL 证书：

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

请注意此示例如何使用 [ListenOptions](#) 为特定终结点配置 SSL。可使用相同 API 为特定终结点配置其他 Kestrel 设置。

在 Windows 上，可以使用 [New-SelfSignedCertificate PowerShell cmdlet](#) 创建自签名证书。有关不支持的示例，请参阅 [UpdateIISExpressSSLForChrome.ps1](#)。

在 macOS、Linux 和 Windows 上，可以使用 [OpenSSL](#) 创建证书。

绑定到 Unix 套接字

可通过 [ListenUnixSocket](#) 倾听 Unix 套接字以提高 Nginx 的性能，如以下示例所示：

```
.UseKestrel(options =>
{
    options.ListenUnixSocket("/tmp/kestrel-test.sock");
    options.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testpassword");
    });
})
```

端口 0

如果指定端口号 `0`，Kestrel 将动态绑定到可用端口。以下示例演示如何确定 Kestrel 在运行时实际绑定到的端口：

在应用运行时，控制台窗口输出指示可用于访问应用的动态端口：

```
Now listening on: http://127.0.0.1:48508
```

`UseUrls`、`--urls` 命令行参数、`urls` 主机配置键以及 `ASPNETCORE_URLS` 环境变量的限制

使用以下方法配置终结点：

- [UseUrls](#)
- `--urls` 命令行参数
- `urls` 主机配置键
- `ASPNETCORE_URLS` 环境变量

若要将代码用于 Kestrel 以外的服务器，这些方法非常有用。但是，请注意以下限制：

- SSL 不能使用这些方法，除非 HTTPS 终结点配置中提供了默认证书（如本主题前面的部分所示，使用 `KestrelServerOptions` 配置或配置文件）。
- 如果同时使用 `Listen` 和 `UseUrls` 方法，`Listen` 终结点将覆盖 `UseUrls` 终结点。

IIS 终结点配置

使用 IIS 时，由 `Listen` 或 `UseUrls` 设置用于 IIS 覆盖绑定的 URL 绑定。有关详细信息，请参阅 [ASP.NET Core 模块主题](#)。

URL 前缀

如果使用 `UseUrls`、`--urls` 命令行参数、`urls` 主机配置键或 `ASPNETCORE_URLS` 环境变量，URL 前缀可采用以下任意格式。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

仅 HTTP URL 前缀是有效的。使用 `UseUrls` 配置 URL 绑定时，Kestrel 不支持 SSL。

- 包含端口号的 IPv4 地址

```
http://65.55.39.10:80/
```

`0.0.0.0` 是一种绑定到所有 IPv4 地址的特殊情况。

- 包含端口号的 IPv6 地址

```
http://[0:0:0:0:0:ffff:4137:270a]:80/
```

`[::]` 是 IPv4 `0.0.0.0` 的 IPv6 等效项。

- 包含端口号的主机名

```
http://contoso.com:80/  
http://*:80/
```

主机名、`*` 和 `+` 并不特殊。没有识别为有效 IP 地址或 `localhost` 的任何内容都将绑定到所有 IPv4 和 IPv6 IP。若要将不同主机名绑定到相同端口上的不同 ASP.NET Core 应用，请使用 [HTTP.sys](#) 或 IIS、Nginx 或 Apache 等反向代理服务器。

警告

如果在启用了主机筛选的情况下不使用反向代理，请启用[主机筛选](#)。

- 包含端口号的主机 `localhost` 名称或包含端口号的环回 IP

```
http://localhost:5000/  
http://127.0.0.1:5000/  
http://[::1]:5000/
```

指定 `localhost` 后，Kestrel 将尝试绑定到 IPv4 和 IPv6 环回接口。如果其他服务正在任一环回接口上使用请求的端口，则 Kestrel 将无法启动。如果任一环回接口出于任何其他原因（通常是因为 IPv6 不受支持）而不可用，则 Kestrel 将记录一个警告。

主机筛选

尽管 Kestrel 支持基于前缀的配置（例如 `http://example.com:5000`），但 Kestrel 在很大程度上会忽略主机名。主机 `localhost` 是一个特殊情况，用于绑定至环回地址。除了显式 IP 地址以外的所有主机都绑定至所有公共 IP 地址。这些信息都不用于验证请求 `Host` 标头。

有两种解决方法：

- 使用主机标头筛选的反向代理的主机。这是 ASP.NET Core 1.x 中唯一的 Kestrel 支持方案。
- 使用中间件来根据 `Host` 标头筛选请求。下面是中间件示例：

```
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Primitives;
using Microsoft.Net.Http.Headers;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

// A normal middleware would provide an options type, config binding, extension methods, etc..
// This intentionally does all of the work inside of the middleware so it can be
// easily copy-pasted into docs and other projects.
public class HostFilteringMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IList<string> _hosts;
    private readonly ILogger<HostFilteringMiddleware> _logger;

    public HostFilteringMiddleware(RequestDelegate next, IConfiguration config,
ILogger<HostFilteringMiddleware> logger)
    {
        if (config == null)
        {
            throw new ArgumentNullException(nameof(config));
        }

        _next = next ?? throw new ArgumentNullException(nameof(next));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));

        // A semicolon separated list of host names without the port numbers.
        // IPv6 addresses must use the bounding brackets and be in their normalized form.
        _hosts = config["AllowedHosts"]?.Split(new[] { ';' }, StringSplitOptions.RemoveEmptyEntries);
        if (_hosts == null || _hosts.Count == 0)
        {
            throw new InvalidOperationException("No configuration entry found for AllowedHosts.");
        }
    }

    public Task Invoke(HttpContext context)
    {
        if (!ValidateHost(context))
        {
            context.Response.StatusCode = 400;
            _logger.LogDebug("Request rejected due to incorrect Host header.");
            return Task.CompletedTask;
        }

        return _next(context);
    }

    // This does not duplicate format validations that are expected to be performed by the host.
    private bool ValidateHost(HttpContext context)
    {
        StringSegment host = context.Request.Headers[HeaderNames.Host].ToString().Trim();

        if (StringSegment.IsNullOrEmpty(host))
        {
            // Http/1.0 does not require the Host header.
            // Http/1.1 requires the header but the value may be empty.
            return true;
        }
    }
}
```

```

// Drop the port

var colonIndex = host.LastIndexOf(':');

// IPv6 special case
if (host.StartsWith("[", StringComparison.OrdinalIgnoreCase))
{
    var endBracketIndex = host.IndexOf(']');
    if (endBracketIndex < 0)
    {
        // Invalid format
        return false;
    }
    if (colonIndex < endBracketIndex)
    {
        // No port, just the IPv6 Host
        colonIndex = -1;
    }
}

if (colonIndex > 0)
{
    host = host.Subsegment(0, colonIndex);
}

foreach (var allowedHost in _hosts)
{
    if (StringSegment.Equals(allowedHost, host, StringComparison.OrdinalIgnoreCase))
    {
        return true;
    }

    // Sub-domain wildcards: *.example.com
    if (allowedHost.StartsWith(".", StringComparison.OrdinalIgnoreCase) && host.Length >=
allowedHost.Length)
    {
        // .example.com
        var allowedRoot = new StringSegment(allowedHost, 1, allowedHost.Length - 1);

        var hostRoot = host.Subsegment(host.Length - allowedRoot.Length, allowedRoot.Length);
        if (hostRoot.Equals(allowedRoot, StringComparison.OrdinalIgnoreCase))
        {
            return true;
        }
    }
}

return false;
}
}

```

在 `Startup.Configure` 中注册上述 `HostFilteringMiddleware`。请注意，[中间件注册的排序](#)非常重要。应在诊断中间件（例如 `app.UseExceptionHandler`）注册完成后立即进行注册。

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

    app.UseMiddleware<HostFilteringMiddleware>();

    app.UseMvcWithDefaultRoute();
}
```

上述中间件需要 appsettings.<EnvironmentName>.json 中的 `AllowedHosts` 键。此键的值是以分号分隔的不带端口号的主机名列表。包括 appsettings.Production.json 中的 `AllowedHosts` 键值对：

```
{
    "AllowedHosts": "example.com"
}
```

appsettings.Development.json (localhost 配置文件)：

```
{
    "AllowedHosts": "localhost"
}
```

其他资源

- [Enforce HTTPS](#)
- [Kestrel 源代码](#)

ASP.NET Core 模块

2018/5/17 • 3 min to read • [Edit Online](#)

作者:Tom Dykstra、Rick Strahl 和 Chris Ross

ASP.NET Core 模块允许 ASP.NET Core 应用在采用反向代理配置的 IIS 后运行。IIS 具有高级 Web 应用安全性和易管理性。

受支持的 Windows 版本:

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

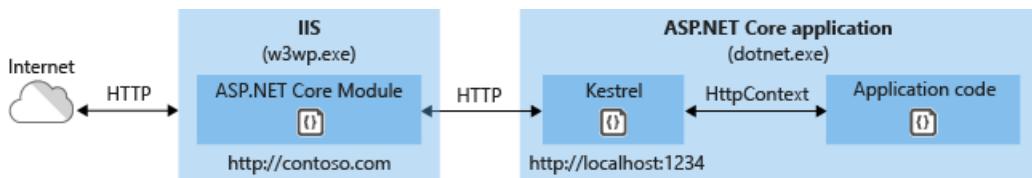
ASP.NET Core 模块仅适用于 Kestrel。该模块与 [HTTP.sys](#)(以前称为 [WebListener](#))不兼容。

ASP.NET Core 模块说明

ASP.NET Core 模块是插入 IIS 管道的本机 IIS 模块, 用于将 Web 请求重定向到后端 ASP.NET Core 应用。许多本机模块(如 Windows 身份验证)仍处于活动状态。要详细了解随该模块活动的 IIS 模块, 请参阅 [IIS 模块](#)。

由于 ASP.NET Core 应用在独立于 IIS 辅助进程的进程中运行, 因此该模块还处理进程管理。该模块在第一个请求到达时启动 ASP.NET Core 应用的进程, 并在应用崩溃时重新启动该应用。这基本上与在 [Windows 进程激活服务 \(WAS\)](#) 托管的 IIS 中在进程内运行的 ASP.NET 4.x 应用中出现的行为相同。

下图说明了 IIS、ASP.NET Core 模块和 ASP.NET Core 应用之间的关系:



请求从 Web 到达内核模式 HTTP.sys 驱动程序。驱动程序将请求路由到网站的配置端口上的 IIS, 通常为 80 (HTTP) 或 443 (HTTPS)。该模块将该请求转发到应用的随机端口(非端口 80/443)上的 Kestrel。

该模块在启动时通过环境变量指定端口, IIS 集成中间件将服务器配置为侦听 `http://localhost:{port}`。执行其他检查, 拒绝不是来自该模块的请求。该模块不支持 HTTPS 转发, 因此即使请求由 IIS 通过 HTTPS 接收, 它们还是通过 HTTP 转发。

Kestrel 从模块获取请求后, 请求会被推送到 ASP.NET Core 中间件管道中。中间件管道处理该请求并将其作为 `HttpContext` 实例传递给应用的逻辑。应用的响应传递回 IIS, IIS 将响应推送给发起请求的 HTTP 客户端。

ASP.NET Core 模块具有一些其他功能。该模块可以:

- 为工作进程设置环境变量。
- 将 stdout 输出记录到文件存储器, 以解决启动问题。
- 转发 Windows 身份验证令牌。

如何安装和使用 ASP.NET Core 模块

有关如何安装和使用 ASP.NET Core 模块的详细说明, 请参阅[使用 IIS 在 Windows 上进行托管](#)。有关配置模块的信息, 请参阅 [ASP.NET Core 模块配置参考](#)。

其他资源

- [使用 IIS 在 Windows 上进行托管](#)
- [ASP.NET Core 模块配置参考](#)
- [ASP.NET Core 模块 GitHub 存储库\(源代码\)](#)

ASP.NET Core 中的 HTTP.sys Web 服务器实现

2018/5/17 • 9 min to read • [Edit Online](#)

作者: [Tom Dykstra](#)、[Chris Ross](#) 和 [Luke Latham](#)

注意

本主题仅适用于 ASP.NET Core 2.0 或更高版本。在早期版本的 ASP.NET Core 的中, HTTP.sys 被命名为 [WebListener](#)。

HTTP.sys 是仅在 Windows 上运行的适用于 ASP.NET Core 的 Web 服务器。HTTP.sys 是 [Kestrel](#) 的替代选择, 提供了一些 Kestrel 不提供的功能。

重要事项

HTTP.sys 与 [ASP.NET Core 模块](#)不兼容, 不能与 IIS 或 IIS Express 结合使用。

HTTP.sys 支持以下功能:

- [Windows 身份验证](#)
- 端口共享
- 具有 SNI 的 HTTPS
- 基于 TLS 的 HTTP/2(Windows 10 或更高版本)
- 直接文件传输
- 响应缓存
- WebSocket(Windows 8 或更高版本)

受支持的 Windows 版本:

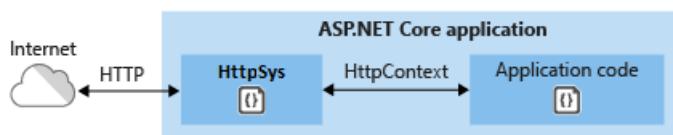
- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

[查看或下载示例代码\(如何下载\)](#)

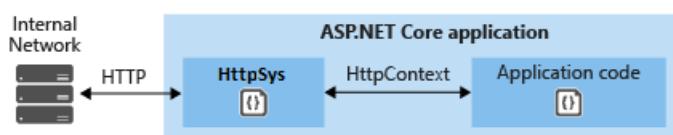
何时使用 HTTP.sys

HTTP.sys 对于以下情形的部署来说很有用:

- 需要将服务器直接公开到 Internet 而不使用 IIS 的部署。



- 内部部署需要 Kestrel 中没有的功能, 如 [Windows 身份验证](#)。



HTTP.sys 是一项成熟的技术, 可以抵御多种攻击, 并提供可靠、安全、可伸缩的全功能 Web 服务器。IIS 本身

作为 HTTP.sys 之上的 HTTP 倾听器运行。

如何使用 HTTP.sys

配置 ASP.NET Core 应用以使用 HTTP.sys

1. 使用 [Microsoft.AspNetCore.All metapackage \(nuget.org\)](#)(ASP.NET Core 2.0 或更高版本)时, 不需要项目文件中的包引用。未使用 `Microsoft.AspNetCore.All` 元包时, 向 `Microsoft.AspNetCore.Server.HttpSys` 添加包引用。
2. 构建 Web 主机时调用 `UseHttpSys` 扩展方法, 同时指定所需的 HTTP.sys 选项:

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
    {
        // The following options are set to default values.
        options.Authentication.Schemes = AuthenticationSchemes.None;
        options.Authentication.AllowAnonymous = true;
        options.MaxConnections = null;
        options.MaxRequestBodySize = 30000000;
        options.UrlPrefixes.Add("http://localhost:5000");
    })
    .Build();
```

通过[注册表设置](#)处理其他 HTTP.sys 配置。

HTTP.sys 选项

属性	描述	默认
<code>AllowSynchronousIO</code>	控制是否允许 <code>HttpContext.Request.Body</code> 和 <code>HttpContext.Response.Body</code> 的同步输入/输出。	<code>true</code>
<code>Authentication.AllowAnonymous</code>	允许匿名请求。	<code>true</code>
<code>Authentication.Schemes</code>	指定允许的身份验证方案。可能在处理倾听器之前随时修改。通过 AuthenticationSchemes 枚举 <code>Basic</code> 、 <code>Kerberos</code> 、 <code>Negotiate</code> 、 <code>None</code> 和 <code>NTLM</code> 提供值。	<code>None</code>
<code>EnableResponseCaching</code>	尝试 内核模式 缓存, 响应合格的标头。该响应可能不包括 <code>Set-Cookie</code> 、 <code>Vary</code> 或 <code>Pragma</code> 标头。它必须包括属性为 <code>public</code> 的 <code>Cache-Control</code> 标头和 <code>shared-max-age</code> 或 <code>max-age</code> 值, 或 <code>Expires</code> 标头。	<code>true</code>
<code>MaxAccepts</code>	最大并发接受数量。	<code>5 × 环境。ProcessorCount</code>

属性	描述	默认
MaxConnections	要接受的最大并发连接数。使用 <code>-1</code> 实现无限。通过 <code>null</code> 使用注册表的计算机范围内的设置。	<code>null</code> (无限制)
MaxRequestBodySize	请参阅 MaxRequestBodySize 部分。	30000000 个字节 (~28.6 MB)
RequestQueueLimit	队列中允许的最大请求数。	1000
ThrowWriteExceptions	指示由于客户端断开连接而失败的响应主体写入应引发异常还是正常完成。	<code>false</code> (正常完成)
超时	<p>公开 <code>HTTP.sys TimeoutManager</code> 配置，也可以在注册表中进行配置。请访问 API 链接详细了解每个设置，包括默认值：</p> <ul style="list-style-type: none"> • <code>Timeouts.DrainEntityBody</code> – 允许 HTTP 服务器 API 在保持活动的连接上排出实体正文的时间。 • <code>Timeouts.EntityBody</code> – 允许请求实体正文到达的时间。 • <code>Timeouts.HeaderWait</code> – 允许 HTTP 服务器 API 分析请求头的时间。 • <code>Timeouts.IdleConnection</code> – 对空闲连接允许的时间。 • <code>Timeouts.MinSendBytesPerSecond</code> – 响应的最小发送速率。 • <code>Timeouts.RequestQueue</code> – 在应用选取请求前，允许请求在请求队列中停留的时间。 	
UrlPrefixes	指定 <code>UrlPrefixCollection</code> 以注册 <code>HTTP.sys</code> 。最有用的是 <code>UrlPrefixCollection.Add</code> ，它用于将前缀添加到集合中。可能在处理侦听器之前随时对这些设置进行修改。	

MaxRequestBodySize

允许的请求正文的最大大小(以字节计)。当设置为 `null` 时，最大请求正文大小不受限制。此限制不会影响升级后的连接，这始终不受限制。

在 ASP.NET Core MVC 应用中为单个 `IActionResult` 替代限制的推荐方法是在操作方法上使用 `RequestSizeLimitAttribute` 属性：

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

如果在应用开始读取请求后尝试配置请求限制，则会引发异常。`IsReadOnly` 属性可用于指示

`MaxRequestBodySize` 属性是否处于只读状态。只读状态意味着已经太迟了，无法配置限制。

如果应用应替代每个请求的 `MaxRequestBodySize`，则使用 [IHttpMaxRequestBodySizeFeature](#)：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILogger<Startup> logger)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get< IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature = app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation($"Addresses: {addresses}");

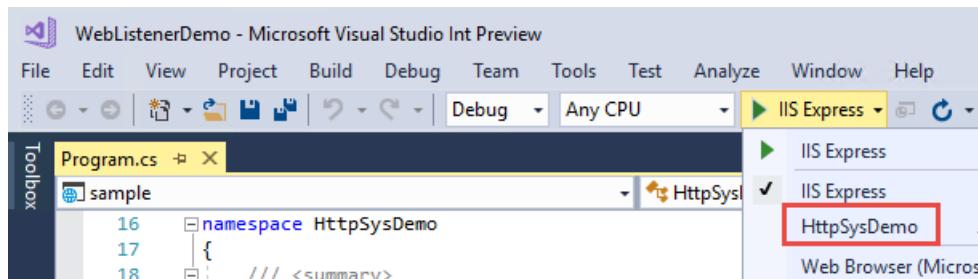
        await next.Invoke();
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorResponse();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseMvc();
}
```

3. 如果使用的是 Visual Studio，请确保应用未经配置以运行 IIS 或 IIS Express。

在 Visual Studio 中，默认启动配置文件是针对 IIS Express 的。若要作为控制台应用运行该项目，请手动更改所选配置文件，如以下屏幕截图中所示：



配置 Windows Server

- 如果应用为[框架相关部署](#)，则安装 .NET Core、.NET Framework 或两者（如果应用是面向 .NET Framework 的 .NET Core 应用）。
 - .NET Core** – 如果应用需要 .NET Core，请从[.NET 所有下载](#)获取并运行 .NET Core 安装程序。
 - .NET Framework** – 如果应用需要 .NET Framework，请参阅[.NET Framework: 安装指南](#)查找安装说明。安装所需的 .NET Framework。最新 .NET Framework 的安装程序可从[.NET 所有下载](#)中找到。
 - 配置应用的 URL 和端口。
- 默认情况下，ASP.NET Core 绑定到 `http://localhost:5000`。若要配置 URL 前缀和端口，选项包括使用：
- [UseUrls](#)

默认情况下，ASP.NET Core 绑定到 `http://localhost:5000`。若要配置 URL 前缀和端口，选项包括使用：

- [UseUrls](#)

- `urls` 命令行参数
- `ASPNETCORE_URLS` 环境变量
- `UrlPrefixes`

下方的代码示例演示了如何使用 `UrlPrefixes`:

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
    {
        // The following options are set to default values.
        options.Authentication.Schemes = AuthenticationSchemes.None;
        options.Authentication.AllowAnonymous = true;
        options.MaxConnections = null;
        options.MaxRequestBodySize = 30000000;
        options.UrlPrefixes.Add("http://localhost:5000");
    })
    .Build();
```

`UrlPrefixes` 的一个优点是会为格式不正确的前缀立即生成一条错误消息。

`UrlPrefixes` 中的设置替代 `UseUrls` / `urls` / `ASPNETCORE_URLS` 设置。因此，`UseUrls`、`urls` 和 `ASPNETCORE_URLS` 环境变量的一个优点是在 Kestrel 和 HTTP.sys 之间切换变得更加容易。有关 `UseUrls`、`urls` 和 `ASPNETCORE_URLS` 的更多信息，请参阅[托管](#)。

HTTP.sys 使用 [HTTP 服务器 API UrlPrefix 字符串格式](#)。

警告

不应使用顶级通配符绑定 (`http://*:80/` 和 `http://+:80`)。顶级通配符绑定可能会为应用带来安全漏洞。此行为同时适用于强通配符和弱通配符。使用显式主机名而不是通配符。如果可控制整个父域(区别于易受攻击的 `*.com`)，则子域通配符绑定(例如，`*.mysub.com`)不具有此安全风险。有关详细信息，请参阅 [rfc7230 第 5.4 条](#)。

3. 预先注册 URL 前缀以绑定到 HTTP.sys，并设置 x.509 证书。

如果未在 Windows 中预先注册 URL 前缀，请使用管理员特权运行应用。唯一的例外是当使用端口号大于 1024 的 HTTP(而非 HTTPS)绑定到 localhost 时。在这种情况下，无需使用管理员特权。

- 用于配置 HTTP.sys 的内置工具为 `netsh.exe`。`netsh.exe` 用于保留 URL 前缀并分配 X.509 证书。此工具需要管理员特权。

以下示例显示了保留端口 80 和 443 的 URL 前缀的命令：

```
netsh http add urlacl url=http://+:80/ user=Users
netsh http add urlacl url=https://+:443/ user=Users
```

以下示例显示了如何分配 X.509 证书：

```
netsh http add sslcert ipport=0.0.0.0:443 certhash=MyCertHash_Here appid="{00000000-0000-0000-0000-000000000000}"
```

`netsh.exe` 的参考文档：

- [Netsh Commands for Hypertext Transfer Protocol \(HTTP\)](#)(超文本传输协议 (HTTP) 的 Netsh 命令)

- [UrlPrefix Strings](#)(UrlPrefix 字符串)

b. 如果需要, 请创建自签名的 X.509 证书。

在 Windows 上, 可以使用 [New-SelfSignedCertificate PowerShell cmdlet](#) 创建自签名证书。有关不支持的示例, 请参阅 [UpdateIISExpressSSLForChrome.ps1](#)。

在 macOS、Linux 和 Windows 上, 可以使用 [OpenSSL](#) 创建证书。

4. 打开防火墙端口以允许流量到达 HTTP.sys。使用 [netsh.exe](#) 或 [PowerShell cmdlet](#)。

代理服务器和负载均衡器方案

如果应用由 HTTP.sys 托管并且与来自 Internet 或公司网络的请求进行交互, 当在代理服务器和负载均衡器后托管时, 可能需要其他配置。有关详细信息, 请参阅[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

其他资源

- [HTTP 服务器 API](#)
- [aspnet/HttpSysServer GitHub 存储库\(源代码\)](#)
- [承载](#)

ASP.NET Core 全球化和本地化

2018/5/17 • 19 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Damien Bowden](#)、[Bart Calixto](#)、[Nadeem Afana](#) 和 [Hisham Bin Ateya](#)

使用 ASP.NET Core 创建多语言网站，可让网站拥有更多受众。ASP.NET Core 提供的服务和中间件可将网站本地化为不同的语言和文化。

国际化涉及[全球化](#)和[本地化](#)。全球化是设计支持不同区域性的应用程序的过程。全球化添加了对一组有关特定地理区域的已定义语言脚本的输入、显示和输出支持。

本地化是将已经针对可本地化性进行处理的全球化应用调整为特定的区域性/区域设置的过程。有关详细信息，请参阅本文档邻近末尾的全球化和本地化术语。

应用本地化涉及以下内容：

1. 使应用内容可本地化
2. 为支持的语言和区域性提供本地化资源
3. 实施策略，为每个请求选择语言/区域性

使应用内容可本地化

ASP.NET Core 中引入并架构了 `IStringLocalizer` 和 `IStringLocalizer<T>`，以提高开发本地化应用的工作效率。`IStringLocalizer` 使用 `ResourceManager` 和 `ResourceReader`，在运行时提供区域性特定资源。简单接口具有一个索引器和一个用于返回本地化字符串的 `IEnumerable`。`IStringLocalizer` 不要求在资源文件中存储默认语言字符串。你可以开发针对本地化的应用，且无需在开发初期创建资源资源文件。下面的代码演示如何针对本地化包装字符串“About Title”。

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.StarterWeb.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

在上面的代码中，`IStringLocalizer<T>` 实现来源于[依赖关系注入](#)。如果找不到“About Title”的本地化值，则返回索引器键，即字符串“About Title”。可将默认语言文本字符串保留在应用中并将它们包装在本地化工具中，以便你可集中精力开发应用。你使用默认语言开发应用，并针对本地化步骤进行准备，而无需首先创建默认资源。

文件。或者，你可以使用传统方法，并提供键以检索默认语言字符串。对于许多开发者而言，不具有默认语言.resx 文件且简单包装字符串文本的新工作流可以减少本地化应用的开销。其他开发者将首选传统工作流，因为它可以更轻松地使用较长字符串文本，更轻松地更新本地化字符串。

对包含 HTML 的资源使用 `IHtmlLocalizer<T>` 实现。`IHtmlLocalizer` 对资源字符串中格式化的参数进行 HTML 编码，但不对资源字符串本身进行 HTML 编码。在下面突出显示的示例中，仅 `name` 参数的值被 HTML 编码。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.StarterWeb.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
    }
}
```

注意：你通常只想要本地化文本，而不是 HTML。

最低程度，你可以从[依赖关系注入](#)获取 `IStringLocalizerFactory`：

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
        }
}
```

上面的代码演示了这两种工厂创建方法。

可以按控制器、区域对本地化字符串分区，或只有一个容器。在示例应用中，名为 `SharedResource` 的虚拟类用于共享资源。

```
// Dummy class to group shared resources

namespace Localization.StarterWeb
{
    public class SharedResource
    {
    }
}
```

某些开发者使用 `Startup` 类，以包含全局或共享字符串。在下面的示例中，使用 `InfoController` 和 `SharedResource` 本地化工具：

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                          IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
}
```

视图本地化

`IViewLocalizer` 服务可为视图提供本地化字符串。`ViewLocalizer` 类可实现此接口，并从视图文件路径找到资源位置。下面的代码演示如何使用 `IViewLocalizer` 的默认实现：

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

 @{
     ViewData["Title"] = Localizer["About"];
 }
 <h2>@ViewData["Title"].</h2>
 <h3>@ViewData["Message"]</h3>

 <p>@Localizer["Use this area to provide additional information."]</p>
```

`IViewLocalizer` 的默认实现可根据视图的文件名查找资源文件。没有使用全局共享资源文件的选项。`ViewLocalizer` 使用 `IHtmlLocalizer` 实现本地化工具，因此 Razor 不会对本地化字符串进行 HTML 编码。你可以参数化资源字符串，`IViewLocalizer` 将对参数进行 HTML 编码，但不会对资源字符串进行。请考虑以下 Razor 标记：

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

法语资源文件可以包含以下信息：

键	“值”
<i>Hello</i> {0}!	<i>Bonjour</i> {0} !

呈现的视图可能包含资源文件中的 HTML 标记。

注意：你通常只想要本地化文本，而不是 HTML。

若要在视图中使用共享资源文件，请注入 `IHtmlLocalizer<T>`：

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.StarterWeb.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

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

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

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations 本地化

DataAnnotations 错误消息已使用 `IStringLocalizer<T>` 本地化。使用选项 `ResourcesPath = "Resources"`，`RegisterViewModel` 中的错误消息可以存储在以下路径之一：

- `Resources/ViewModels/Account/RegisterViewModel.fr.resx`
- `Resources/ViewModels/Account/RegisterViewModel.fr.resx`

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

在 ASP.NET Core MVC 1.1.0 和更高版本中，非验证属性已经进行了本地化。ASP.NET Core MVC 1.0 不会为非验证属性查找本地化字符串。

对多个类使用一个资源字符串

下面的代码演示如何针对具有多个类的验证属性使用一个资源字符串：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

在上面的代码中，`SharedResource` 是对应于存储验证消息的 resx 的类。使用此方法，DataAnnotations 将仅使用 `SharedResource`，而不是每个类的资源。

为支持的语言和区域性提供本地化资源

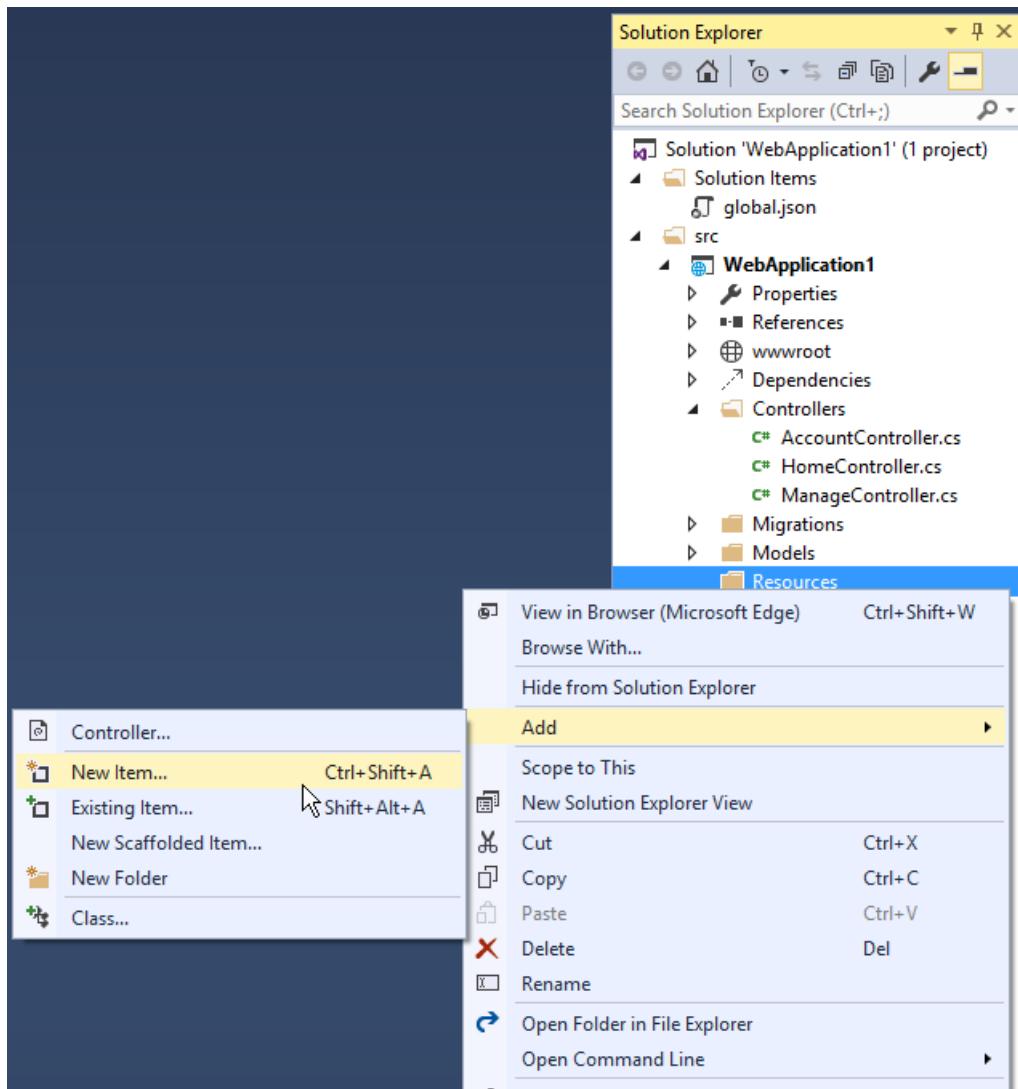
SupportedCultures 和 SupportedUICultures

ASP.NET Core 允许指定两个区域性值，`SupportedCultures` 和 `SupportedUICultures`。`SupportedCultures` 的 `CultureInfo` 对象可决定区域性相关函数的结果，如日期、时间、数字和货币格式等。`SupportedCultures` 确定文本的排序顺序、大小写约定和字符串比较。请参阅 [CultureInfo.CurrentCulture](#) 详细了解服务器如何获取区域性。`SupportedUICultures` 可确定哪些转换字符串(.resx 文件中)按 `ResourceManager` 查找。`ResourceManager` 只需查找 `CurrentUICulture` 决定的区域性特定字符串。`.NET` 中的每个线程都具有 `CurrentCulture` 和 `CurrentUICulture` 对象。呈现区域性相关函数时，ASP.NET Core 可检查这些值。例如，如果当前线程的区域性设置为“en-US”(英语，美国)，`DateTime.Now.ToString()` 将显示“Thursday, February 18, 2016”，但如果 `CurrentCulture` 设置为“es-ES”(西班牙语，西班牙)，则输出将为“jueves, 18 de febrero de 2016”。

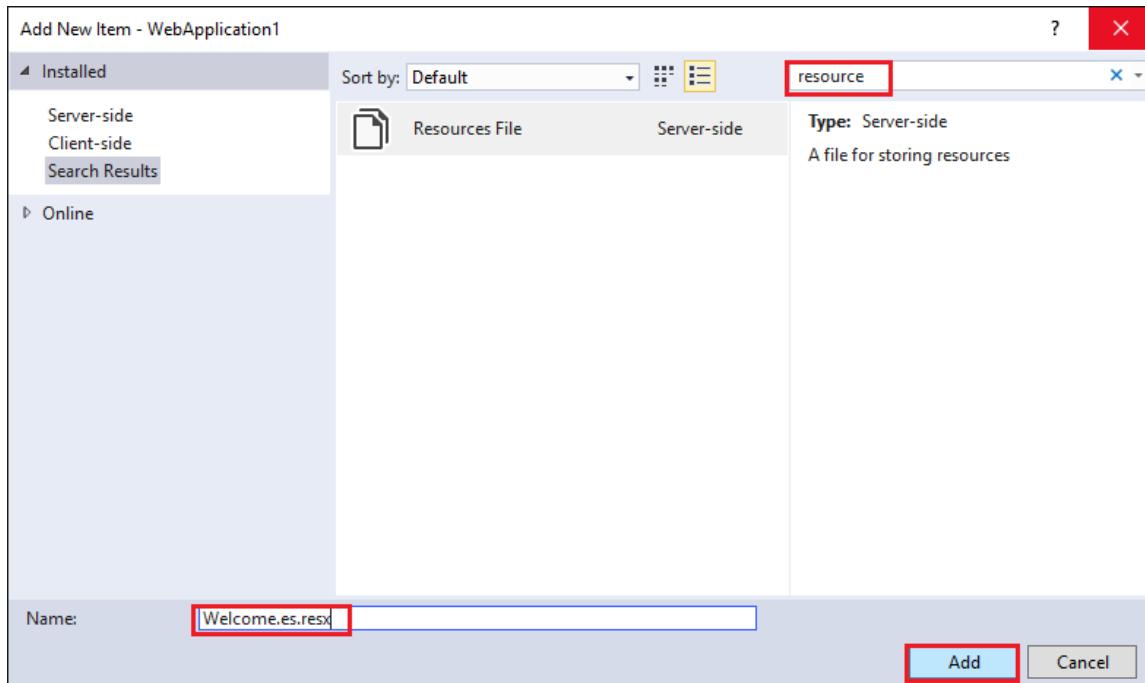
资源文件

资源文件是将可本地化的字符串与代码分离的有用机制。非默认语言的转换字符串是独立的.resx 资源文件。例如，你可能想要创建包含转换字符串、名为 Welcome.es.resx 的西班牙语资源文件。“es”是西班牙语的语言代码。要在 Visual Studio 中创建此资源文件，请支持以下操作：

1. 在“解决方案资源管理器”中，右键单击将包含资源文件的文件夹 > “添加” > “新项”。



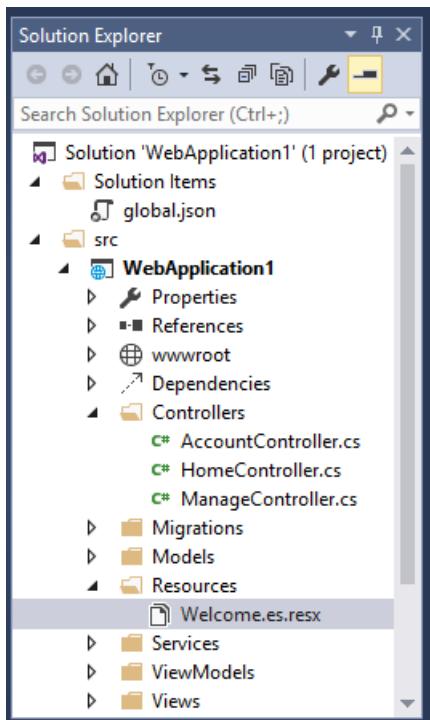
2. 在“搜索已安装的模板”框中，输入“资源”并命名该文件。



3. 在“名称”列中输入键值(本机字符串)，在“值”列中输入转换字符串。

	Name	Value	Comment
*	Hello	Hola	

Visual Studio 将显示 Welcome.es.resx 文件。



资源文件命名

资源名称是类的完整类型名称减去程序集名称。例如，类 `LocalizationWebsite.Web.Startup` 的主要程序集为 `LocalizationWebsite.Web.dll` 的项目中的法语资源将命名为 `Startup.fr.resx`。类 `LocalizationWebsite.Web.Controllers.HomeController` 的资源将命名为 `Controllers.HomeController.fr.resx`。如果目标类的命名空间与将需要完整类型名称的程序集名称不同。例如，在示例项目中，类型 `ExtraNamespace.Tools` 的资源将命名为 `ExtraNamespace.Tools.fr.resx`。

在示例项目中，`ConfigureServices` 方法将 `ResourcesPath` 设置为“资源”，因此主控制器的法语资源文件的项目相对路径是 `Resources/Controllers.HomeController.fr.resx`。或者，你可以使用文件夹组织资源文件。对于主控制器，该路径将为 `Resources/Controllers/HomeController.fr.resx`。如果不使用 `ResourcesPath` 选项，`.resx` 文件将转到项目的基目录中。`HomeController` 的资源文件将命名为 `Controllers.HomeController.fr.resx`。选择使用圆点或路径命名约定具体取决于你想如何组织资源文件。

资源名称	圆点或路径命名
<code>Resources/Controllers.HomeController.fr.resx</code>	圆点

资源名称	圆点或路径命名
Resources/Controllers/HomeController.fr.resx	路径

Razor 视图中使用 `@inject IViewLocalizer` 的资源文件遵循类似的模式。可以使用圆点命名或路径命名约定对视图的资源文件进行命名。Razor 视图资源文件可模拟其关联视图文件的路径。假设我们将 `ResourcesPath` 设置为“资源”，与 `Views/Home/About.cshtml` 视图关联的法语资源文件可能是下面其中之一：

- `Resources/Views/Home/About.fr.resx`
- `Resources/Views.Home.About.fr.resx`

如果不使用 `ResourcesPath` 选项，视图的 `.resx` 文件将位于视图所在的文件夹。

区域性回退行为

在搜索资源时，本地化会进行“区域性回退”。从所请求的区域性开始，如果未能找到，则还原至该区域性的父区域性。另外，`CultureInfo.Parent` 属性代表父区域性。这通常（但并不是总是）意味着从 ISO 中移除区域签名。例如，墨西哥的西班牙语方言为“es-MX”。它具备一个父级“es”西班牙，没有特别指定国家。

假设你的站点接收到一个区域性为“fr-CA”的“Welcome”资源的请求。本地化系统按顺序查找以下资源，并选择第一个匹配项：

- `Welcome.fr-CA.resx`
- `Welcome.fr.resx`
- `Welcome.resx`（如果 `NeutralResourcesLanguage` 为“fr-CA”）

例如，如果删除了“.fr”区域性指示符，而且已将区域性设置为“法语”，将读取默认资源文件，并本地化字符串。对于不满足所请求区域性的情况，资源管理器可指定默认资源或回退资源。缺少适用于请求区域性的资源时，如果只想返回键，不得具有默认资源文件。

使用 Visual Studio 生成资源文件

如果在 Visual Studio 中创建文件名没有区域性的资源文件（例如 `Welcome.resx`），Visual Studio 将创建一个 C# 类，并且具有每个字符串的属性。这通常不是你想在 ASP.NET Core 中使用的。你通常没有默认的 `.resx` 资源文件（没有区域性名称的 `.resx` 文件）。建议创建具有区域性名称（例如 `Welcome.fr.resx`）的 `.resx` 文件。创建具有区域性名称的 `.resx` 文件时，Visual Studio 不会生成类文件。我们预计许多开发者不会创建默认语言资源文件。

添加其他区域性

每个语言和区域性组合（除默认语言外）都需要唯一资源文件。通过新建 ISO 语言代码属于名称一部分的资源文件，为不同的区域性和区域设置创建资源文件（例如，`en-us`、`fr-ca` 和 `en-gb`）。这些 ISO 编码位于文件名和 `.resx` 文件扩展之间，如 `Welcome.es-MX.resx`（西班牙语/墨西哥）。

实施策略，为每个请求选择语言/区域性

配置本地化

通过 `ConfigureServices` 方法配置本地化：

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` 将本地化服务添加到服务容器。上面的代码还可将资源路径设置为“资源”。

- `AddViewLocalization` 添加对本地化视图文件的支持。在此示例视图中，本地化基于视图文件后缀。例如，`Index.fr.cshtml` 文件中的“fr”。
- `AddDataAnnotationsLocalization` 添加通过 `IStringLocalizer` 抽象对本地化 `DataAnnotations` 验证消息的支持。

本地化中间件

在本地化中间件中设置有关请求的当前区域性。在 `Configure` 方法中启用本地化中间件。必须在中间件前面配置本地化中间件，它可能检查请求区域性（例如，`app.UseMvcWithDefaultRoute()`）。

```
var supportedCultures = new[]
{
    new CultureInfo(enUSCulture),
    new CultureInfo("en-AU"),
    new CultureInfo("en-GB"),
    new CultureInfo("en"),
    new CultureInfo("es-ES"),
    new CultureInfo("es-MX"),
    new CultureInfo("es"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
};

app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture(enUSCulture),
    // Formatting numbers, dates, etc.
    SupportedCultures = supportedCultures,
    // UI strings that we have localized.
    SupportedUICultures = supportedCultures
});

app.UseStaticFiles();
// To configure external authentication,
// see: http://go.microsoft.com/fwlink/?LinkID=532715
app.UseAuthentication();
app.UseMvcWithDefaultRoute();
```

`UseRequestLocalization` 初始化 `RequestLocalizationOptions` 对象。在每个请求上，枚举了 `RequestLocalizationOptions` 的 `RequestCultureProvider` 列表，使用了可成功决定请求区域性的第一个提供程序。默认提供程序来自 `RequestLocalizationOptions` 类：

1. `QueryStringRequestCultureProvider`
2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

默认列表从最具体到最不具体排序。在本文的后面部分，我们将了解如何更改顺序，甚至添加一个自定义区域性提供程序。如果没有一个提供程序可以确定请求区域性，则使用 `DefaultRequestCulture`。

QueryStringRequestCultureProvider

某些应用将使用查询字符串来设置**区域性和 UI 区域性**。对于使用 Cookie 或接受语言标题方法的应用，向 URL 添加查询字符串有助于调试和测试代码。默认情况下，`QueryStringRequestCultureProvider` 注册为 `RequestCultureProvider` 列表中的第一个本地化提供程序。传递查询字符串参数 `culture` 和 `ui-culture`。下面的示例将特定区域性（语言和区域）设置为“西班牙语/墨西哥”：

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

如果仅传入两种区域性之一（`culture` 或 `ui-culture`），查询字符串提供程序将使用你传入的区域性设置这两个值。例如，仅设置区域性将同时设置 `Culture` 和 `UICulture`：

<http://localhost:5000/?culture=es-MX>

CookieRequestCultureProvider

通常，生产应用将提供一种机制来使用 ASP.NET Core 区域性 Cookie 设置区域性。若要创建 Cookie，请使用 `MakeCookieValue` 方法。

`CookieRequestCultureProvider` `DefaultCookieName` 将返回用来跟踪用户首选区域性信息的默认 Cookie 名称。默认 Cookie 名称是 `.AspNetCore.Culture`。

Cookie 格式为 `c=%LANGCODE%|uic=%LANGCODE%`，其中 `c` 是 `Culture`，`uic` 是 `UICulture`，例如：

```
c=en-UK|uic=en-US
```

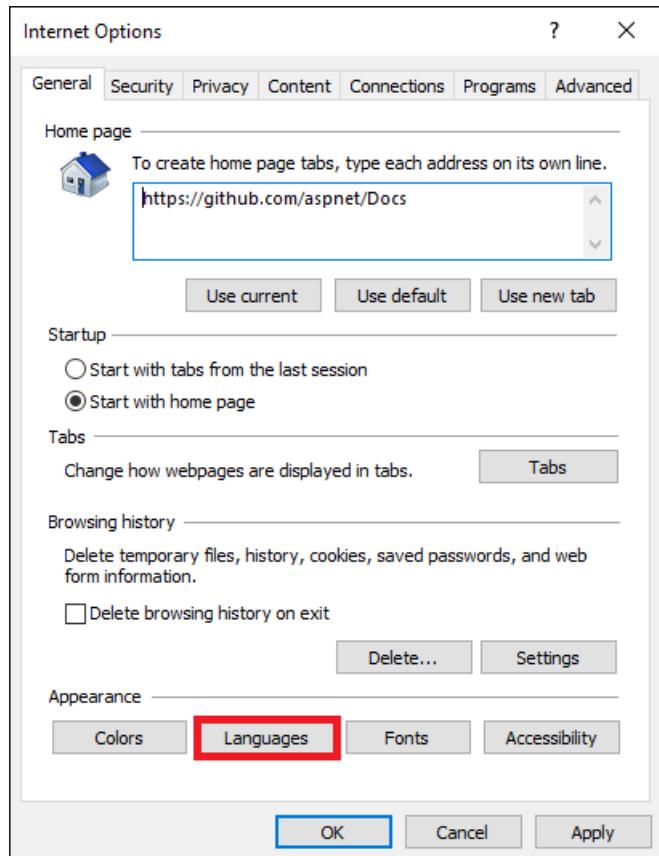
如果仅指定其中一个区域性信息和 UI 区域性，则指定的区域性将同时用于区域性信息和 UI 区域性。

接受语言 HTTP 标题

接受语言标题在大多数浏览器中可设置，最初用于指定用户的语言。此设置指示浏览器已设置为发送或已从基础操作系统继承的内容。浏览器请求的接受语言 HTTP 标题不是检测用户首选语言的可靠方法（请参阅 [Setting language preferences in a browser](#)（在浏览器中设置首选项）。生产应用应包括一种用户可以自定义区域性选择的方法。

在 IE 中设置接受语言 HTTP 标题

1. 在齿轮图标中，点击“Internet 选项”。
2. 点击“语言”。



3. 点击“设置语言首选项”。
4. 点击“添加语言”。
5. 添加语言。
6. 点击语言，然后点击“向上移动”。

使用自定义提供程序

假设你想要让客户在数据库中存储其语言和区域性。你可以编写一个提供程序来查找用户的这些值。下面的代码演示如何添加自定义提供程序：

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUITcultures = supportedCultures;

    options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});
});
```

使用 `RequestLocalizationOptions` 添加或删除本地化提供程序。

以编程方式设置区域性

[GitHub](#) 上的示例项目 `Localization.StarterWeb` 包含设置 `Culture` 的 UI。

`Views/Shared/_SelectLanguagePartial.cshtml` 文件允许你从支持的区域性列表中选择区域性：

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

 @{
    var requestCulture = Context.Features.Get< IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUITcultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
        asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
        method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
        <select name="culture"
            onchange="this.form.submit();"
            asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

`Views/Shared/_SelectLanguagePartial.cshtml` 文件添加到了布局文件的 `footer` 部分，使它将可供所有视图使用：

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization.StarterWeb</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

`SetLanguage` 方法可设置区域性 Cookie。

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

不能将 `_SelectLanguagePartial.cshtml` 插入此项目的示例代码。GitHub 上的项目 Localization.StarterWeb 包含代码，可部分通过 [依赖关系注入](#) 容器将 `RequestLocalizationOptions` 流到 Razor。

全球化和本地化术语

本地化应用的过程还要求基本了解现代软件开发中常用的相关字符集，以及与之相关的问题。尽管所有计算机将文本都存储为数字（代码），但不同的系统使用不同的数字存储相同的文本。本地化过程是指针对特定区域性/区域设置转换应用的用户界面（UI）。

[本地化性](#) 是一个中间过程，用于验证全球化应用是否准备好进行本地化。

区域性名称的 [RFC 4646](#) 格式为 `<languagecode2>-<country/regioncode2>`，其中 `<languagecode2>` 是语言代码，`<country/regioncode2>` 是子区域性代码。例如，`es-CL` 表示西班牙语（智利），`en-US` 表示英语（美国），而 `en-AU` 表示英语（澳大利亚）。[RFC 4646](#) 是一个与语言相关的 ISO 639 双小写字母的区域性代码和一个与国家/地区相关的 ISO 3166 双大写字母子区域性代码的组合。请参阅[语言区域性名称](#)。

国际化常缩写为“i18N”。缩写采用第一个和最后一个字母以及它们之间的字母数，因此 18 代表第一个字母“l”和最后一个“N”之间的字母数。这同样适用于全球化（G11N）和本地化（L10N）。

术语：

- 全球化（G11N）：使应用支持不同语言和区域的过程。
- 本地化（L10N）：针对给定语言和区域，自定义应用的过程。
- 国际化（I18N）：同时描绘全球化和本地化。
- 区域性：它是一种语言和区域（可选）。
- 非特定区域性：具有指定语言但不具有区域的区域性。（例如，“en”，“es”）
- 特定区域性：具有指定语言和区域的区域性。（例如，“en-US”，“en-GB”，“es-CL”）
- 父区域性：包含特定区域性的非特定区域性。（例如，“en”是“en-US”和“en-GB”的父区域性）
- 区域设置：区域设置与区域性相同。

其他资源

- 本文所用的 [Localization Starter Web 项目](#)。
- Visual Studio 中的 [资源文件](#)
- [.resx 文件中的资源](#)
- [Microsoft 多语言应用工具包](#)

在 ASP.NET Core 中配置可移植对象本地化

2018/5/14 • 8 min to read • [Edit Online](#)

作者: Sébastien Ros 和 Scott Addie

本文演示通过 [Orchard Core](#) 框架在 ASP.NET Core 应用程序中使用可移植对象 (PO) 文件的步骤。

请注意: Orchard Core 不是 Microsoft 产品。因此, Microsoft 不提供针对此功能的支持。

[查看或下载示例代码\(如何下载\)](#)

什么是 PO 文件?

PO 文件作为包含给定语言的已转换字符串的文本文件分发。使用 PO 文件替代 .resx 文件的一些优势包括:

- PO 文件支持复数形式;而 .resx 文件不支持复数形式。
- PO 文件的编译方法与 .resx 文件不同。同样, 无需专用工具和生成步骤。
- PO 文件可很好地与协作联机编辑工具结合使用。

示例

下面是一个包含两个法语字符串(其中一个具有复数形式)转换的示例 PO 文件:

fr.po

```
#: Services/EmailService.cs:29
msgid "Enter a comma separated list of email addresses."
msgstr "Entrez une liste d'emails séparés par une virgule.

#: Views/Email.cshtml:112
msgid "The email address is \"{0}\"."
msgid_plural "The email addresses are \"{0}\"."
msgstr[0] "L'adresse email est \"{0}\"."
msgstr[1] "Les adresses email sont \"{0}\""
```

此示例使用下列语法:

- `#:` :注释, 用于指示要转换的字符串的上下文。根据使用的位置, 可对相同字符串进行不同转换。
- `msgid` :未转换的字符串。
- `msgstr` :已转换的字符串。

在支持复数形式的情况下, 可定义多个条目。

- `msgid_plural` :未转换的复数形式字符串。
- `msgstr[0]` :针对事例 0 的已转换的字符串。
- `msgstr[N]` :针对事例 N 的已转换的字符串。

可在[此处](#)找到 PO 文件规范。

在 ASP.NET Core 中配置 PO 文件支持

此示例基于从 Visual Studio 2017 项目模板中生成的 ASP.NET Core MVC 应用程序。

引用包

添加对 `OrchardCore.Localization.Core` NuGet 包的引用。它可在 [MyGet](#) 上的以下包源中获得:

<https://www.myget.org/F/orchardcore-preview/api/v3/index.json>

.csproj 文件现在包含类似于以下内容的行(版本号可能不同)：

```
<PackageReference Include="OrchardCore.Localization.Core" Version="1.0.0-beta1-3187" />
```

注册服务

将所需服务添加到 Startup.cs 的 `ConfigureServices` 方法：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);

    services.AddPortableObjectLocalization();

    services.Configure<RequestLocalizationOptions>(options =>
    {
        var supportedCultures = new List<CultureInfo>
        {
            new CultureInfo("en-US"),
            new CultureInfo("en"),
            new CultureInfo("fr-FR"),
            new CultureInfo("fr")
        };

        options.DefaultRequestCulture = new RequestCulture("en-US");
        options.SupportedCultures = supportedCultures;
        options.SupportedUICultures = supportedCultures;
    });
}
```

将所需中间件添加到 Startup.cs 的 `Configure` 方法：

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

    app.UseStaticFiles();

    app.UseRequestLocalization();

    app.UseMvcWithDefaultRoute();
}
```

将以下代码添加到所选的 Razor 视图中。在此示例中，使用了 About.cshtml。

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer

<p>@Localizer["Hello world!"]</p>
```

注入了 `IViewLocalizer` 实例并将其用于转换文本“Hello world！”。

创建 PO 文件

在应用程序根文件夹中创建名为 .po 的文件。在此示例中，文件名为 fr.po，因为使用了法语：

```
msgid "Hello world!"  
msgstr "Bonjour le monde!"
```

此文件存储了要转换的字符串和已转换为法语的字符串。如有必要，转换将还原为其父级区域性。在此示例中，如果请求的区域性为 `fr-FR` 或 `fr-CA`，则使用 fr.po 文件。

测试应用程序

运行应用程序并导航到 URL `/Home/About`。此时将显示文本 Hello world!。

导航到 URL `/Home/About?culture=fr-FR`。此时将显示文本 Bonjour le monde!。

复数形式

PO 文件支持复数形式，在相同字符串需要基于基数以不同方式进行转换时，这非常有用。此任务较为复杂，因为每种语言均定义了自定义规则，以基于基数选择要使用的字符串。

Orchard 本地化包提供了一个 API 以自动调用这些不同的复数形式。

创建复数形式 PO 文件

将以下内容添加到前面所述的 fr.po 文件：

```
msgid "There is one item."  
msgid_plural "There are {0} items."  
msgstr[0] "Il y a un élément."  
msgstr[1] "Il y a {0} éléments."
```

有关此示例中每个条目所表示的内容的说明，请参阅[什么是 PO 文件？](#)。

使用不同的复数形式添加语言

前面的示例中使用了英语和法语字符串。英语和法语只有两种复数形式且拥有相同形式规则，即一的基数映射到第一种复数形式。任何其他基数映射到第二种复数形式。

并非所有语言都拥有相同的规则。捷克语就是一个例子，它具有三种复数形式。

如下所示，创建 `cs.po` 文件，并记下复数形式如何需要三种不同转换：

```
msgid "Hello world!"  
msgstr "Ahoj světe!"  
  
msgid "There is one item."  
msgid_plural "There are {0} items."  
msgstr[0] "Existuje jedna položka."  
msgstr[1] "Existují {0} položky."  
msgstr[2] "Existuje {0} položek."
```

若要接受捷克语本地化，请将 `"cs"` 添加到 `ConfigureServices` 方法中受支持的区域性列表：

```
var supportedCultures = new List<CultureInfo>
{
    new CultureInfo("en-US"),
    new CultureInfo("en"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
    new CultureInfo("cs")
};
```

编辑 Views/Home/About.cshtml 文件以呈现一些基数的已本地化复数形式字符串：

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

注意：在实际方案中，变量将用于表示计数。此处，我们通过三个不同的值重复相同代码，以公开非常特定的事例。

切换区域性时，将显示如下内容：

对于 `/Home/About`：

```
There is one item.
There are 2 items.
There are 5 items.
```

对于 `/Home/About?culture=fr`：

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

对于 `/Home/About?culture=cs`：

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

请注意，对于捷克语区域性，这三种转换各不相同。对于最后两个已转换字符串，法语和英语区域性具有相同构造。

高级任务

将字符串置于上下文中理解

应用程序通常包含要在多个位置中进行转换的字符串。在应用中的特定位置(Razor 视图或类文件)，相同字符串可能具有不同转换。PO 文件支持文件上下文概念，此概念可用于对所表示的字符串进行分类。使用文件上下文，可将字符串进行不同转换，具体取决于文件上下文(或缺乏文件上下文)。

PO 本地化服务使用完整类的名称或转换字符串时使用的视图。这通过在 `msgctxt` 条目上设置值来完成。

请考虑对以前的 fr.po 示例作一点小小的补充。可通过设置保留的 `msgctxt` 条目的值将位于 Views/Home/About.cshtml 的 Razor 视图定义为文件上下文：

```
msgctxt "Views.Home.About"
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

这样设置 `msgctxt` 后，导航到 `/Home/About?culture=fr-FR` 时将发生文本转换。而导航到 `/Home/Contact?culture=fr-FR` 时，则不发生转换。

当没有特定条目与给定文件上下文相匹配时，Orchard Core 的回退机制将在没有上下文的情况下查找适当的 PO 文件。假设不存在针对 `Views/Home/Contact.cshtml` 定义的特定文件上下文，导航到 `/Home/Contact?culture=fr-FR`，加载 PO 文件，如：

```
msgid "Hello world!"  
msgstr "Bonjour le monde!"
```

更改 PO 文件的位置

可以在 `ConfigureServices` 中更改 PO 文件的默认位置：

```
services.AddPortableObjectLocalization(options => options.ResourcesPath = "Localization");
```

在此示例中，从本地化文件夹加载 PO 文件。

实现用于查找本地化文件的自定义逻辑

当需要更复杂的逻辑以查找 PO 文件时，可实现

`OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` 接口并将其注册为服务。在可将 PO 文件存储于不同位置或在文件夹层次结构中找到文件时，这非常有用。

使用不同默认复数形式语言

此包包含特定于两种复数形式的 `Plural` 扩展方法。对于需要更多复数形式的语言，请创建扩展方法。通过扩展方法，无需提供默认语言的任何本地化文件 — 可在代码中直接使用原始字符串。

可使用更加广泛的、接受转换的字符串数组的 `Plural(int count, string[] pluralForms, params object[] arguments)` 重载。

启动 HTTP 请求

2018/5/14 • 14 min to read • [Edit Online](#)

作者: [Glenn Condon](#) [Ryan Nowak](#) 和 [Steve Gordon](#)

注意

ASP.NET Core 2.1 is in preview and not recommended for production use.

可以注册 `IHttpClientFactory` 并将其用于配置和创建应用中的 `HttpClient` 实例。这能带来以下好处：

- 提供一个中心位置，用于命名和配置逻辑 `HttpClient` 实例。例如，可以注册一个“github”客户端，将其配置为访问 GitHub。可以注册一个默认客户端用于其他用途。
- 通过委托 `HttpClient` 中的处理程序整理出站中间件的概念，并提供适用于基于 Polly 的中间件的扩展来利用概念。
- 管理基础 `HttpClientMessageHandler` 实例的池和生存期，避免在手动管理 `HttpClient` 生存期时出现常见的 DNS 问题。
- (通过 `ILogger`) 添加可配置的记录体验，以处理工厂创建的客户端发送的所有请求。

消耗模式

在应用中可以通过以下多种方式使用 `IHttpClientFactory`：

- [基本用法](#)
- [命名客户端](#)
- [类型化客户端](#)
- [生成的客户端](#)

它们之间不存在严格的优先级。最佳方法取决于应用的约束条件。

基本用法

在 `Startup.cs` 的 `ConfigureServices` 方法中，通过在 `IServiceCollection` 上调用 `AddHttpClient` 扩展方法可以注册 `IHttpClientFactory`。

```
services.AddHttpClient();
```

注册后，在可以使用[依赖关系注入 \(DI\)](#) 注入服务的任何位置，代码都能接受 `IHttpClientFactory`。

`IHttpClientFactory` 可以用于创建 `HttpClient` 实例：

```

public class BasicUsageModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubBranch> Branches { get; private set; }

    public bool GetBranchesError { get; private set; }

    public BasicUsageModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
"https://api.github.com/repos/aspnet/docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = _clientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            Branches = await response.Content.ReadAsAsync<IEnumerable<GitHubBranch>>();
        }
        else
        {
            GetBranchesError = true;
            Branches = Array.Empty<GitHubBranch>();
        }
    }
}

```

以这种方式使用 `IHttpClientFactory` 非常适合重构现有应用。这不会影响 `HttpClient` 的使用方式。在当前创建 `HttpClient` 实例的位置，使用对 `CreateClient` 的调用替换这些匹配项。

命名客户端

如果应用需要区别使用多个 `HttpClient`（每个的配置都不同），可以选择使用“命名客户端”。可以在 `HttpClient` 中注册时指定命名 `ConfigureServices` 的配置。

```

services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json"); // Github API versioning
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample"); // Github requires a user-agent
});

```

上述代码调用了 `AddHttpClient`，并提供名称“github”。此客户端应用了一些默认配置，也就是需要基址和两个标头来使用 GitHub API。

每次调用 `CreateClient` 时，都会创建 `HttpClient` 的新实例，并调用配置操作。

要使用命名客户端，可将字符串参数传递到 `CreateClient`。指定要创建的客户端的名称：

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubPullRequest> PullRequests { get; private set; }

    public bool GetPullRequestsError { get; private set; }

    public bool HasPullRequests => PullRequests.Any();

    public NamedClientModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get, "repos/aspnet/docs/pulls");

        var client = _clientFactory.CreateClient("github");

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            PullRequests = await response.Content.ReadAsAsync<IEnumerable<GitHubPullRequest>>();
        }
        else
        {
            GetPullRequestsError = true;
            PullRequests = Array.Empty<GitHubPullRequest>();
        }
    }
}
```

在上述代码中，请求不需要指定主机名。可以仅传递路径，因为采用了为客户端配置的基址。

类型化客户端

类型化客户端提供与命名客户端一样的功能，不需要将字符串用作密钥。类型化客户端方法在使用客户端时提供 IntelliSense 和编译器帮助。它们提供单个地址来配置特定 `HttpClient` 并与其进行交互。例如，单个类型化客户端可能用于单个后端终结点，并封装此终结点的所有处理逻辑。另一个优势是它们使用 DI 且可以被注入到应用中需要的位置。

类型化客户端在构造函数中接收 `HttpClient` 参数：

```
public class GitHubService
{
    public HttpClient Client { get; }

    public GitHubService(HttpClient client)
    {
        client.BaseAddress = new Uri("https://api.github.com/");
        client.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json"); // GitHub API versioning
        client.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample"); // GitHub requires a user-agent

        Client = client;
    }

    public async Task<IEnumerable<GitHubIssue>> GetAspNetDocsIssues()
    {
        var response = await Client.GetAsync("/repos/aspnet/docs/issues?state=open&sort=created&direction=desc");

        response.EnsureSuccessStatusCode();

        var result = await response.Content.ReadAsAsync<IEnumerable<GitHubIssue>>();

        return result;
    }
}
```

在上述代码中，配置转移到了类型化客户端中。`HttpClient` 对象公开为公共属性。可以定义公开 `HttpClient` 功能的特定于 API 的方法。`GetLatestDocsIssue` 方法从 GitHub 存储库封装查询和分析最新问题所需的代码。

要注册类型化客户端，可在 `ConfigureServices` 中使用通用的 `AddHttpClient` 扩展方法，指定类型化客户端类：

```
services.AddHttpClient<GitHubService>();
```

使用 DI 将类型客户端注册为暂时客户端。可以直接插入或使用类型化客户端：

```

public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public IEnumerable<GitHubIssue> LatestIssues { get; private set; }

    public bool HasIssue => LatestIssues.Any();

    public bool GetIssuesError { get; private set; }

    public TypedClientModel(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    public async Task OnGet()
    {
        try
        {
            LatestIssues = await _gitHubService.GetAspNetDocsIssues();
        }
        catch(HttpRequestException)
        {
            GetIssuesError = true;
            LatestIssues = Array.Empty<GitHubIssue>();
        }
    }
}

```

根据你的喜好，可以在 `ConfigureServices` 中注册时指定类型化客户端的配置，而不是在类型化客户端的构造函数中指定：

```

services.AddHttpClient<RepoService>(c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});

```

可以将 `HttpClient` 完全封装在类型化客户端中。不是将它公开为属性，而是可以提供公共方法，用于在内部调用 `HttpClient`。

```

public class RepoService
{
    private readonly HttpClient _httpClient; // not exposed publicly

    public RepoService(HttpClient client)
    {
        _httpClient = client;
    }

    public async Task<IEnumerable<string>> GetRepos()
    {
        var response = await _httpClient.GetAsync("aspnet/repos");

        response.EnsureSuccessStatusCode();

        var result = await response.Content.ReadAsAsync<IEnumerable<string>>();

        return result;
    }
}

```

在上述代码中，`HttpClient` 存储未私有字段。进行外部调用的所有访问都经由 `GetRepos` 方法。

生成的客户端

`IHttpClientFactory` 可结合其他第三方库(例如 [Refit](#))使用。Refit 是.NET 的 REST 库。它将 REST API 转换为实时接口。`RestService` 动态生成该接口的实现，使用 `HttpClient` 进行外部 HTTP 调用。

定义了接口和答复来代表外部 API 及其响应：

```
public interface IHelloClient
{
    [Get("/helloworld")]
    Task<Reply> GetMessageAsync();
}

public class Reply
{
    public string Message { get; set; }
}
```

可以添加类型化客户端，使用 Refit 生成实现：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("hello", c =>
    {
        c.BaseAddress = new Uri("http://localhost:5000");
    })
    .AddTypedClient(c => Refit.RestService.For<IHelloClient>(c));

    services.AddMvc();
}
```

可以在必要时使用定义的接口，以及由 DI 和 Refit 提供的实现：

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHelloClient _client;

    public ValuesController(IHelloClient client)
    {
        _client = client;
    }

    [HttpGet("/")]
    public async Task<ActionResult<Reply>> Index()
    {
        return await _client.GetMessageAsync();
    }
}
```

出站请求中间件

`HttpClient` 已经具有委托处理程序的概念，这些委托处理程序可以链接在一起，处理出站 HTTP 请求。

`IHttpClientFactory` 可以轻松定义处理程序并应用于每个命名客户端。它支持注册和链接多个处理程序，以生成出站请求中间件管道。每个处理程序都可以在出站请求前后执行工作。此模式类似于 ASP.NET Core 中的入站中间件管道。此模式提供了一种用于管理围绕 HTTP 请求的横切关注点的机制，包括缓存、错误处理、序列化以及日志记录。

要创建处理器，请定义一个派生自 `DelegatingHandler` 的类。重写 `SendAsync` 方法，在将请求传递至管道中的下一个处理器之前执行代码：

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest);
        }
        Content = new StringContent("You must supply an API key header called X-API-KEY");
    }

    return await base.SendAsync(request, cancellationToken);
}
}
```

上述代码定义了基本处理器。它会在遇到请求时检查是否包括 X-API-KEY 标头。如果标头缺失，它可以避免 HTTP 调用，并返回合适的响应。

在注册期间可将一个或多个标头添加到 `HttpClient` 的配置。此任务通过 `IHttpClientBuilder` 上的扩展方法完成。

```
services.AddTransient<ValidateHeaderHandler>();

services.AddHttpClient("externalservice", c =>
{
    c.BaseAddress = new Uri("https://localhost:5000/"); // assume this is an "external" service which requires
    an API KEY
})
.AddHttpMessageHandler<ValidateHeaderHandler>();
```

在上述代码中通过 DI 注册了 `ValidateHeaderHandler`。处理器必须在 DI 中注册为临时处理器。注册后可以调用 `AddHttpMessageHandler`，传入标头的类型。

可以按处理器应该执行的顺序注册多个处理器。每个处理器都会覆盖下一个处理器，直到最终 `HttpClientHandler` 执行请求：

```
services.AddTransient<SecureRequestHandler>();
services.AddTransient<requestDataHandler>();

services.AddHttpClient("clientwithhandlers")
    // This handler is on the outside and called first during the request, last during the response.
    .AddHttpMessageHandler<SecureRequestHandler>()
    // This handler is on the inside, closest to the request being sent.
    .AddHttpMessageHandler<requestDataHandler>();
```

使用基于 Polly 的处理器

`IHttpClientFactory` 与一个名为 [Polly](#) 的热门第三方库集成。Polly 是适用于 .NET 的全面恢复和临时故障处理库。开发人员通过它可以表达策略，例如以流畅且线程安全的方式处理重试、断路器、超时、Bulkhead 隔离和回退。

提供了扩展方法，以实现将 Polly 策略用于配置的 `HttpClient` 实例。在名为“Microsoft.Extensions.Http.Polly”的 NuGet 包中可以使用 Polly 扩展。“Microsoft.AspNetCore.App”元包在默认情况下不包含此包。若要使用扩展，需要将 `PackageReference` 显式包含在项目中。

```

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

<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" Version="2.1.0-rc1-final" />
  <PackageReference Include="Microsoft.Extensions.Http.Polly" Version="2.1.0-rc1-final" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.1.0-rc1-final" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.0-rc1-final" />
</ItemGroup>

</Project>

```

还原此包后，可以使用扩展方法来支持将基于 Polly 的处理程序添加至客户端。

处理临时故障

在执行外部 HTTP 调用时，最可能遇到的故障是临时故障。包含了一种简便的扩展方法，该方法名为 `AddTransientHttpErrorHandler`，允许定义策略来处理临时故障。使用这种扩展方法配置的策略可以处理 `HttpRequestException`、HTTP 5xx 响应以及 HTTP 408 响应。

`AddTransientHttpErrorHandler` 扩展可在 `ConfigureServices` 内使用。该扩展可以提供 `PolicyBuilder` 对象的访问权限，该对象配置为处理表示可能的临时故障的错误：

```

services.AddHttpClient<UnreliableEndpointCallerService>()
    .AddTransientHttpErrorHandler(p => p.WaitAndRetryAsync(3, _ => TimeSpan.FromMilliseconds(600)));

```

上述代码中定义了 `WaitAndRetryAsync` 策略。请求失败后最多可以重试三次，每次尝试间隔 600 ms。

动态选择策略

存在其他扩展方法，可以用于添加基于 Polly 的处理程序。这类扩展的其中一个 `AddPolicyHandler`，它具备多个重载。一个重载允许在定义要应用的策略时检查该请求：

```

var timeout = Policy.TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(10));
var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")
// Run some code to select a policy based on the request
    .AddPolicyHandler(request => request.Method == HttpMethod.Get ? timeout : longTimeout);

```

在上述代码中，如果出站请求为 GET，则应用 10 秒超时。其他所有 HTTP 方法应用 30 秒超时。

添加多个 Polly 处理程序

嵌套 Polly 策略以增强功能是很常见的：

```

services.AddHttpClient("multiplepolicies")
    .AddTransientHttpErrorHandler(p => p.RetryAsync(3))
    .AddTransientHttpErrorHandler(p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));

```

在上述示例中，添加两个处理程序。第一个使用 `AddTransientHttpErrorHandler` 扩展添加重试策略。若请求失败，最多可重试三次。第一个调用 `AddTransientHttpErrorHandler` 添加断路器策略。如果尝试连续失败了五次，则会阻止后续外部请求 30 秒。断路器策略处于监控状态。通过此客户端进行的所有调用都共享同样的线路状态。

从 Polly 注册表添加策略

管理常用策略的一种方法是一次性定义它们并使用 `PolicyRegistry` 注册它们。提供了一种扩展方法，可以使用注

册表中的策略添加处理程序：

```
var registry = services.AddPolicyRegistry();

registry.Add("regular", timeout);
registry.Add("long", longTimeout);

services.AddHttpClient("regulartimeouthandler")
    .AddPolicyHandlerFromRegistry("regular");
```

在上述代码中，将 `PolicyRegistry` 添加到了 `ServiceCollection`，并使用它注册了两个策略。要使用注册表中的策略，请使用 `AddPolicyHandlerFromRegistry` 方法传递要应用的策略的名称。

要进一步了解 `IHttpClientFactory` 和 Polly 集成，请参考 [Polly Wiki](#)。

HttpClient 和生存期管理

每次在 `IHttpClientFactory` 上调用 `CreateClient` 都会返回一个新的 `HttpClient` 实例。每个命名客户端都会有一个 `HttpMessageHandler`。`IHttpClientFactory` 将汇集工厂创建的 `HttpMessageHandler` 实例，以减少资源消耗。如果 `HttpMessageHandler` 实例的生存期尚未过期，那么在创建新的 `HttpClient` 实例时，可能会从池中重用该实例。

由于每个处理程序通常都管理自己的基础 HTTP 连接，所以有必要汇集处理程序；创建的处理程序数量如果多于必需的数量，则可能导致连接延迟。部分处理程序还保持连接无期限地打开，这样可以防止处理程序对 DNS 更改作出反应。

处理程序的默认生存期为两分钟。可在每个命名客户端上重写默认值。要重写该值，请在创建客户端时在返回的 `IHttpClientBuilder` 上调用 `SetHandlerLifetime`：

```
services.AddHttpClient("extendedhandlerlifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

日志记录

通过 `IHttpClientFactory` 创建的客户端记录所有请求的日志消息。你将需要在日志记录配置中启用合适的信息级别，从而查看默认日志消息。仅在跟踪级别包含附加日志记录（例如请求标头的日志记录）。

用于每个客户端的日志类别包含客户端名称。例如，名为“`MyNamedClient`”的客户端会使用 `System.Net.Http.HttpClient.MyNamedClient.LogicalHandler` 类别记录消息。请求处理程序管道的外部会出现带有“`LogicalHandler`”后缀的消息。在请求时，在管道中的任何其他处理程序处理请求之前记录消息。在响应时，在任何其他管道处理程序接收响应之后记录消息。

在请求处理程序管道内部也会进行日志记录。在“`MyNamedClient`”的例子中，针对日志类别 `System.Net.Http.HttpClient.MyNamedClient.ClientHandler` 记录了这些消息。在请求时，在所有其他处理程序运行后，以及刚好在通过网络发出请求之前记录消息。在响应时，此日志记录包含响应在通过处理程序管道被传递回去之前的状态。

在管道内外启用日志记录，可以检查其他管道处理程序所作的更改。例如，其中可能包含对请求标头的更改，或者对响应状态代码的更改。

通过在日志类别中包含客户端名称，可以在必要时对特定的命名客户端筛选日志。

配置 `HttpMessageHandler`

控制客户端使用的内部 `HttpMessageHandler` 的配置是有必要的。

在添加命名客户端或类型化客户端时，会返回 `IHttpClientBuilder`。`ConfigurePrimaryHttpMessageHandler` 扩展方法

可以用于定义委托。委托用于创建和配置客户端使用的主要 `HttpMessageHandler`：

```
services.AddHttpClient("configured-inner-handler")
    .ConfigurePrimaryHttpMessageHandler(() =>
{
    return new HttpClientHandler()
    {
        AllowAutoRedirect = false,
        UseDefaultCredentials = true
    };
});
```

ASP.NET Core 中的请求功能

2018/2/8 • 3 min to read • [Edit Online](#)

作者: Steve Smith

与 HTTP 请求和响应相关的 Web 服务器实现详细信息在接口中定义。服务器实现和中间件使用这些接口来创建和修改应用程序的托管管道。

功能接口

ASP.NET Core 在 `Microsoft.AspNetCore.Http.Features` 中定义了许多 HTTP 功能接口，服务器使用这些接口来标识其支持的功能。以下功能接口处理请求并返回响应：

`IHttpRequestFeature` 定义 HTTP 请求的结构，包括协议、路径、查询字符串、标头和正文。

`IHttpResponseFeature` 定义 HTTP 响应的结构，包括状态代码、标头和响应的正文。

`IHttpAuthenticationFeature` 定义支持基于 `ClaimsPrincipal` 来标识用户并指定身份验证处理程序。

`IHttpUpgradeFeature` 定义对 [HTTP 升级](#) 的支持，允许客户端指定在服务器需要切换协议时要使用的其他协议。

`IHttpBufferingFeature` 定义禁用请求和/或响应缓冲的方法。

`IHttpConnectionFeature` 为本地和远程地址以及端口定义属性。

`IHttpRequestLifetimeFeature` 定义支持中止连接，或者检测是否已提前终止请求（如由于客户端断开连接）。

`IHttpSendFileFeature` 定义异步发送文件的方法。

`IHttpWebSocketFeature` 定义支持 Web 套接字的 API。

`IHttpRequestIdentifierFeature` 添加一个可以实现的属性来唯一标识请求。

`ISessionFeature` 为支持用户会话定义 `ISessionFactory` 和 `ISession` 抽象。

`ITlsConnectionFeature` 定义用于检索客户端证书的 API。

`ITlsTokenBindingFeature` 定义使用 TLS 令牌绑定参数的方法。

注意

`ISessionFeature` 不是服务器功能，而是由 `SessionMiddleware` 实现（请参阅[管理应用程序状态](#)）。

功能集合

`HttpContext` 的 `Features` 属性为获取和设置当前请求的可用 HTTP 功能提供了一个接口。由于功能集合即使在请求的上下文中也是可变的，所以可使用中间件来修改集合并添加对其他功能的支持。

中间件和请求功能

虽然服务器负责创建功能集合，但中间件既可以添加到该集合中，也可以使用集合中的功能。例如，`StaticFileMiddleware` 访问 `IHttpSendFileFeature` 功能。如果该功能存在，则用于从其物理路径发送所请求的静态文件。否则，使用较慢的替代方法来发送文件。如果可用，`IHttpSendFileFeature` 允许操作系统打开文件并执行直接内核模式复制到网卡。

另外，中间件可以添加到由服务器建立的功能集合中。中间件甚至可以取代现有的功能，以便增加服务器的功能。添加到集合中的功能稍后将在请求管道中立即用于其他中间件或基础应用程序本身。

通过结合自定义服务器实现和特定的中间件增强功能，可构造应用程序所需的精确功能集。这样一来，无需更改服务器即可添加缺少的功能，并确保只公开最少的功能，从而限制攻击外围应用并提高性能。

摘要

功能接口定义给定请求可能支持的特定 HTTP 功能。服务器定义功能的集合，以及该服务器支持的初始功能集，但中间件可用于增强这些功能。

其他资源

- [服务器](#)
- [中间件](#)
- [.NET 的开放 Web 接口 \(OWIN\)](#)

ASP.NET Core 中的基元

2018/1/31 • 1 min to read • [Edit Online](#)

ASP.NET Core 基元是由框架扩展所共享的低级别构建基块。你可以在自己的代码中使用这些构建基块。

[使用更改令牌检测更改](#)

使用 ASP.NET Core 中的更改令牌检测更改

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Luke Latham](#)

更改令牌是用于跟踪更改的通用、低级别构建基块。

[查看或下载示例代码\(如何下载\)](#)

IChangeToken 接口

`IChangeToken` 传播已发生更改的通知。`IChangeToken` 位于 `Microsoft.Extensions.Primitives` 命名空间中。对于不使用 `Microsoft.AspNetCore.All` 元包的应用，将在项目文件中引用 `Microsoft.Extensions.Primitives` NuGet 包。

`IChangeToken` 具有以下两个属性：

- `ActiveChangedCallbacks`, 指示令牌是否主动引发回调。如果将 `ActiveChangedCallbacks` 设置为 `false`，则不会调用回调，并且应用必须轮询 `HasChanged` 获取更改。如果未发生任何更改，或者丢弃或禁用基础更改侦听器，还可能永远不会取消令牌。
- `HasChanged`, 获取一个指示是否发生更改的值。

此接口具有一种方法，即 `RegisterChangeCallback(Action<Object>, Object)`，用于注册在令牌更改时调用的回调。调用回调之前，必须设置 `HasChanged`。

ChangeToken 类

`ChangeToken` 是静态类，用于传播已发生更改的通知。`ChangeToken` 位于 `Microsoft.Extensions.Primitives` 命名空间中。对于不使用 `Microsoft.AspNetCore.All` 元包的应用，将在项目文件中引用 `Microsoft.Extensions.Primitives` NuGet 包。

`ChangeToken` `OnChange(Func<IChangeToken>, Action)` 方法注册令牌更改时要调用的 `Action`：

- `Func<IChangeToken>` 生成令牌。
- 令牌更改时，调用 `Action`。

`ChangeToken` 具有 `OnChange<TState>(Func<IChangeToken>, Action<TState>, TState)` 重载，它接受传递到令牌使用者 `Action` 的附加 `TState` 参数。

`OnChange` 返回 `IDisposable`。调用 `Dispose` 将使令牌停止侦听更多更改并释放令牌的资源。

ASP.NET Core 中更改令牌的使用示例

更改令牌主要用于 ASP.NET Core 监视对象更改：

- 为了监视文件更改，`IFileProvider` 的 `Watch` 方法将为要监视的指定文件或文件夹创建 `IChangeToken`。
- 可以将 `IChangeToken` 令牌添加到缓存条目，以便在更改时触发缓存逐出。
- 对于 `IOptions` 更改，`IOptionsMonitor` 的默认 `OptionsMonitor` 实现具有重载，该重载接受一个或多个 `IOptionsChangeTokenSource` 实例。每个实例返回 `IChangeToken`，以注册用于跟踪选项更改的通知回调。

监视配置更改

默认情况下，ASP.NET Core 模板使用 `JSON 配置文件`(`appsettings.json`、`appsettings.Development.json` 和

`appsettings.Production.json`) 来加载应用配置设置。

使用接受 `reloadOnChange` 参数(ASP.NET Core 1.1 以及更高版本)的 `ConfigurationBuilder` 上的 `AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean)` 扩展方法配置这些文件。`reloadOnChange` 指示文件更改时是否应该重载配置。请参阅 [WebHost 便捷方法 CreateDefaultBuilder](#) 中的此设置：

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);
```

基于文件的配置由 `FileConfigurationSource` 表示。`FileConfigurationSource` 使用 `IFileProvider` 监视文件。

默认情况下，由使用 `FileSystemWatcher` 来监视配置文件更改的 `PhysicalFileProvider` 提供 `IFileMonitor`。

示例应用演示监视配置更改的两个实现。如果 `appsettings.json` 文件更改或者文件的 `Environment` 版本更改，则每个实现执行自定义代码。示例应用向控制台写入消息。

配置文件的 `FileSystemWatcher` 可以触发多个令牌回调，以用于单个配置文件更改。该示例的实现通过检查配置文件上的文件哈希来防范此问题。检查文件哈希可确保在运行自定义代码之前，至少已更改了其中一个配置文件。本示例使用 SHA1 文件哈希 (`Utilities/Utilities.cs`)：

```
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1.Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return new byte[20];
}
```

使用指数回退实现重试。进行重试是因为文件锁定可能暂时阻止对其中一个文件计算新的哈希。

简单启动更改令牌

将用于更改通知的令牌使用者 `Action` 回调注册到配置重载令牌 (`Startup.cs`)：

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

config.GetReloadToken() 提供令牌。回调是 InvokeChanged 方法：

```
private void InvokeChanged(IHostingEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

回调的 state 用于在 IHostingEnvironment 中传递。这有助于确定要监视的正确 appsettings 配置 JSON 文件，即 appsettings.<Environment>.json。只更改了一次配置文件时，文件哈希用于防止 WriteConsole 语句由于多个令牌回调而多次运行。

只要应用正在运行，该系统就会运行，并且用户不能禁用。

将配置更改作为服务进行监视

示例实现：

- 基本启动令牌监视。
- 作为服务监视。
- 启用和禁用监视的机制。

示例建立 IConfigurationMonitor 接口 (Extensions/ConfigurationMonitor.cs)：

```
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

已实现的类的构造函数 ConfigurationMonitor 注册用于更改通知的回调：

```
public ConfigurationMonitor(IConfiguration config, IHostingEnvironment env)
{
    _env = env;

    ChangeToken.OnChange< IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` 提供令牌。 `InvokeChanged` 是回调方法。此实例中的 `state` 是描述监视状态的字符串。

使用了以下两个属性：

- `MonitoringEnabled`，指示回调是否应该运行其自定义代码。
- `CurrentState`，描述在 UI 中使用的当前监视状态。

`InvokeChanged` 方法类似于前面的方法，不同之处在于：

- 不运行其代码，除非 `MonitoringEnabled` 为 `true`。
- 将 `CurrentState` 属性字符串设置为记录代码运行时间的描述性消息。
- 注意其 `WriteConsole` 输出中的当前 `state`。

```
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";

            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole($"Configuration changed (ConfigurationMonitor Class) {message}, state:
{state.CurrentState}");
        }
    }
}
```

将实例 `ConfigurationMonitor` 作为服务注册在 `Startup.cs` 的 `ConfigureServices` 中：

```
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

索引页提供对配置监视的用户控制。将 `IConfigurationMonitor` 的实例注入到 `IndexModel` 中：

```
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

按钮启用和禁用监视：

```
<button class="btn btn-danger" asp-page-handler="StopMonitoring">Stop Monitoring</button>
```

```
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = string.Empty;

    return RedirectToAction();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToAction();
}
```

触发 `OnPostStartMonitoring` 时，会启用监视并清除当前状态。触发 `OnPostStopMonitoring` 时，会禁用监视并设置状态以反映未进行监视。

监视缓存文件更改

可以使用 `IMemoryCache` 将文件内容缓存在内存中。[内存中缓存](#) 主题中介绍了在内存中缓存。无需采取其他步骤（如下面所述的实现），如果源数据更改，将从缓存返回陈旧（过时）数据。

当续订可调过期时段造成陈旧缓存数据时，不考虑所缓存源文件的状态。数据的每个请求续订可调过期时段，但不会将文件重载到缓存中。任何使用文件缓存内容的应用功能都可能会收到陈旧的内容。

在文件缓存方案中使用更改令牌可防止缓存中出现陈旧的文件内容。示例应用演示方法的实现。

示例使用 `GetFileContent` 来完成以下操作：

- 返回文件内容。
- 实现具有指数回退的重试算法，以涵盖文件锁暂时阻止读取文件的情况。

Utilities/Utilities.cs：

```

public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return null;
}

```

创建 `FileService` 以处理缓存文件查找。服务的 `GetFileContent` 方法调用尝试从内存中缓存获取文件内容并将其返回到调用方 (`Services/FileService.cs`)。

如果使用缓存键未找到缓存的内容，则将执行以下操作：

1. 使用 `GetFileContent` 获取文件内容。
2. 使用 `IFileProviders.Watch` 从文件提供程序获取更改令牌。修改该文件时，会触发令牌的回调。
3. 可调过期时段将缓存文件内容。如果缓存文件时发生了更改，则将更改令牌与 `MemoryCacheEntryExtensions.AddExpirationToken` 连接在一起，以逐出缓存条目。

```

public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IHostingEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        // For the purposes of this example, files are stored
        // in the content root of the app. To obtain the physical
        // path to a file at the content root, use the
        // ContentRootFileProvider on IHostingEnvironment.
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);
        }
    }

    return string.Empty;
}
}

```

将 `FileService` 和内存缓存服务 (`Startup.cs`) 一起注册在服务容器中：

```

services.AddMemoryCache();
services.AddSingleton<FileService>();

```

页面模型使用服务 (`Pages/Index.cshtml.cs`) 加载文件内容：

```

var fileContent = await _fileService.GetFileContents("poem.txt");

```

CompositeChangeToken 类

要在单个对象中表示一个或多个 `IChangeToken` 实例, 请使用 [CompositeChangeToken](#) 类。

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
    {
        firstCancellationChangeToken,
        secondCancellationChangeToken
    });
}
```

如果任何表示的令牌 `HasChanged` 为 `true`, 则复合令牌上的 `HasChanged` 报告 `true`。如果任何表示的令牌 `ActiveChangeCallbacks` 为 `true`, 则复合令牌上的 `ActiveChangeCallbacks` 报告 `true`。如果发生多个并发更改事件, 则调用一次复合更改回调。

请参阅

- [内存中缓存](#)
- [使用分布式缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存](#)
- [响应缓存中间件](#)
- [缓存标记帮助程序](#)
- [分布式缓存标记帮助程序](#)

ASP.NET Core 中 .NET 的开放 Web 接口 (OWIN)

2018/5/14 • 7 min to read • [Edit Online](#)

作者:Steve Smith 和 Rick Anderson

ASP.NET Core 支持 .NET 的开放 Web 接口 (OWIN)。OWIN 允许 Web 应用从 Web 服务器分离。它定义了在管道中使用中间件来处理请求和相关响应的标准方法。ASP.NET Core 应用程序和中间件可以与基于 OWIN 的应用程序、服务器和中间件进行互操作。

OWIN 提供了一个分离层，可一起使用具有不同对象模型的两个框架。[Microsoft.AspNetCore.Owin](#) 包提供了两个适配器实现：

- ASP.NET Core 到 OWIN
- OWIN 到 ASP.NET Core

此方法可将 ASP.NET Core 托管在兼容 OWIN 的服务器/主机上，或在 ASP.NET Core 上运行其他兼容 OWIN 的组件。

注意：使用这些适配器会带来性能成本。仅使用 ASP.NET Core 组件的应用程序不应使用 Owin 包或适配器。

[查看或下载示例代码\(如何下载\)](#)

在 ASP.NET 管道中运行 OWIN 中间件

ASP.NET Core 的 OWIN 支持作为 [Microsoft.AspNetCore.Owin](#) 包的一部分进行部署。可通过安装此包将 OWIN 支持导入到项目中。

OWIN 中间件符合 [OWIN 规范](#)，该规范要求使用 [Func<IDictionary<string, object>, Task>](#) 接口，并设置特定的键（如 `owin.ResponseBody`）。以下简单的 OWIN 中间件显示“Hello World”：

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: http://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string, string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
        responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

示例签名返回 `Task`，并接受 OWIN 所要求的 `IDictionary<string, object>`。

以下代码显示了如何使用 `UseOwin` 扩展方法将 `OwinHello` 中间件（如上所示）添加到 ASP.NET 管道。

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

可配置在 OWIN 管道中要进行的其他操作。

注意

响应标头只能在首次写入响应流之前进行修改。

注意

由于性能原因，不建议多次调用 `UseOwin`。组合在一起时 OWIN 组件的性能最佳。

```
app.UseOwin(pipeline =>
{
    pipeline(async (next) =>
    {
        // do something before
        await OwinHello(new OwinEnvironment(HttpContext));
        // do something after
    });
});
```

在基于 OWIN 的服务器中使用 ASP.NET 托管

基于 OWIN 的服务器可托管 ASP.NET 应用程序。[Nowin](#)(.NET OWIN Web 服务器)属于这种服务器。本文的示例中包含了一个引用 Nowin 的项目，并使用它创建了一个自托管 ASP.NET Core 的 `IIServer`。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

`I Server` 是需要 `Features` 属性和 `Start` 方法的接口。

`Start` 负责配置和启动服务器，在此情况下，此操作通过一系列 Fluent API 调用完成，这些调用设置从 `I Server Addresses Feature` 分析的地址。请注意，`_builder` 变量的 Fluent 配置指定请求将由方法中之前定义的 `appFunc` 来处理。对于每个请求，都会调用此 `Func` 以处理传入请求。

我们还将添加一个 `IWebHostBuilder` 扩展，以便添加和配置 Nowin 服务器。

```
using System;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.Extensions.DependencyInjection;
using Nowin;
using NowinSample;

namespace Microsoft.AspNetCore.Hosting
{
    public static class NowinWebHostBuilderExtensions
    {
        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder)
        {
            return builder.ConfigureServices(services =>
            {
                services.AddSingleton<IServer, NowinServer>();
            });
        }

        public static IWebHostBuilder UseNowin(this IWebHostBuilder builder, Action<ServerBuilder> configure)
        {
            builder.ConfigureServices(services =>
            {
                services.Configure(configure);
            });
            return builder.UseNowin();
        }
    }
}
```

完成此操作后，调用 `Program.cs` 中的扩展以使用此自定义服务器运行 ASP.NET Core 应用：

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace NowinSample
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseNowin()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

了解有关 ASP.NET 服务器的更多信息。

在基于 OWIN 的服务器上运行 ASP.NET Core 并使用其 WebSocket 支持

ASP.NET Core 如何利用基于 OWIN 的服务器功能的另一个示例是访问 WebSocket 等功能。前面示例中使用的 .NET OWIN Web 服务器支持内置的 Web 套接字，可由 ASP.NET Core 应用程序利用。下面的示例显示了简单的 Web 应用，它支持 Web 套接字并回显通过 WebSocket 发送到服务器的所有内容。

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            if (context.WebSockets.IsWebSocketRequest)
            {
                WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
                await EchoWebSocket(webSocket);
            }
            else
            {
                await next();
            }
        });
    }

    app.Run(context =>
    {
        return context.Response.WriteAsync("Hello World");
    });
}

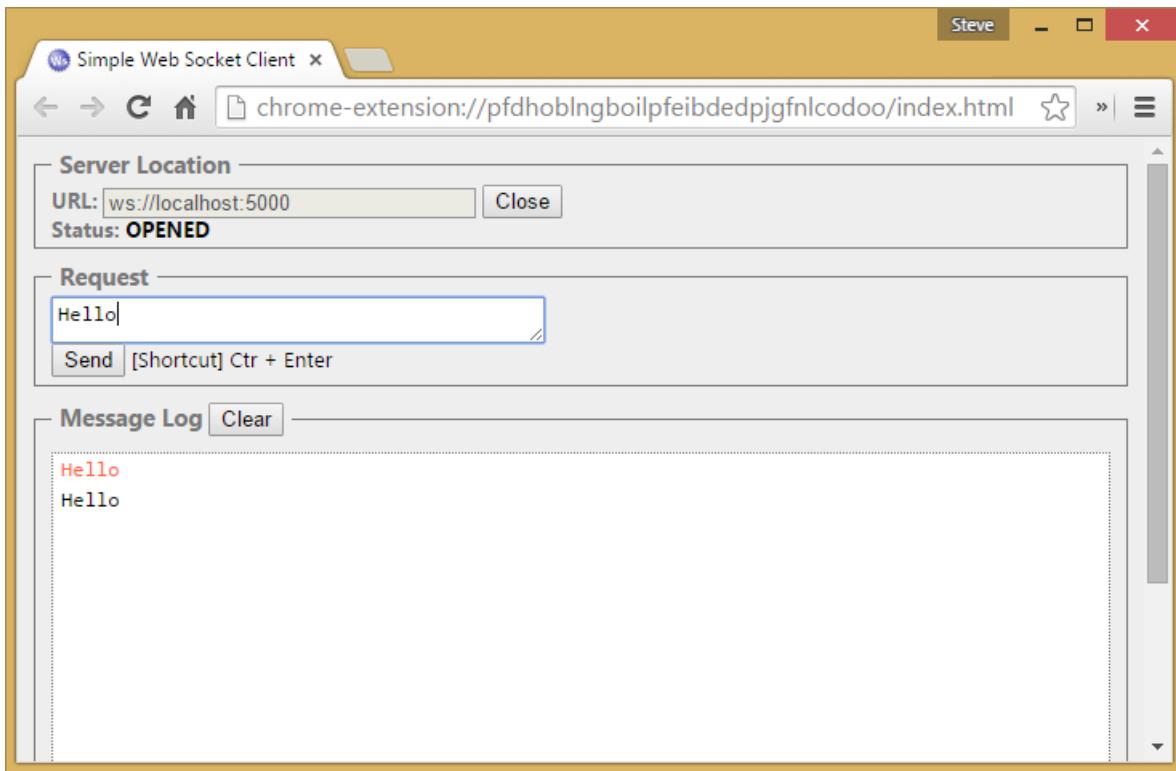
private async Task EchoWebSocket(WebSocket webSocket)
{
    byte[] buffer = new byte[1024];
    WebSocketReceiveResult received = await webSocket.ReceiveAsync(
        new ArraySegment<byte>(buffer), CancellationToken.None);

    while (!webSocket.CloseStatus.HasValue)
    {
        // Echo anything we receive
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
            received.MessageType, received.EndOfMessage, CancellationToken.None);

        received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
            CancellationToken.None);
    }

    await webSocket.CloseAsync(webSocket.CloseStatus.Value,
        webSocket.CloseStatusDescription, CancellationToken.None);
}
}
```

使用与前一个相同的 `NowinServer` 来配置此[示例](#) - 唯一的区别是如何在其 `Configure` 方法中配置应用程序。使用[简单的 websocket 客户端](#)的测试演示应用程序：



OWIN 环境

可使用 `HttpContext` 来构造 OWIN 环境。

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN 键

OWIN 依赖于 `IDictionary<string,object>` 对象，以在整个 HTTP 请求/响应交换中传达信息。ASP.NET Core 实现以下所列的键。请参阅[主规范、扩展和 OWIN Key Guidelines and Common Keys](#)(OWIN 键指南和常用键)。

请求数据 (OWIN v1.0.0)

键	值(类型)	描述
<code>owin.RequestScheme</code>	<code>String</code>	
<code>owin.RequestMethod</code>	<code>String</code>	
<code>owin.RequestPathBase</code>	<code>String</code>	
<code>owin.RequestPath</code>	<code>String</code>	
<code>owin.RequestQueryString</code>	<code>String</code>	
<code>owin.RequestProtocol</code>	<code>String</code>	
<code>owin.RequestHeaders</code>	<code>IDictionary<string, string[]></code>	

键	值(类型)	描述
owin.RequestBody	Stream	

请求数据 (OWIN v1.1.0)

键	值(类型)	描述
owin.RequestId	String	Optional

响应数据 (OWIN v1.0.0)

键	值(类型)	描述
owin.ResponseStatusCode	int	Optional
owin.ResponseReasonPhrase	String	Optional
owin.ResponseHeaders	IDictionary<string, string[]>	
owin.ResponseBody	Stream	

其他数据 (OWIN v1.0.0)

键	值(类型)	描述
owin.CallCancelled	CancellationToken	
owin.Version	String	

常用键

键	值(类型)	描述
ssl.ClientCertificate	X509Certificate	
ssl.LoadClientCertAsync	Func<Task>	
server.RemoteIpAddress	String	
server.RemotePort	String	
server.LocalIpAddress	String	
server.LocalPort	String	
server.IsLocal	bool	
server.OnSendingHeaders	Action<Action<object>, object>	

SendFiles v0.3.0

键	值(类型)	描述
sendfile.SendAsync	请参阅 委托签名	每请求

Opaque v0.3.0

键	值(类型)	描述
opaque.Version	<code>String</code>	
opaque.Upgrade	<code>OpaqueUpgrade</code>	请参阅 委托签名
opaque.Stream	<code>Stream</code>	
opaque.CallCancelled	<code>CancellationToken</code>	

WebSocket v0.3.0

键	值(类型)	描述
websocket.Version	<code>String</code>	
websocket.Accept	<code>WebSocketAccept</code>	请参阅 委托签名
websocket.AcceptAlt		非规范
websocket.SubProtocol	<code>String</code>	请参阅 RFC6455 4.2.2 节 步骤 5.5
websocket.SendAsync	<code>WebSocketSendAsync</code>	请参阅 委托签名
websocket.ReceiveAsync	<code>WebSocketReceiveAsync</code>	请参阅 委托签名
websocket.CloseAsync	<code>WebSocketCloseAsync</code>	请参阅 委托签名
websocket.CallCancelled	<code>CancellationToken</code>	
websocket.ClientCloseStatus	<code>int</code>	Optional
websocket.ClientCloseDescription	<code>String</code>	Optional

其他资源

- [中间件](#)
- [服务器](#)

ASP.NET Core 中的 WebSocket 支持

2018/5/14 • 6 min to read • [Edit Online](#)

作者: Tom Dykstra 和 Andrew Stanton-Nurse

本文介绍 ASP.NET Core 中 WebSocket 的入门方法。WebSocket (RFC 6455) 是一个协议，支持通过 TCP 连接建立持久的双向信道。它用于从快速实时通信中获益的应用，如聊天、仪表板和游戏应用。

[查看或下载示例代码\(如何下载\)](#)。有关详细信息，请参阅[后续步骤部分](#)。

系统必备

- ASP.NET Core 1.1 或更高版本
- 支持 ASP.NET Core 的任何操作系统：
 - Windows 7/Windows Server 2008 或更高版本
 - Linux
 - macOS
- 如果应用在安装了 IIS 的 Windows 上运行：
 - Windows 8 / Windows Server 2012 及更高版本
 - IIS 8 / IIS 8 Express
 - 必须在 IIS 中启用 WebSocket(请参阅 [IIS/IIS Express 支持部分](#)。)
- 如果应用在 [HTTP.sys](#) 上运行：
 - Windows 8 / Windows Server 2012 及更高版本
- 有关支持的浏览器，请参阅 <https://caniuse.com/#feat=websockets>。

何时使用 WebSocket

通过 WebSocket 可直接使用套接字连接。例如，使用 WebSocket 可以让实时游戏达到最佳性能。

[ASP.NET SignalR](#) 为实时功能提供了更丰富的应用模型，但它仅在 ASP.NET 4.x 上运行，不能在 ASP.NET Core 上运行。SignalR 的 ASP.NET Core 版本计划随 ASP.NET Core 2.1 一起发布。请参阅 [ASP.NET Core 2.1 高级计划](#)。

发布 SignalR Core 前，可使用 WebSocket。但 SignalR 的功能必须由开发人员提供和支持。例如：

- 通过自动回退到替代传输方法来支持更广泛的浏览器版本。
- 连接断开时自动重新连接。
- 支持服务器上的客户端调用方法，反之亦然。
- 支持缩放到多个服务器。

使用方法

- 安装 [Microsoft.AspNetCore.WebSockets](#) 包。
- 配置中间件。
- 接受 WebSocket 请求。
- 发送和接收消息。

配置中间件

在 `Startup` 类的 `Configure` 方法中添加 WebSocket 中间件：

```
app.UseWebSockets();
```

可配置以下设置：

- `KeepAliveInterval` - 向客户端发送“ping”帧的频率，以确保代理保持连接处于打开状态。
- `ReceiveBufferSize` - 用于接收数据的缓冲区的大小。高级用户可能需要对其进行更改，以便根据数据大小调整性能。

```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
app.UseWebSockets(webSocketOptions);
```

接受 WebSocket 请求

在请求生命周期后期（例如在 `Configure` 方法或 MVC 操作的后期），检查它是否是 WebSocket 请求并接受 WebSocket 请求。

以下示例来自 `Configure` 方法的后期：

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
            await Echo(context, webSocket);
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }
});
```

WebSocket 请求可以来自任何 URL，但此示例代码只接受 `/ws` 的请求。

发送和接收消息

`AcceptWebSocketAsync` 方法将 TCP 连接升级到 WebSocket 连接，并提供 `WebSocket` 对象。使用 `WebSocket` 对象发送和接收消息。

之前显示的接受 WebSocket 请求的代码将 `WebSocket` 对象传递给 `Echo` 方法。代码接收消息并立即发回相同的消息。循环发送和接收消息，直到客户端关闭连接：

```

private async Task Echo(HttpContext context, WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType,
result.EndOfMessage, CancellationToken.None);

        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }
    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription,
CancellationToken.None);
}

```

如果在开始循环之前接受 WebSocket 连接，中间件管道会结束。关闭套接字后，管道展开。即接受 WebSocket 时，请求停止在管道中推进。循环结束且套接字关闭时，请求继续回到管道。

IIS/IIS Express 支持

安装了 IIS/IIS Express 8 或更高版本的 Windows Server 2012 或更高版本以及 Windows 8 或更高版本支持 WebSocket 协议。

在 Windows Server 2012 或更高版本上启用对 WebSocket 协议的支持：

1. 通过“管理”菜单或“服务器管理器”中的链接使用“添加角色和功能”向导。
2. 选择“基于角色或基于功能的安装”。选择“下一步”。
3. 选择适当的服务器(默认情况下选择本地服务器)。选择“下一步”。
4. 在“角色”树中展开“Web 服务器(IIS)”、然后依次展开“Web 服务器”和“应用程序开发”。
5. 选择“WebSocket 协议”。选择“下一步”。
6. 如果无需其他功能，请选择“下一步”。
7. 选择“安装”。
8. 安装完成后，选择“关闭”以退出向导。

在 Windows 8 或更高版本上启用对 WebSocket 协议的支持：

1. 导航到“控制面板”>“程序”>“程序和功能”>“打开或关闭 Windows 功能”(位于屏幕左侧)。
2. 打开以下节点：“Internet Information Services”>“万维网服务”>“应用程序开发功能”。
3. 选择“WebSocket 协议”功能。选择“确定”。

在 node.js 上使用 socket.io 时禁用 WebSocket

如果在 Node.js 的 `socket.io` 中使用 WebSocket 支持，请使用 `web.config` 或 `applicationHost.config` 中的 `<webSocket>` 元素禁用默认的 IIS WebSocket 模块。如果不执行此步骤，IIS WebSocket 模块将尝试处理 WebSocket 通信而不是 Node.js 和应用。

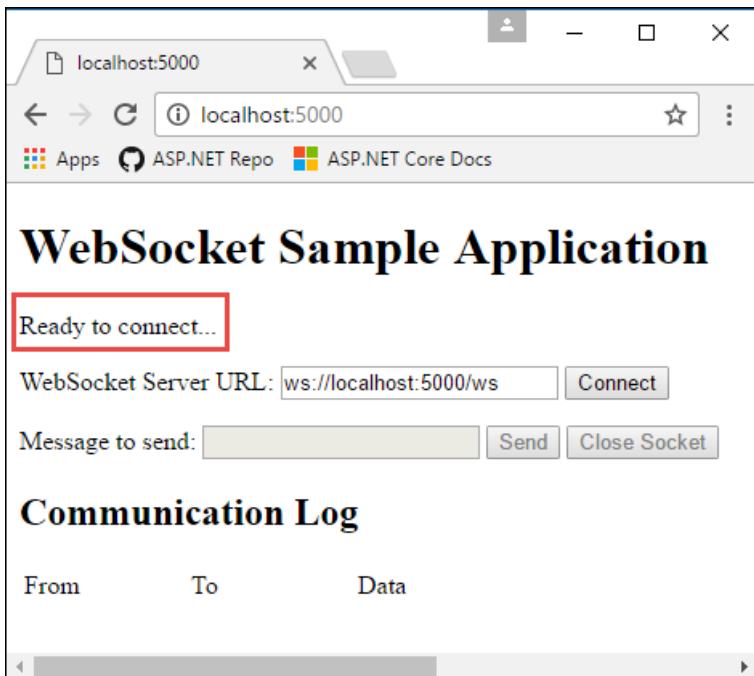
```

<system.webServer>
  <webSocket enabled="false" />
</system.webServer>

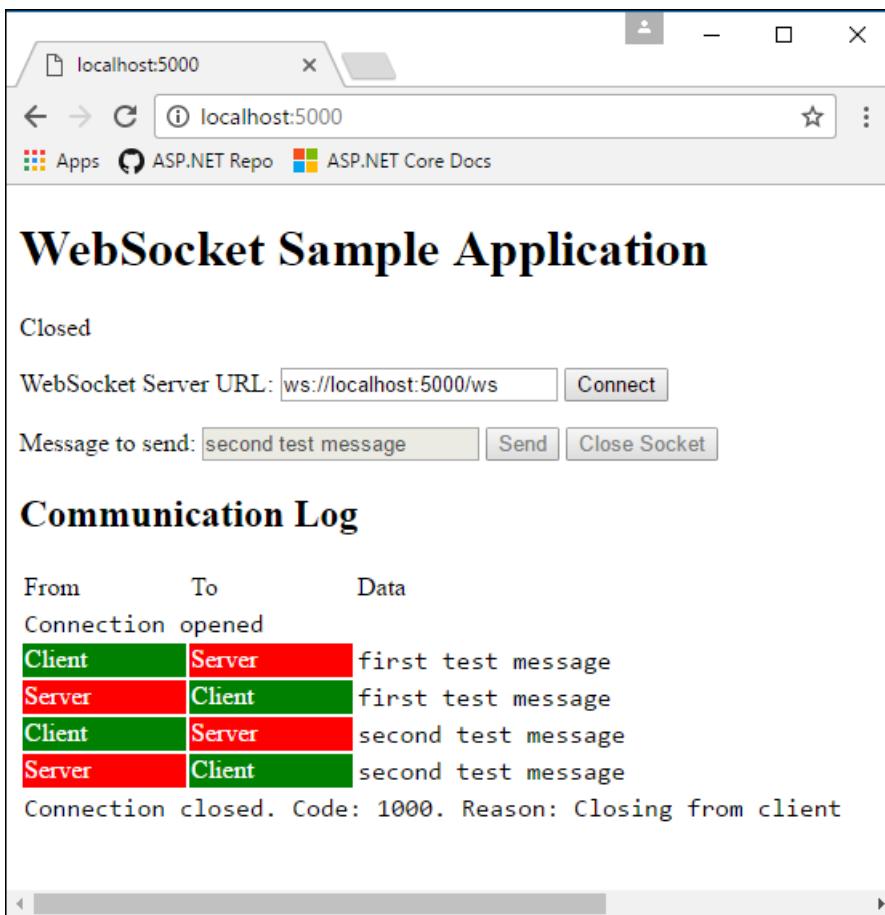
```

后续步骤

本文附带的[示例应用](#)是一个 echo 应用。它有一个可建立 WebSocket 连接的网页，且服务器将其收到的消息重新发回到客户端。从命令提示符运行该应用(它未设置为在安装了 IIS Express 的 Visual Studio 中运行)并导航到 <http://localhost:5000>。该网页的左上方显示连接状态：



选择“连接”，向显示的 URL 发送 WebSocket 请求。输入测试消息并选择“发送”。完成后，请选择“关闭套接字”。“通信日志”部分会报告每一个发生的“打开”、“发送”和“关闭”操作。



ASP.NET Core 2.x 的 Microsoft.AspNetCore.All 元包

2018/5/4 • 1 min to read • [Edit Online](#)

此功能需要面向 .NET Core 2.x 的 ASP.NET Core 2.x。

ASP.NET Core 的 [Microsoft.AspNetCore.All](#) 元包包括：

- ASP.NET Core 团队支持的所有包。
- Entity Framework Core 支持的所有包。
- ASP.NET Core 和 Entity Framework Core 使用的内部和第三方依赖关系。

[Microsoft.AspNetCore.All](#) 包中包含了 ASP.NET Core 2.x 和 Entity Framework Core 2.x 的所有功能。定目标到 ASP.NET Core 2.0 的默认项目模板使用此包。

[Microsoft.AspNetCore.All](#) 元包的版本号表示 ASP.NET Core 版本和 Entity Framework Core 版本(与 .NET Core 版本对齐)。

使用 [Microsoft.AspNetCore.All](#) 元包的应用程序会自动使用 [.NET Core 运行时存储](#)。此运行时存储包含运行 ASP.NET Core 2.x 应用程序所需的所有运行时资产。使用 [Microsoft.AspNetCore.All](#) 元包时，应用程序不会部署引用的 ASP.NET Core NuGet 包中的任何资产—.NET Core 运行时存储包含这些资产。运行时存储中的资产已经过预编译，以便缩短应用程序启动时间。

可使用包修整过程来删除不使用的包。已发布的应用程序输出中排除了修整的包。

以下 .csproj 文件引用了 ASP.NET Core 的 [Microsoft.AspNetCore.All](#) 元包：

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

</Project>
```

在 ASP.NET 和 ASP.NET Core 之间进行选择

2018/5/14 • 1 min to read • [Edit Online](#)

不论创建怎样的 Web 应用, ASP.NET 都可提供解决方案:从面向 Windows Server 的企业 Web 应用到面向 Linux 容器的小微服务, 以及中间的所有应用。

ASP.NET Core

ASP.NET Core 是一个跨平台的开源框架, 用于在 Windows、macOS 或 Linux 上生成基于云的新式 Web 应用。

ASP.NET

ASP.NET 是一个成熟的框架, 提供在 Windows 上生成基于服务器的企业级 Web 应用所需的所有服务。

哪一个适合我?

ASP.NET CORE	ASP.NET
针对 Windows、macOS 或 Linux 进行生成	针对 Windows 进行生成
Razor 页面 是在 ASP.NET Core 2.x 及更高版本中创建 Web UI 时建议使用的方法。另请参阅 MVC 、 Web API 和 SignalR 。	使用 Web 窗体 、 SignalR 、 MVC 、 Web API 或 网页
每个计算机多个版本	每个计算机一个版本
使用 C# 或 F# 通过 Visual Studio 、 Visual Studio for Mac 或 Visual Studio Code 进行开发	使用 C#、VB 或 F# 通过 Visual Studio 进行开发
比 ASP.NET 性能更高	良好的性能
选择 .NET Framework 或 .NET Core 运行时	使用 .NET Framework 运行时

ASP.NET Core 方案

- [Razor 页面](#) 是在 ASP.NET Core 2.x 及更高版本中创建 Web UI 时建议使用的方法。
- [网站](#)
- [API](#)
- [实时](#)

ASP.NET 方案

- [网站](#)
- [API](#)
- [实时](#)

资源

- [ASP.NET 简介](#)
- [ASP.NET Core 简介](#)

ASP.NET Core 中的 Razor 页面介绍

2018/5/17 • 22 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Ryan Nowak](#)

Razor 页面是 ASP.NET Core MVC 的一个新特性, 它可以使基于页面的编码方式更简单高效。

若要查找使用模型视图控制器方法的教程, 请参阅 [ASP.NET Core MVC 入门](#)。

本文档介绍 Razor 页面。它并不是分步教程。如果认为某些部分过于复杂, 请参阅 [Razor 页面入门](#)。有关 ASP.NET Core 的概述, 请参阅 [ASP.NET Core 简介](#)。

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

创建 Razor 页面项目

- [Visual Studio](#)
- [Visual Studio for Mac](#)
- [Visual Studio Code](#)
- [.NET Core CLI](#)

请参阅 [Razor 页面入门](#), 获取关于如何使用 Visual Studio 创建 Razor 页面项目的详细说明。

Razor 页面

Startup.cs 中已启用 Razor 页面:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

请考虑一个基本页面:

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

上述代码看上去类似于一个 Razor 视图文件。不同之处在于 `@page` 指令。`@page` 使文件转换为一个 MVC 操作，这意味着它将直接处理请求，而无需通过控制器处理。`@page` 必须是页面上的第一个 Razor 指令。`@page` 将影响其他 Razor 构造的行为。

将在以下两个文件中显示使用 `PageModel` 类的类似页面。Pages/Index2.cshtml 文件：

```
@page
@using RazorPagesIntro.Pages
@model IndexModel2

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

Pages/Index2.cshtml.cs 页面模型：

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System;

namespace RazorPagesIntro.Pages
{
    public class IndexModel2 : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

按照惯例，`PageModel` 类文件的名称与追加 `.cs` 的 Razor 页面文件名称相同。例如，前面的 Razor 页面的名称为 Pages/Index2.cshtml。包含 `PageModel` 类的文件的名称为 Pages/Index2.cshtml.cs。

页面的 URL 路径的关联由页面在文件系统中的位置决定。下表显示了 Razor 页面路径及匹配的 URL：

文件名和路径	匹配的 URL
/Pages/Index.cshtml	/ 或 /Index
/Pages/Contact.cshtml	/Contact
/Pages/Store/Contact.cshtml	/Store/Contact
/Pages/Store/Index.cshtml	/Store 或 /Store/Index

注意：

- 默认情况下，运行时在“Pages”文件夹中查找 Razor 页面文件。
- URL 未包含页面时，`Index` 为默认页面。

编写基本窗体

由于 Razor 页面的设计，在构建应用时可轻松实施用于 Web 浏览器的常用模式。[模型绑定、标记帮助程序](#) 和 [HTML 帮助程序](#) 均只可用于 Razor 页面类中定义的属性。请参考为 [Contact](#) 模型实现基本的“联系我们”窗体的页面：

在本文档中的示例中，[DbContext](#) 在 [Startup.cs](#) 文件中进行初始化。

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

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

数据模型：

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

数据库上下文：

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

Pages/Create.cshtml 视图文件：

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

Pages/Create.cshtml.cs 页面模型：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly ApplicationDbContext _db;

        public CreateModel(ApplicationDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}

```

按照惯例，`PageModel` 类命名为 `<PageName>Model` 并且它与页面位于同一个命名空间中。

使用 `PageModel` 类，可以将页面的逻辑与其展示分离开来。它定义了页面处理程序，用于处理发送到页面的请求和用于呈现页面的数据。借助这种分离，可以通过[依赖关系注入](#)管理页面依赖关系，并对页面执行[单元测试](#)。

页面包含 `OnPostAsync` 处理程序方法，它在 `POST` 请求上运行（当用户发布窗体时）。可以为任何 HTTP 谓词添加处理程序方法。最常见的处理程序是：

- `OnGet`，用于初始化页面所需的状态。[OnGet 示例](#)。
- `OnPost`，用于处理窗体提交。

`Async` 命名后缀为可选，但是按照惯例通常会将它用于异步函数。前面示例中的 `OnPostAsync` 代码看上去与通常在控制器中编写的内容相似。前面的代码通常用于 Razor 页面。多数 MVC 基元（例如[模型绑定](#)、[验证](#)和[操作结果](#)）都是共享的。

之前的 `OnPostAsync` 方法：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

OnPostAsync 的基本流：

检查验证错误。

- 如果没有错误，则保存数据并重定向。
- 如果有错误，则再次显示页面并附带验证消息。客户端验证与传统的 ASP.NET Core MVC 应用程序相同。很多情况下，都会在客户端上检测到验证错误，并且从不将它们提交到服务器。

成功输入数据后，`OnPostAsync` 处理程序方法调用 `RedirectToPage` 帮助程序方法来返回 `RedirectResult` 的实例。`RedirectToPage` 是新的操作结果，类似于 `RedirectToAction` 或 `RedirectToRoute`，但是已针对页面进行自定义。在前面的示例中，它将重定向到根索引页（/Index）。[页面 URL 生成](#)部分中详细介绍了 `RedirectToPage`。

提交的窗体存在（已传递到服务器的）验证错误时，`OnPostAsync` 处理程序方法调用 `Page` 帮助程序方法。`Page` 返回 `PageResult` 的实例。返回 `Page` 的过程与控制器中的操作返回 `View` 的过程相似。`PageResult` 是处理程序方法的默认返回类型。返回 `void` 的处理程序方法将显示页面。

`Customer` 属性使用 `[BindProperty]` 特性来选择加入模型绑定。

```
public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}
```

默认情况下，Razor 页面只绑定带有非 GET 谓词的属性。绑定属性可以减少需要编写的代码量。绑定通过使用相同的属性显示窗体字段（`<input asp-for="Customer.Name" />`）来减少代码，并接受输入。

注意

出于安全原因，必须选择绑定 GET 请求数据以对模型属性进行分页。请在将用户输入映射到属性前对其进行验证。当处理依赖查询字符串或路由值的方案时，选择加入此行为非常有用。

若要将属性绑定在 GET 请求上，请将 `[BindProperty]` 特性的 `SupportsGet` 属性设置为 `true`：
`[BindProperty(SupportsGet = true)]`

主页（Index.cshtml）：

```
@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customers)
            {
                <tr>
                    <td>@contact.Id</td>
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page=".Edit" asp-route-id="@contact.Id">edit</a>
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>

    <a asp-page=".Create">Create</a>
</form>
```

Index.cshtml.cs 隱藏文件：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}

```

Index.cshtml 文件包含以下标记来创建每个联系人项的编辑链接：

```
<a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
```

[定位点标记帮助程序](#) 使用 `asp-route-{value}` 属性生成“编辑”页面的链接。此链接包含路由数据及联系人 ID。例如 `http://localhost:5000/Edit/1`。

Pages/Edit.cshtml 文件：

```
@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

 @{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

第一行包含 `@page "{id:int}"` 指令。路由约束 `"{id:int}"` 告诉页面接受包含 `int` 路由数据的页面请求。如果页面请求未包含可转换为 `int` 的路由数据，则运行时返回 HTTP 404(未找到)错误。若要使 ID 可选，请将 `?` 追加到路由约束：

```
@page "{id:int?}"
```

Pages/Edit.cshtml.cs 文件：

```

using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}

```

Index.cshtml 文件还包含用于为每个客户联系人创建删除按钮的标记：

```

<button type="submit" asp-page-handler="delete"
asp-route-id="@contact.Id">delete</button>

```

删除按钮采用 HTML 呈现，其 `formaction` 包括参数：

- `asp-route-id` 属性指定的客户联系人 ID。

- `asp-page-handler` 属性指定的 `handler`。

下面是呈现的删除按钮的示例，其中客户联系人 ID 为 `1`：

```
<button type="submit" formaction="/?id=1&handler=delete">Delete</button>
```

选中按钮时，向服务器发送窗体 `POST` 请求。按照惯例，根据方案 `OnPost[handler]Async` 基于 `handler` 参数的值来选择处理程序方法的名称。

因为本示例中 `handler` 是 `delete`，因此 `OnPostDeleteAsync` 处理程序方法用于处理 `POST` 请求。如果 `asp-page-handler` 设置为不同值（如 `remove`），则选择名称为 `OnPostRemoveAsync` 的页面处理程序方法。

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

OnPostDeleteAsync 方法：

- 接受来自查询字符串的 `id`。
- 使用 `FindAsync` 查询客户联系人的数据库。
- 如果找到客户联系人，则从客户联系人列表将其删除。数据库将更新。
- 调用 `RedirectToPage`，重定向到根索引页（`/Index`）。

标记所需的页属性

`PageModel` 上的属性可通过 `Required` 特性进行修饰：

[!code-cs]

有关详细信息，请参阅[模型验证](#)。

使用 OnGet 处理程序管理 HEAD 请求

通常，针对 HEAD 请求创建和调用 HEAD 处理程序：

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("HandledBy", "Handled by OnHead!");
}
```

如果未定义 HEAD 处理程序（`OnHead`），Razor 页面会回退以调用 ASP.NET Core 2.1 或更高版本中的 GET 页处理程序（`OnGet`）。使用 ASP.NET Core 2.1 到 2.x 版本 `Startup.Configure` 中的 `SetCompatibilityVersion` 方法，选择加入此行为：

```
services.AddMvc()
    .SetCompatibilityVersion(Microsoft.AspNetCore.Mvc.CompatibilityVersion.Version_2_1);
```

`SetCompatibilityVersion` 有效地将 Razor 页面选项 `AllowMappingHeadRequestsToGetHandler` 设置为 `true`。

可以显式地选择使用特定行为，而不是通过 `SetCompatibilityVersion` 选择使用所有 2.1 行为。以下代码选择使用将 HEAD 映射到 GET 处理程序这一行为。

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.AllowMappingHeadRequestsToGetHandler = true;
});
```

XSRF/CSRF 和 Razor 页面

无需为[防伪验证](#)编写任何代码。Razor 页面自动将防伪标记生成过程和验证过程包含在内。

将布局、分区、模板和标记帮助程序用于 Razor 页面

页面可使用 Razor 视图引擎的所有功能。布局、分区、模板、标记帮助程序、`_ViewStart.cshtml` 和 `_ViewImports.cshtml` 的工作方式与它们在传统的 Razor 视图中的工作方式相同。

让我们使用其中的一些功能来整理此页面。

向 Pages/_Layout.cshtml 添加[布局页面](#)：

```
<!DOCTYPE html>
<html>
<head>
    <title>Razor Pages Sample</title>
</head>
<body>
    <a asp-page="/Index">Home</a>
    @RenderBody()
    <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>
```

布局：

- 控制每个页面的布局(页面选择退出布局时除外)。
- 导入 HTML 结构, 例如 JavaScript 和样式表。

请参阅[布局页面](#)了解详细信息。

在 Pages/_ViewStart.cshtml 中设置 `Layout` 属性：

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

布局位于“页面”文件夹中。页面按层次结构从当前页面的文件夹开始查找其他视图(布局、模板、分区)。可以从“页面”文件夹下的任意 Razor 页面使用“页面”文件夹中的布局。

建议不要将布局文件放在“视图/共享”文件夹中。视图/共享 是一种 MVC 视图模式。Razor 页面旨在依赖文件夹层次结构, 而非路径约定。

Razor 页面中的视图搜索包含“页面”文件夹。用于 MVC 控制器和传统 Razor 视图的布局、模板和分区可直接工作。

添加 Pages/_ViewImports.cshtml 文件：

```
@namespace RazorPagesContacts.Pages  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

本教程的后续部分中将介绍 `@namespace`。`@addTagHelper` 指令将[内置标记帮助程序](#)引入“页面”文件夹中的所有页面。

页面上显式使用 `@namespace` 指令后：

```
@page  
@namespace RazorPagesIntro.Pages.Customers  
  
@model NameSpaceModel  
  
<h2>Name space</h2>  
<p>  
    @Model.Message  
</p>
```

此指令将为页面设置命名空间。`@model` 指令无需包含命名空间。

_ViewImports.cshtml 中包含 `@namespace` 指令后，指定的命名空间将为在导入 `@namespace` 指令的页面中生成的命名空间提供前缀。生成的命名空间的剩余部分(后缀部分)是包含 _ViewImports.cshtml 的文件夹与包含页面的文件夹之间以点分隔的相对路径。

例如，代码隐藏文件 Pages/Customers/Edit.cshtml.cs 显式设置命名空间：

```
namespace RazorPagesContacts.Pages  
{  
    public class EditModel : PageModel  
    {  
        private readonly AppDbContext _db;  
  
        public EditModel(AppDbContext db)  
        {  
            _db = db;  
        }  
  
        // Code removed for brevity.
```

Pages/_ViewImports.cshtml 文件设置以下命名空间：

```
@namespace RazorPagesContacts.Pages  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

为 Pages/Customers/Edit.cshtml Razor 页面生成的命名空间与代码隐藏文件相同。已对 `@namespace` 指令进行设计，因此添加到项目的 C# 类和页面生成的代码可直接工作，而无需添加代码隐藏文件的 `@using` 指令。

`@namespace` 也可用于传统的 Razor 视图。

原始的 Pages/Create.cshtml 视图文件：

```

@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>

```

更新后的 Pages/Create.cshtml 视图文件：

```

@page
@model CreateModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>

```

Razor 页面初学者项目包含 Pages/_ValidationScriptsPartial.cshtml，它与客户端验证联合。

页面的 URL 生成

之前显示的 `Create` 页面使用 `RedirectToPage`：

```

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}

```

应用具有以下文件/文件夹结构：

- /Pages
 - Index.cshtml
 - /Customers
 - Create.cshtml

- Edit.cshtml
- Index.cshtml

成功后，Pages/Customers/Create.cshtml 和 Pages/Customers/Edit.cshtml 页面将重定向到 Pages/Index.cshtml。字符串 `/Index` 是用于访问上一页的 URI 的组成部分。可以使用字符串 `/Index` 生成 Pages/Index.cshtml 页面的 URI。例如：

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">My Index Page`
- `RedirectToPage("/Index")`

页面名称是从根“/Pages”文件夹到页面的路径（包含前导 `/`，例如 `/Index`）。与硬编码 URL 相比，前面的 URL 生成示例提供了改进的选项和功能。URL 生成使用[路由](#)，并且可以根据目标路径定义路由的方式生成参数并对参数编码。

页面的 URL 生成支持相对名称。下表显示了 Pages/Customers/Create.cshtml 中不同的 `RedirectToPage` 参数选择的索引页：

REDIRECTTOPAGE(X)	页
<code>RedirectToPage("/Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("./Index");</code>	<code>Pages/Customers/Index</code>
<code>RedirectToPage("../Index")</code>	<code>Pages/Index</code>
<code>RedirectToPage("Index")</code>	<code>Pages/Customers/Index</code>

`RedirectToPage("Index")`、`RedirectToPage("./Index")` 和 `RedirectToPage("../Index")` 是相对名称。结合 `RedirectToPage` 参数与当前页的路径来计算目标页面的名称。

构建结构复杂的站点时，相对名称链接很有用。如果使用相对名称链接文件夹中的页面，则可以重命名该文件夹。所有链接仍然有效（因为这些链接未包含此文件夹名称）。

ViewData 特性

可以通过 [ViewDataAttribute](#) 将数据传递到页面。控制器或 Razor 页面模型上使用 `[ViewData]` 修饰的属性将其值存储在 [ViewDataDictionary](#) 中并从此处进行加载。

在下面的示例中，`AboutModel` 包含使用 `[ViewData]` 修饰的 `Title` 属性。`Title` 属性设置为“关于”页面的标题：

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

在“关于”页面中，以模型属性的形式访问 `Title` 属性：

```
<h1>@Model.Title</h1>
```

在布局中，从 ViewData 字典读取标题：

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>@ViewData["Title"] - WebApplication</title>
...

```

TempData

ASP.NET 在[控制器](#)上公开了 `TempData` 属性。此属性存储未读取的数据。`Keep` 和 `Peek` 方法可用于检查数据，而不执行删除。多个请求需要数据时，`TempData` 有助于进行重定向。

`[TempData]` 是 ASP.NET Core 2.0 中的新属性，在控制器和页面上受支持。

下面的代码使用 `TempData` 设置 `Message` 的值：

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

Pages/Customers/Index.cshtml 文件中的以下标记使用 `TempData` 显示 `Message` 的值。

```
<h3>Msg: @Model.Message</h3>
```

Pages/Customers/Index.cshtml.cs 页面模型将 `[TempData]` 属性应用到 `Message` 属性。

```
[TempData]
public string Message { get; set; }
```

请参阅 [TempData](#) 了解详细信息。

针对一个页面的多个处理程序

以下页面使用 `asp-page-handler` 标记帮助程序为两个页面处理程序生成标记：

```
@page
@model CreateFATHModel

<html>
<body>
<p>
    Enter your name.
</p>
<div asp-validation-summary="All"></div>
<form method="POST">
    <div>Name: <input asp-for="Customer.Name" /></div>
    <input type="submit" asp-page-handler="JoinList" value="Join" />
    <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
</form>
</body>
</html>
```

前面示例中的窗体包含两个提交按钮，每个提交按钮均使用 `FormActionTagHelper` 提交到不同的 URL。

`asp-page-handler` 是 `asp-page` 的配套属性。`asp-page-handler` 生成提交到页面定义的各个处理方法的 URL。未指定 `asp-page`，因为示例已链接到当前页面。

页面模型：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpper();
            return await OnPostJoinListAsync();
        }
    }
}

```

前面的代码使用已命名处理程序方法。已命名处理程序方法通过采用名称中 `On<HTTP Verb>` 之后及 `Async` 之前的文本(如果有)创建。在前面的示例中，页面方法是 `OnPostJoinListAsync` 和 `OnPostJoinListUCAsync`。删除 `OnPost` 和 `Async` 后，处理程序名称为 `JoinList` 和 `JoinListUC`。

```

<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />

```

使用前面的代码时，提交到 `OnPostJoinListAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH?handler=JoinList`。提交到 `OnPostJoinListUCAsync` 的 URL 路径为 `http://localhost:5000/Customers/CreateFATH?handler=JoinListUC`。

自定义路由

如果你不喜欢 URL 中的查询字符串 `?handler=JoinList`，可以更改路由，将处理程序名称放在 URL 的路径部分。可以通过在 `@page` 指令后面添加使用双引号括起来的路由模板来自定义路由。

```

@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>

```

前面的路由将处理程序放在了 URL 路径中，而不是查询字符串中。`handler` 前面的 `?` 表示路由参数为可选。

可以使用 `@page` 将其他段和参数添加到页面的路由中。其中的任何内容均会被追加到页面的默认路由中。不支持使用绝对路径或虚拟路径更改页面的路由（如 `"~/Some/Other/Path"`）。

配置和设置

若要配置高级选项，请在 MVC 生成器上使用 `AddRazorPagesOptions` 扩展方法：

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
    {
        options.RootDirectory = "/MyPages";
        options.Conventions.AuthorizeFolder("/MyPages/Admin");
    });
}

```

目前，可以使用 `RazorPagesOptions` 设置页面的根目录，或者为页面添加应用程序模型约定。通过这种方式，我们在将来会实现更多扩展功能。

若要预编译视图，请参阅 [Razor 视图编译](#)。

[下载或查看示例代码](#).

请参阅 [Razor 页面入门](#)，这篇文章以本文为基础编写。

指定 Razor 页面位于内容根目录中

默认情况下，Razor 页面位于 `/Pages` 目录的根位置。向 `AddMvc` 添加 `WithRazorPagesAtContentRoot`，以指定 Razor 页面位于应用的内容根目录 (`ContentRootPath`) 中：

```

services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    ...
})
    .WithRazorPagesAtContentRoot();

```

指定 Razor 页面位于自定义根目录中

向 `AddMvc` 添加 `WithRazorPagesRoot`，以指定 Razor 页面位于应用中自定义根目录位置（提供相对路

径)：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    ...
})
.WithRazorPagesRoot("/path/to/razor/pages");
```

请参阅

- [ASP.NET Core 简介](#)
- [Razor 语法](#)
- [Razor 页面入门](#)
- [Razor 页面授权约定](#)
- [Razor 页面自定义路由和页面模型提供程序](#)
- [Razor 页面单位与集成测试](#)

ASP.NET Core 中的 Razor 页面的筛选方法

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

Razor 页面筛选器 [IPageFilter](#) 和 [IAsyncPageFilter](#) 允许 Razor 页面在运行 Razor 页面处理程序的前后运行代码。Razor 页面筛选器与 [ASP.NET Core MVC 操作筛选器](#)类似, 但它们不能应用于单个页面处理程序方法。

Razor 页面筛选器:

- 在选择处理程序方法后但在模型绑定发生前运行代码。
- 在模型绑定完成后, 执行处理程序方法前运行代码。
- 在执行处理程序方法后运行代码。
- 可在页面或全局范围内实现。
- 无法应用于特定的页面处理程序方法。

代码可在使用页面构造函数或中间件执行处理程序方法前运行, 但仅 Razor 页面筛选器可访问 [HttpContext](#)。筛选器具有 [FilterContext](#) 派生参数, 可用于访问 [HttpContext](#)。例如, [实现筛选器属性示例](#)将标头添加到响应中, 而使用构造函数或中间件则无法做到这点。

[查看或下载示例代码\(如何下载\)](#)

Razor 页面筛选器提供的以下方法可在全局或页面级应用:

- 同步方法:
 - [OnPageHandlerSelected](#): 在选择处理程序方法后但在模型绑定发生前调用。
 - [OnPageHandlerExecuting](#): 在执行处理器方法前, 模型绑定完成后调用。
 - [OnPageHandlerExecuted](#): 在执行处理器方法后, 生成操作结果前调用。
- 异步方法:
 - [OnPageHandlerSelectionAsync](#): 在选择处理程序方法后, 但在模型绑定发生前, 进行异步调用。
 - [OnPageHandlerExecutionAsync](#): 在调用处理程序方法前, 但在模型绑定结束后, 进行异步调用。

注意

筛选器接口的同步和异步版本任意实现一个, 而不是同时实现。该框架会先查看筛选器是否实现了异步接口, 如果是, 则调用该接口。如果不是, 则调用同步接口的方法。如果两个接口都已实现, 则只会调用异步方法。对页面中的替代应用相同的规则, 同步替代或异步替代只能任选其一实现, 不可二者皆选。

全局实现 Razor 页面筛选器

以下代码实现了 [IAsyncPageFilter](#):

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace PageFilter.Filters
{
    public class SampleAsyncPageFilter : IAsyncPageFilter
    {
        private readonly ILogger _logger;

        public SampleAsyncPageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public async Task OnPageHandlerSelectionAsync(
            PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global OnPageHandlerSelectionAsync called.");
            await Task.CompletedTask;
        }

        public async Task OnPageHandlerExecutionAsync(
            PageHandlerExecutingContext context,
            PageHandlerExecutionDelegate next)
        {
            _logger.LogDebug("Global OnPageHandlerExecutionAsync called.");
            await next.Invoke();
        }
    }
}
```

在前面的代码中，`ILogger` 不是必需的。它在示例中用于提供应用程序的跟踪信息。

以下代码启用 `Startup` 类中的 `SampleAsyncPageFilter`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SampleAsyncPageFilter(_logger));
    });
}
```

以下代码显示完整的 `Startup` 类：

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using PageFilter.Filters;

namespace PageFilter
{
    public class Startup
    {
        ILogger _logger;
        public Startup	ILoggerFactory loggerFactory, IConfiguration configuration
        {
            _logger = loggerFactory.CreateLogger<GlobalFiltersLogger>();
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc(options =>
            {
                options.Filters.Add(new SampleAsyncPageFilter(_logger));
            });
        }

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

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();

            app.UseMvc();
        }
    }
}

```

以下代码调用 `AddFolderApplicationModelConvention` 将 `SampleAsyncPageFilter` 仅应用于 /subFolder 中的页面：

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
    {
        options.Conventions.AddFolderApplicationModelConvention(
            "/subFolder",
            model => model.Filters.Add(new SampleAsyncPageFilter(_logger)));
    });
}

```

以下代码实现同步的 `IPageFilter`：

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;

namespace PageFilter.Filters
{
    public class SamplePageFilter : IPageFilter
    {
        private readonly ILogger _logger;

        public SamplePageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public void OnPageHandlerSelected(PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerSelected called.");
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
        {
            _logger.LogDebug("Global sync PageHandlerExecutingContext called.");
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerExecuted called.");
        }
    }
}
```

以下代码启用 `SamplePageFilter`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SamplePageFilter(_logger));
    });
}
```

通过重写筛选器方法实现 Razor 页面筛选器

以下代码替代同步的 Razor 页面筛选器：

```
[!code-csharp>Main]
```

实现筛选器属性

基于属性的内置筛选器 `OnResultExecutionAsync` 可以进行子类化。以下筛选器向响应添加标头：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(_name, new string[] { _value });
        }
    }
}
```

以下代码应用 `AddHeader` 属性：

```
[AddHeader("Author", "Rick")]
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILogger<ContactModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public async Task OnGetAsync()
    {
        Message = "Your contact page.";
        _logger.LogDebug("Contact/OnGet");
        await Task.CompletedTask;
    }
}
```

有关重写顺序的说明，请参阅[重写默认顺序](#)。

有关将筛选器管道与筛选器短路的说明，请参阅[取消和设置短路](#)。

授权筛选器属性

`Authorize` 属性可应用于 `PageModel`：

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel
    {
        public IActionResult OnGet() => Page();
    }
}
```

使用 ASP.NET Core 中的 Razor 类库项目创建可重用 UI。

2018/5/14 • 6 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

Razor 视图、页面、控制器、页模型和数据模型可以创建在 Razor 类库 (RCL) 中。RCL 可以打包并重复使用。应用程序可以包括 RCL，并重写其中包含的视图和页面。如果在 Web 应用和 RCL 中都能找到视图、分部视图或 Razor 页面，则 Web 应用中的 Razor 标记 (.cshtml 文件) 优先。

此功能需要 [.NET Core SDK 2.1 RC1 or later](#)

注意

ASP.NET Core 2.1 是预览版，不推荐用于生产环境。

[查看或下载示例代码 \(如何下载\)](#)

创建一个包含 Razor UI 的类库

- [Visual Studio](#)
 - [.NET Core CLI](#)
- 从 Visual Studio“文件”菜单中选择“新建”>“项目”。
 - 选择“ASP.NET Core Web 应用程序”。
 - 验证是否已选择 ASP.NET Core 2.1 或更高版本。
 - 选择“Razor 类库”>“确定”。

将 Razor 文件添加到 RCL。

建议将 RCL 内容转至“区域”文件夹。

引用 Razor 类库内容

可以通过以下方式引用 RCL:

- NuGet 包。请参阅[创建 NuGet 包](#)、[dotnet 添加包](#)和[创建和发布 NuGet 包](#)。
- {ProjectName}.csproj。请查看[dotnet-add 引用](#)。

RCL 中的分部文件访问

对于 RCL 之外的内容，ASP.NET Core 运行时不会搜索 RCL 中的分部文件。

例如，在示例下载中，不能在 WebApp1\Pages\About.cshtml 中引用

RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 分部视图。然而 RCL (RazorUIClassLib/) 中的页面可以访问 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtm。

演练: 创建 Razor 类库项目并从 Razor 页面项目使用它

可以下载并测试[完整项目](#)，无需创建项目。示例下载包含附加代码和链接，以方便测试项目。可以在[此 GitHub 问题](#)中留下反馈，评论下载示例和分步说明的对比。

测试下载应用

如果尚未下载已完成的应用，并更愿意创建演练项目，请跳转至[下一节](#)。

- [Visual Studio](#)
- [.NET Core CLI](#)

在 Visual Studio 中打开 .sln 文件。运行应用。

按“测试 Test WebApp1”中的说明进行操作

创建 Razor 类库

本部分创建 Razor 类库 (RCL)。将 Razor 文件添加到 RCL。

- [Visual Studio](#)
- [.NET Core CLI](#)

创建 RCL 项目：

- 从 Visual Studio“文件”菜单中选择“新建”>“项目”。
- 选择“ASP.NET Core Web 应用程序”。
- 将应用命名为 RazorUIClassLib。
- 验证是否已选择 ASP.NET Core 2.1 或更高版本。
- 选择“Razor 类库”>“确定”。

创建 Razor 页面 Web 应用：

- 在解决方案资源管理器中，右键单击解决方案 >“添加”>“新建项目”。
- 选择“ASP.NET Core Web 应用程序”。
- 将应用命名为 WebApp1。
- 验证是否已选择 ASP.NET Core 2.1 或更高版本。
- 选择“Web 应用程序”>“确定”。

将 Razor 文件和文件夹添加到该项目。

- 添加一个名为 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 的 Razor 分部视图文件。
- 使用以下代码替换 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 中的标记：

```
<h3> WebApp1 _Message.cshtml partial view.</h3>

<p>RazorUIClassLib\Areas\MyFeature\Pages\Shared\_Message.cshtml</p>
```

- 将 WebApp1 项目中的 _ViewStart.cshtml 文件复制到 RazorUIClassLib/Areas/MyFeature/Pages/_ViewStart.cshtml。

使用 Razor 页面项目的布局需要 [Viewstart](#) 文件。

从 Razor 页面项目使用 Razor UI 库

- [Visual Studio](#)
- [.NET Core CLI](#)

- 在解决方案资源管理器中，右键单击“WebApp1”，然后选择“设为启动项目”。
- 在解决方案资源管理器中，右键单击“WebApp1”，然后选择“生成依赖项”>“项目依赖项”。
- 将 RazorUIClassLib 勾选为 WebApp1 的依赖项。
- 在解决方案资源管理器中，右键单击“WebApp1”，选择“添加”>“引用”。
- 在“引用管理器”对话框中勾选“RazorUIClassLib”>“确定”。

运行应用。

测试 WebApp1

验证正在使用的 Razor UI 类库。

- 浏览到 `/MyFeature/Page1`。

重写视图、分部视图和页面

如果在 Web 应用和 Razor 类库中都能找到视图、分部视图或 Razor 页面，则 Web 应用中的 Razor 标记(.cshtml 文件)优先。例如，将 WebApp1/Areas/MyFeature/Pages/Page1.cshtml 添加到 WebApp1，则 WebApp1 中的 Page1 会优先于 Razor 类库中的 Page1。

在示例下载中，将 WebApp1/Areas/MyFeature2 重命名为 WebApp1/Areas/MyFeature 来测试优先级。

将 RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml 分部视图复制到 WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml。更新标记以指示新的位置。生成并运行应用，验证使用部分的应用版本。

ASP.NET Core Razor SDK

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

注意

ASP.NET Core 2.1 is in preview and not recommended for production use.

[.NET Core SDK 2.1 RC1 or later](#) 包含 `Microsoft.NET.Sdk.Razor` MSBuild SDK (Razor SDK)。Razor SDK:

- 针对基于 ASP.NET Core MVC 的项目，围绕包含 `Razor` 文件的项目的生成、打包和发布设定了体验标准。
- 包含一组预定义的目标、属性和项目，它们允许自定义 `Razor` 文件的编译。

系统必备

[.NET Core SDK 2.1 RC1 or later](#)

使用 Razor SDK

大多数 Web 应用无需明确引用 Razor SDK。

要使用 Razor SDK 来生成包含 Razor 视图或 Razor 页面的类库，请执行以下操作：

- 使用 `Microsoft.NET.Sdk.Razor` 而非 `Microsoft.NET.Sdk`：

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  ...
</Project>
```

- 通常需要对 `Microsoft.AspNetCore.Mvc` 的包引用以引入生成和编译 Razor 页面和 Razor 视图所需的其他依赖项。至少，项目需要将包引用添加到：

- `Microsoft.AspNetCore.Razor.Design`
- `Microsoft.AspNetCore.Mvc.Razor.Extensions`

前面的包包含在 `Microsoft.AspNetCore.Mvc` 中。以下标记显示了使用 Razor SDK 为 ASP.NET Core Razor 页面应用生成 Razor 文件的基本 .csproj 文件：

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.0-preview2-final" />
  </ItemGroup>

</Project>
```

属性

以下属性控制项目生成过程中 Razor 的 SDK 行为：

- `RazorCompileOnBuild` : 为 `true` 时, 在生成项目过程中, 编译并发出 Razor 程序集。默认为 `true`。
- `RazorCompileOnPublish` : 为 `true` 时, 在发布项目过程中, 编译并发出 Razor 程序集。默认为 `true`。

以下属性和项用于配置 Razor SDK 的输入和输出：

项	描述
<code>RazorGenerate</code>	输入到代码生成目标的项元素 (.cshtml 文件)。
<code>RazorCompile</code>	输入到 Razor 编译目标的项元素 (.cs 文件)。使用此 <code>ItemGroup</code> 指定要编译到 Razor 程序集中的其他文件。
<code>RazorAssemblyAttribute</code>	用于编码生成 Razor 程序集属性的项元素。例如: <pre><RazorAssemblyAttribute Include="System.Reflection.AssemblyMetadataAttribute" _Parameter1="BuildSource" _Parameter2="https://docs.asp.net/"></pre>
<code>RazorEmbeddedResource</code>	作为嵌入的资源添加到生成的 Razor 程序集中的项元素
属性	描述
<code>RazorTargetName</code>	Razor 生成的程序集的文件名(不含扩展名)。
<code>RazorOutputPath</code>	Razor 输出目录。
<code>RazorCompileToolset</code>	用于确定用于生成 Razor 程序集的工具集。有效值为 <code>Implicit</code> 和 <code>PrecompilationTool</code> 。
<code>EnableDefaultContentItems</code>	为 <code>true</code> 时, 包括某些文件类型(例如 .cshtml 文件)作为项目中内容。当通过 Microsoft.NET.Sdk.Web 引用时, 还包括 wwwroot 下的所有文件和配置文件。
<code>EnableDefaultRazorGenerateItems</code>	为 <code>true</code> 时, 包括 <code>RazorGenerate</code> 项中 <code>Content</code> 项的 .cshtml 文件。
<code>GenerateRazorTargetAssemblyInfo</code>	为 <code>true</code> 时, 生成 .cs 文件(其中包含由 <code>RazorAssemblyAttribute</code> 指定的属性), 并将其包含在编译输出中。
<code>EnableDefaultRazorTargetAssemblyInfoAttributes</code>	为 <code>true</code> 时, 将一组默认的程序集属性添加到 <code>RazorAssemblyAttribute</code> 。
<code>CopyRazorGenerateFilesToPublishDirectory</code>	为 <code>true</code> 时, 将 <code>RazorGenerate</code> 项 (.cshtml) 文件复制到发布目录。如果在生成时或发布时参与编译, 则通常发布的应用程序无需 Razor 文件。默认为 <code>false</code> 。
<code>CopyRefAssembliesToPublishDirectory</code>	为 <code>true</code> 时, 将引用程序集项复制到发布目录。如果在生成时或发布时出现 Razor 编译, 则通常发布的应用程序无需引用程序集。如果发布的应用程序需要运行时编译(例如, 在运行时修改 cshtml 文件或使用嵌入的视图), 则设置为 <code>true</code> 。默认为 <code>false</code> 。

属性	描述
IncludeRazorContentInPack	为 <code>true</code> 时, 所有 Razor 内容项(.cshtml 文件)将标记为包含在生成的 NuGet 包中。默认为 <code>false</code> 。
EmbedRazorGenerateSources	为 <code>true</code> 时, 将 RazorGenerate (.cshtml) 项作为嵌入的文件添加到生成的 Razor 程序集中。默认为 <code>false</code> 。
UseRazorBuildServer	为 <code>true</code> 时, 使用永久生成服务器进程来卸载代码生成工作。 默认值为 <code>UseSharedCompilation</code> 。

目标

Razor SDK 定义两个主要目标:

- `RazorGenerate` - 代码从 RazorGenerate 项元素生成 .cs 文件。使用 `RazorGenerateDependsOn` 属性指定可在此目标之前或之后运行的其他目标。
- `RazorCompile` - 将生成的 .cs 文件编译到 Razor 程序集中。使用 `RazorCompileDependsOn` 指定可在此目标之前或之后运行的其他目标。

ASP.NET Core MVC 概述

2018/4/10 • 11 min to read • [Edit Online](#)

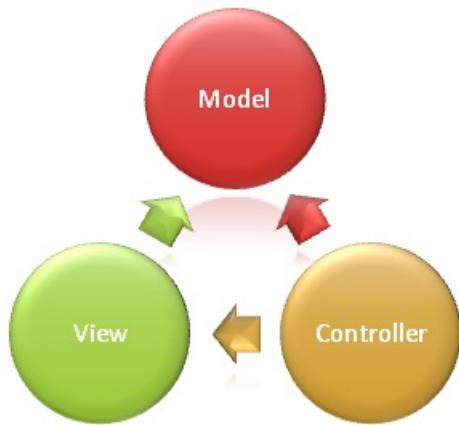
作者: Steve Smith

ASP.NET Core MVC 是使用“模型-视图-控制器”设计模式构建 Web 应用和 API 的丰富框架。

什么是 MVC 模式？

模型-视图-控制器 (MVC) 体系结构模式将应用程序分成 3 个主要组件组: 模型、视图和控制器。此模式有助于实现**关注点分离**。使用此模式，用户请求被路由到控制器，后者负责使用模型来执行用户操作和/或检索查询结果。控制器选择要显示给用户的视图，并为其提供所需的任何模型数据。

下图显示 3 个主要组件及其相互引用关系：



这种责任划分有助于根据复杂性缩放应用程序，因为这更易于编码、调试和测试有单一作业（并遵循 [Single Responsibility Principle](#)（单一责任原则）的某些内容（模型、视图或控制器）。但这会加大更新、测试和调试代码的难度，该代码在这 3 个领域的两个或多个领域间存在依赖关系。例如，用户界面逻辑的变更频率往往高于业务逻辑。如果将表示代码和业务逻辑组合在单个对象中，则每次更改用户界面时都必须修改包含业务逻辑的对象。这常常会引发错误，并且需要在每次进行细微的用户界面更改后重新测试业务逻辑。

注意

视图和控制器均依赖于模型。但是，模型既不依赖于视图，也不依赖于控制器。这是分离的一个关键优势。这种分离允许模型独立于可视化展示进行构建和测试。

模型责任

MVC 应用程序的模型 (M) 表示应用程序和任何应由其执行的业务逻辑或操作的状态。业务逻辑应与保持应用程序状态的任何实现逻辑一起封装在模型中。强类型视图通常使用 ViewModel 类型，旨在包含要在该视图上显示的数据。控制器从模型创建并填充 ViewModel 实例。

注意

可通过多种方法在使用 MVC 体系结构模式的应用中组织模型。详细了解某些[不同种类的模型类型](#)。

视图责任

视图 (V) 负责通过用户界面展示内容。它们使用 [Razor 视图引擎](#) 在 HTML 标记中嵌入 .NET 代码。视图中应该有最小逻辑，并且其中的任何逻辑都必须与展示内容相关。如果发现需要在视图文件中执行大量逻辑以显示复杂模型中的数据，请考虑使用 [View Component](#)、[ViewModel](#) 或视图模板来简化视图。

控制器职责

控制器 (C) 是处理用户交互、使用模型并最终选择要呈现的视图的组件。在 MVC 应用程序中，视图仅显示信息；控制器处理并响应用户输入和交互。在 MVC 模式中，控制器是初始入口点，负责选择要使用的模型类型和要呈现的视图（因此得名 - 它控制应用如何响应给定请求）。

注意

控制器不应由于责任过多而变得过于复杂。要阻止控制器逻辑变得过于复杂，请使用 [Single Responsibility Principle](#)（单一责任原则）将业务逻辑推出控制器并推入域模型。

提示

如果发现控制器操作经常执行相同类型的操作，则可将这些常见操作移入[筛选器](#)，并遵守“[不要自我重复](#)”原则。

什么是 ASP.NET Core MVC

ASP.NET Core MVC 框架是轻量级、开源、高度可测试的演示框架，并针对 ASP.NET Core 进行了优化。

ASP.NET Core MVC 提供一种基于模式的方式，用于生成可彻底分开管理事务的动态网站。它提供对标记的完全控制，支持 TDD 友好开发并使用最新的 Web 标准。

功能

ASP.NET Core MVC 包括以下功能：

- [路由](#)
- [模型绑定](#)
- [模型验证](#)
- [依赖关系注入](#)
- [筛选器](#)
- [区域](#)
- [Web API](#)
- [可测试性](#)
- [Razor 视图引擎](#)
- [强类型视图](#)
- [标记帮助程序](#)
- [视图组件](#)

路由

ASP.NET Core MVC 建立在 [ASP.NET Core 的路由](#)之上，是一个功能强大的 URL 映射组件，可用于生成具有易于理解和可搜索 URL 的应用程序。它可让你定义适用于搜索引擎优化 (SEO) 和链接生成的应用程序 URL 命名模式，而不考虑如何组织 Web 服务器上的文件。可以使用支持路由值约束、默认值和可选值的方便路由模板语法来定义路由。

通过基于约定的路由，可以全局定义应用程序接受的 URL 格式以及每个格式映射到给定控制器上特定操作方法的方式。接收传入请求时，路由引擎分析 URL 并将其匹配到定义的 URL 格式之一，然后调用关联的控制器操作方法。

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```

借助属性路由，可以通过用定义应用程序路由的属性修饰控制器和操作来指定路由信息。这意味着路由定义位于与之相关联的控制器和操作旁。

```
[Route("api/{controller}")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        ...
    }
}
```

模型绑定

ASP.NET Core MVC [模型绑定](#)将客户端请求数据(窗体值、路由数据、查询字符串参数、HTTP 头)转换到控制器可以处理的对象中。因此，控制器逻辑不必找出传入的请求数据；它只需具备作为其操作方法的参数的数据。

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null) { ... }
```

模型验证

ASP.NET Core MVC 通过使用数据注释验证属性修饰模型对象来支持[验证](#)。验证属性在值发布到服务器前在客户端上进行检查，并在调用控制器操作前在服务器上进行检查。

```
using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

控制器操作：

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}
```

框架处理客户端和服务器上的验证请求数据。在模型类型上指定的验证逻辑作为非介入式注释添加到呈现的视图，并使用[jQuery 验证](#)在浏览器中强制执行。

依赖关系注入

ASP.NET Core 内置有对依赖关系注入 (DI) 的支持。在 ASP.NET Core MVC 中，**控制器**可通过其构造函数请求所需服务，使其能够遵循 [Explicit Dependencies Principle](#)(显式依赖关系原则)。

应用还可通过 `@inject` 指令使用[视图文件中的依赖关系注入](#)：

```
@inject SomeService ServiceName
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>
</body>
</html>
```

筛选器

[筛选器](#)帮助开发者封装横切关注点，例如异常处理或授权。筛选器允许操作方法运行自定义预处理和后处理逻辑，并且可以配置为在给定请求的执行管道内的特定点上运行。筛选器可以作为属性应用于控制器或操作(也可以全局运行)。此框架中包括多个筛选器(例如 `Authorize`)。

```
[Authorize]
public class AccountController : Controller
{
```

区域

[区域](#)提供将大型 ASP.NET Core MVC Web 应用分区为较小功能分组的方法。区域是应用程序内的一个 MVC 结构。在 MVC 项目中，模型、控制器和视图等逻辑组件保存在不同的文件夹中，MVC 使用命名约定来创建这些组件之间的关系。对于大型应用，将应用分区为独立的高级功能区域可能更有利。例如，具有多个业务单位(如结账、计费、搜索等)的电子商务应用。每个单位都有自己的逻辑组件视图、控制器和模型。

Web API

除了作为生成网站的强大平台，ASP.NET Core MVC 还对生成 Web API 提供强大的支持。可以生成可连接大量客户端(包括浏览器和移动设备)的服务。

Framework 包括到内置支持通过 HTTP 内容协商支持[设置数据的格式](#)作为 JSON 或 XML。编写[自定义格式化程序](#)以添加对自己格式的支持。

使用链接生成启用对超媒体的支持。轻松启用对[跨域资源共享 \(CORS\)](#) 的支持，以便 Web API 可以跨多个 Web 应用程序共享。

可测试性

接口和依赖关系注入框架的使用使其适合对单元测试，和框架包括功能(如 `TestHost` 和 `InMemory` 实体框架提供程序)，使[集成测试](#)快速和轻松以及。详细了解[如何测试控制器逻辑](#)。

Razor 视图引擎

ASP.NET Core MVC 视图使用 [Razor 视图引擎](#)呈现视图。Razor 是一种紧凑、富有表现力且流畅的模板标记语言，用于使用嵌入式 C# 代码定义视图。Razor 用于在服务器上动态生成 Web 内容。可以完全混合服务器代码与客户端内容和代码。

```
<ul>
    @for (int i = 0; i < 5; i++) {
        <li>List item @i</li>
    }
</ul>
```

使用 Razor 视图引擎可以定义 [布局](#)、[分部视图](#) 和可替换部分。

强类型视图

可以基于模型强类型化 MVC 中的 Razor 视图。控制器可以将强类型化的模型传递给视图，使视图具备类型检查和 IntelliSense 支持。

例如，以下视图呈现类型为 `IEnumerable<Product>` 的模型：

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
    }
</ul>
```

标记帮助程序

[标记帮助程序](#) 使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。可以使用标记帮助程序定义自定义标记（例如 `<environment>`），或者修改现有标记的行为（例如 `<label>`）。标记帮助程序基于元素名称及其属性绑定到特定的元素。它们提供了服务器端呈现的优势，同时仍然保留了 HTML 编辑体验。

有多种常见任务（例如创建窗体、链接、加载资产等）的内置标记帮助程序，公共 GitHub 存储库和 NuGet 包中甚至还有更多可用标记帮助程序。标记帮助程序使用 C# 创建，基于元素名称、属性名称或父标记以 HTML 元素为目标。例如，内置 `LinkTagHelper` 可以用来创建指向 `AccountsController` 的 `Login` 操作的链接：

```
<p>
    Thank you for confirming your email.
    Please <a asp-controller="Account" asp-action="Login">Click here to Log in</a>.
</p>
```

可以使用 `EnvironmentTagHelper` 在视图中包括基于运行时环境（例如开发、暂存或生产）的不同脚本（例如原始或缩减脚本）：

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
           asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
           asp-fallback-test="window.jQuery">
    </script>
</environment>
```

标记帮助程序提供 HTML 友好型开发体验和用于创建 HTML 和 Razor 标记的丰富 IntelliSense 环境。大多数内置标记帮助程序以现有 HTML 元素为目标，为该元素提供服务器端属性。

视图组件

通过 [视图组件](#) 可以包装呈现逻辑并在整个应用程序中重用它。这些组件类似于 [分部视图](#)，但具有关联逻辑。

ASP.NET Core 中的模型绑定

2018/5/14 • 8 min to read • [Edit Online](#)

作者: [Rachel Appel](#)

模型绑定简介

ASP.NET Core MVC 中的模型绑定将 HTTP 请求中的数据映射到操作方法参数。这些参数可能是简单类型的参数，如字符串、整数或浮点数，也可能是复杂类型的参数。这是 MVC 的一项强大功能，因为不管数据的大小和复杂性，将传入数据映射到对应位置都是经常重复的方案。MVC 通过将绑定抽象出来解决了这一问题，使开发者不必在每个应用中重写同代码的稍微不同版本。向类型转换器代码写入自己的文本不仅繁琐乏味，而且容易出错。

模型绑定的工作原理

当 MVC 收到 HTTP 请求时，它会将此请求路由到控制器的特定操作方法。它基于路由数据中的内容决定要运行的操作方法，然后将 HTTP 请求中的值绑定到该操作方法的参数。以下列 URL 为例：

```
http://contoso.com/movies/edit/2
```

由于路由模板为 `{controller=Home}/{action=Index}/{id?}`，因此，`movies/edit/2` 将路由到 `Movies` 控制器及其 `Edit` 操作方法。此外，它还接受名为 `id` 的可选参数。操作方法的代码应如下所示：

```
public IActionResult Edit(int? id)
```

注意：URL 路由中的字符串不区分大小写。

MVC 将尝试按名称将请求数据绑定到操作参数。MVC 将使用参数名称和其公共可设置属性的名称查找每个参数的值。在以上示例中，唯一的操作参数名为 `id`，MVC 会将此参数绑定到路由值中具有相同名称的值。除路由值外，MVC 还会绑定来自请求各个部分的数据，并按一定顺序执行此操作。下面是一个数据源列表(按模型绑定查看的顺序排列)：

1. `Form values`：这些是使用 POST 方法进入 HTTP 请求的窗体值。(包括 jQuery POST 请求)。
2. `Route values`：路由提供的路由值集
3. `Query strings`：URI 的查询字符串部分。

注意：窗体值、路由数据和查询字符串均存储为名称/值对。

由于模型绑定请求名为 `id` 的键，而窗体值中没有任何名为 `id` 的键，因此，它将转到路由值以查找该键。在本示例中，该键是一个匹配项。发生绑定，并且值被转换为整数 2。使用 `Edit(string id)` 的同一请求将转换为字符串“2”。

到目前为止，示例使用的都是简单类型。在 MVC 中，简单类型是任何 .NET 基元类型或包含字符串类型转换器的类型。如果操作方法的参数是一个属性为简单类型和复杂类型的类，如 `Movie` 类型，则 MVC 的模型绑定仍可对其进行妥善处理。它使用反射和递归来遍历查找匹配项的复杂类型的属性。模型绑定查找模式 `parameter_name.property_name` 以将值绑定到属性。如果未找到此窗体的匹配值，它将尝试仅使用属性名称进行绑定。对于诸如 `Collection` 类型的类型，模型绑定将查找 `parameter_name[index]` 或仅 `[index]` 的匹配项。对于 `Dictionary` 类型，模型绑定的处理方

法与之类似，即请求 `parameter_name[key]` 或仅 `[key]`（只要键是简单类型）。受支持的键匹配字段名称 HTML 和针对相同模型类型生成的标记帮助程序。这样就可以启用往返值，以便用户仍能方便地输入内容来填充窗体字段，例如当创建或编辑的绑定数据未通过验证时。

为了进行绑定，类必须具有公共默认构造函数，并且要绑定的成员必须是公共可写属性。发生模型绑定时，在仅使用公共默认构造函数对类进行实例化后才可设置属性。

绑定参数时，模型绑定将停止查找具有该名称的值并向前移动以绑定下一个参数。否则，默认模型绑定行为将根据参数的类型将参数设置为其默认值：

- `T[]`：除 `byte[]` 类型的数组外，绑定将 `T[]` 类型的参数设置为 `Array.Empty<T>()`。
`byte[]` 类型的数组设置为 `null`。
- 引用类型：绑定通过默认构造函数创建类的实例而不设置属性。但是，模型绑定将 `string` 参数设置为 `null`。
- 可以为 `null` 的类型：可以为 `null` 的类型设置为 `null`。在上面的示例中，模型绑定将 `id` 设置为 `null`，因为其类型为 `int?`。
- 值类型：`T` 类型的不可以为 `null` 的值类型设置为 `default(T)`。例如，模型绑定将参数 `int id` 设置为 0。请考虑使用模型验证或可以为 `null` 的类型，而不是依赖于默认值。

如果绑定失败，MVC 不会引发错误。接受用户输入的每个操作均应检查 `ModelState.IsValid` 属性。

注意：控制器的 `ModelState` 属性中的每个输入均为包含 `Errors` 属性的 `ModelStateEntry`。很少需要自行查询此集合。请改用 `ModelState.IsValid`。

此外，MVC 在执行模型绑定时必须考虑一些特殊数据类型：

- `IFormFile`、`IEnumerable<IFormFile>`：作为 HTTP 请求一部分的一个或多个已上传文件。
- `CancellationToken`：用于取消异步控制器中的活动。

这些类型可绑定到操作参数或类类型的属性。

模型绑定完成后，将发生验证。对于绝大多数开发方案，默认模型绑定效果极佳。它还可以扩展，因此如果你有特殊需求，则可自定义内置行为。

通过特性自定义模型绑定行为

MVC 包含多种特性，可用于将其默认模型绑定行为定向到不同的源。例如，可以使用 `[BindRequired]` 或 `[BindNever]` 特性指定属性是否需要绑定，或绑定是否根本不应发生。或者，可以替代默认数据源，并指定模型绑定器的数据源。下面是模型绑定特性的列表：

- `[BindRequired]`：如果无法发生绑定，此特性将添加模型状态错误。
- `[BindNever]`：指示模型绑定器从不绑定到此参数。
- `[FromHeader]`、`[FromQuery]`、`[FromRoute]`、`[FromForm]`：使用这些特性指定要应用的确切绑定源。
- `[FromServices]`：此特性使用[依赖关系注入](#)绑定服务中的参数。
- `[FromBody]`：使用配置的格式化程序绑定请求正文中的数据。基于请求的内容类型，选择格式化程序。
- `[ModelBinder]`：用于替代默认模型绑定器、绑定源和名称。

需要替代模型绑定的默认行为时，特性是非常有用的工具。

绑定请求正文中的带格式数据

请求数据可以有各种格式，包括 JSON、XML 和许多其他格式。使用 `[FromBody]` 特性指示要将参数绑定到请求正文中的数据时，MVC 会使用一组已配置的格式化程序基于请求数据的内容类型对请求数据进行处理。默认情况下，MVC 包括用于处理 JSON 数据的 `JsonInputFormatter` 类，但你可以添加用于处理 XML 和其他自定义格式的其他格式化程序。

注意

对于用 `[FromBody]` 修饰的每个操作，最多可以有一个参数。ASP.NET Core MVC 运行时向格式化程序委托读取请求流的责任。读取参数的请求流后，通常不能为绑定其他 `[FromBody]` 参数而再次读取请求流。

注意

`JsonInputFormatter` 为默认格式化程序且基于 [Json.NET](#)。

除非有特性应用于 ASP.NET，否则它将基于 `Content-Type` 标头和参数类型来选择输入格式化程序。如果想要使用 XML 或其他格式，则必须在 `Startup.cs` 文件中配置该格式，但可能必须先使用 NuGet 获取对 `Microsoft.AspNetCore.Mvc.Formatters.Xml` 的引用。启动代码应如下所示：

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

`Startup.cs` 文件中的代码包含具有 `services` 参数的 `ConfigureServices` 方法，此方法可用于为 ASP.NET 应用生成服务。在此示例中，我们将添加 XML 格式化程序作为 MVC 将为此应用提供的服务。通过传递给 `AddMvc` 方法的 `options` 参数，可在应用启动后添加和管理筛选器、格式化程序和其他 MVC 系统选项。然后，将 `Consumes` 特性应用于控制器类或操作方法，以使用所需格式。

自定义模型绑定

可通过编写自己的自定义模型绑定器来扩展模型绑定。详细了解[自定义模型绑定](#)。

ASP.NET Core MVC 中的模型验证

2018/5/14 • 17 min to read • [Edit Online](#)

作者: [Rachel Appel](#)

模型验证简介

在将数据存储到数据库之前，应用必须先验证数据。必须检查数据是否存在潜在的安全威胁，确保数据已设置适当的类型和大小格式，并且必须符合相关规则。实施验证的过程可能有些单调乏味，但却必不可少。在 MVC 中，验证发生在客户端和服务器上。

幸运的是，.NET 已将验证抽象化为验证属性。这些属性包含验证代码，从而减少了所需编写的代码量。

[查看或下载 GitHub 中的示例。](#)

验证属性

验证属性用于配置模型验证，因此，在概念上类似于数据库表中字段上的验证。它包括诸如分配数据类型或必填字段之类的约束。其他类型的验证包括将向数据应用模式以强制实施业务规则，比如信用卡、电话号码或电子邮件地址。验证属性更易使用，并使这些要求的实施变得更容易。

下面是一个应用的已批注 `Movie` 模型，该应用用于存储电影和电视节目的相关信息。大多数属性都是必需属性，多个字符串属性具有长度要求。此外，还有一个针对 `Price` 属性设置的从 0 到 \$999.99 的数值范围限制，以及一个自定义验证特性。

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

通过读取整个模型即可显示有关此应用的数据的规则，从而使代码维护变得更轻松。下面是几个常用的内置验证属性：

- `[CreditCard]`：验证属性是否具有信用卡格式。
- `[Compare]`：验证某个模型中的两个属性是否匹配。

- `[EmailAddress]` : 验证属性是否具有电子邮件格式。
- `[Phone]` : 验证属性是否具有电话格式。
- `[Range]` : 验证属性值是否落在给定范围内。
- `[RegularExpression]` : 验证数据是否与指定的正则表达式匹配。
- `[Required]` : 将属性设置为必需属性。
- `[StringLength]` : 验证字符串属性是否最多具有给定的最大长度。
- `[Url]` : 验证属性是否具有 URL 格式。

MVC 支持从 `ValidationAttribute` 派生的所有用于验证的属性。在 `System.ComponentModel.DataAnnotations` 命名空间中可找到许多有用的验证属性。

在某些情况下，内置属性可能无法提供所需的功能。这时，就可以通过从 `ValidationAttribute` 派生或将模型更改为实现 `IValidableObject`，来创建自定义验证属性。

必需属性的使用说明

从本质上来说，需要不可以为 null 的值类型（如 `decimal`、`int`、`float` 和 `DateTime`），但不需要 `Required` 特性。应用不会对标记为 `Required` 的不可为 null 的类型执行任何服务器端验证检查。

对于不可为 null 的类型，MVC 模型绑定（与验证和验证属性无关）会拒绝包含缺失值或空白的表单域提交。如果目标属性上缺少 `BindRequired` 特性，模型绑定会忽略不可为 null 的类型的缺失数据，导致传入表单数据中缺少表单域。

`BindRequired` 属性（另请参阅[使用属性自定义模型绑定行为](#)）可用于确保表单数据的完整性。当应用于某个属性时，模型绑定系统要求该属性具有值。当应用于某个类型时，模型绑定系统要求该类型的所有属性都具有值。

使用 `Nullable<T>` 类型（例如，`decimal?` 或 `System.Nullable<decimal>`）并将其标记为 `Required` 时，将执行服务器端验证检查，就像该属性是标准的可以为 null 的类型（例如，`string`）一样。

客户端验证要求与标记为 `Required` 的模型属性对应的表单域以及未标记为 `Required` 的不可为 null 的类型属性具有值。`Required` 可用于控制客户端验证错误消息。

模型状态

模型状态表示已提交的 HTML 表单值中的验证错误。

MVC 将继续验证字段，直至达到错误数上限（默认为 200 个）。通过向 `Startup.cs` 文件中的 `ConfigureServices` 方法插入以下代码，可配置该数字：

```
services.AddMvc(options => options.MaxModelValidationErrors = 50);
```

处理模型状态错误

在调用每个控制器操作之前都会执行模型验证，将由操作方法负责检查 `ModelState.IsValid` 并作出正确反应。在许多情况下，正确的反应是返回错误响应，理想状况下会详细说明模型验证失败的原因。

某些应用会选择遵循标准约定来处理模型验证错误，在这种情况下，可以在筛选器中实现此类策略。应测试操作在有效模型状态和无效模型状态下的行为方式。

手动验证

完成模型绑定和验证后，可能需要重复其中的某些步骤。例如，用户可能在应输入整数的字段中输入了文本，或者你可能需要计算模型的某个属性的值。

你可能需要手动运行验证。为此，请调用 `TryValidateModel` 方法，如下所示：

```
TryValidateModel(movie);
```

自定义验证

验证属性适用于大多数验证需求。但是，某些验证规则特定于你的业务。你的规则可能不是常见的数据验证技术，比如确保字段是必填字段或符合一系列值。在这些情况下，自定义验证属性是一种不错的解决方案。在 MVC 中创建你自己的自定义验证属性很简单。只需从 `ValidationAttribute` 继承并重写 `IsValid` 方法。`IsValid` 方法采用两个参数，第一个是名为 `value` 的对象，第二个是名为 `validationContext` 的 `ValidationContext` 对象。`Value` 引用自定义验证程序要验证的字段中的实际值。

在下面的示例中，一项业务规则规定，用户不能将 1960 年以后发行的电影的流派设置为 `Classic`。

`[ClassicMovie]` 属性会先检查流派，如果是经典流派，则查看发行日期是否晚于 1960 年。如果晚于 1960 年，则验证失败。此属性采用一个表示年份的整数参数，可用于验证数据。可以在该属性的构造函数中捕获该参数的值，如下所示：

```
public class ClassicMovieAttribute : ValidationAttribute, IClientModelValidator
{
    private int _year;

    public ClassicMovieAttribute(int Year)
    {
        _year = Year;
    }

    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        Movie movie = (Movie)validationContext.ObjectInstance;

        if (movie.Genre == Genre.Classic && movie.ReleaseDate.Year > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

上面的 `movie` 变量表示一个 `Movie` 对象，其中包含要验证的表单提交中的数据。在此例中，验证代码会根据规则检查 `ClassicMovieAttribute` 类的 `IsValid` 方法中的日期和流派。如果验证成功，`IsValid` 会返回 `ValidationResult.Success` 代码；如果验证失败，则返回包含错误消息的 `ValidationResult`。当用户修改 `Genre` 字段并提交表单时，`ClassicMovieAttribute` 的 `IsValid` 方法将验证该电影是否为经典电影。将 `ClassicMovieAttribute` 像所有内置特性一样应用于属性（如 `ReleaseDate`）以确保执行验证，如前面的代码示例所示。由于此示例仅适用于 `Movie` 类型，因此建议使用 `IValidatableObject`，如下一段中所示。

也可以通过实现 `IValidatableObject` 接口上的 `Validate` 方法，将这段代码直接放入模型中。如果自定义验证特性可用于验证各个属性，则可使用 `IValidatableObject` 来实现类级别的验证，如下所示。

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
    {
        yield return new ValidationResult(
            $"Classic movies must have a release year earlier than {_classicYear}.",
            new[] { "ReleaseDate" });
    }
}
```

客户端验证

客户端验证极大地方便了用户。它节省了时间，让用户不必浪费时间等待服务器往返。从商业角度而言，即使每次只有几分之一秒，但如果每天有几百次，也会耗费大量的时间和成本，带来很多不必要的烦恼。简单直接的验证能够提高用户的工作效率和投入产出比。

你必须有一个包含适当的 JavaScript 脚本引用的视图，才能让客户端验证正常工作，如下所示。

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"></script>
```

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"></script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.
min.js"></script>
```

[jQuery 非介入式验证](#)脚本是一个基于热门 [jQuery Validate](#) 插件的自定义 Microsoft 前端库。如果没有 jQuery 非介入式验证，则必须在两个位置编码相同的验证逻辑：一次是在模型属性上的服务器端验证特性中，一次是在客户端脚本中（jQuery Validate 的 `validate()` 方法示例展示了这种情况可能的复杂程度）。MVC 的 [标记帮助程序](#) 和 [HTML 帮助程序](#) 则能够使用模型属性中的验证特性和类型元数据，呈现需要验证的表单元素中的 HTML 5 [data- 特性](#)。MVC 为内置属性和自定义属性生成 `data-` 属性。然后，jQuery 非介入式验证分析这些 `data-` 属性并将逻辑传递给 jQuery Validate，从而将服务器端验证逻辑有效地“复制”到客户端。可以使用相关标记帮助程序在客户端上显示验证错误，如下所示：

```
<div class="form-group">
    <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="ReleaseDate" class="form-control" />
        <span asp-validation-for="ReleaseDate" class="text-danger"></span>
    </div>
</div>
```

上面的标记帮助程序将呈现以下 HTML。请注意，HTML 输出中的 `data-` 特性与 `ReleaseDate` 属性的验证特性相对应。下面的 `data-val-required` 属性包含在用户未填写发行日期字段时将显示的错误消息。jQuery 非介入式验证将此值传递给 jQuery Validate `required()` 方法，该方法随后在随附的 `` 元素中显示该消息。

```

<form action="/Movies/Create" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <div class="text-danger"></div>
        <div class="form-group">
            <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
            <div class="col-md-10">
                <input class="form-control" type="datetime"
                    data-val="true" data-val-required="The ReleaseDate field is required."
                    id="ReleaseDate" name="ReleaseDate" value="" />
                <span class="text-danger field-validation-valid"
                    data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
            </div>
        </div>
    </div>
</form>

```

客户端验证将阻止提交，直到表单变为有效为止。“提交”按钮运行 JavaScript：要么提交表单要么显示错误消息。

MVC 基于属性的 .NET 数据类型确定类型特性值（有可能使用 `[DataType]` 特性进行重写）。`[DataType]` 基本特性不执行真正的服务器端验证。浏览器选择自己的错误消息，并根据需要显示这些错误，但 jQuery 非介入式验证包可以重写消息，并使它们与其他消息的显示保持一致。当用户应用 `[DataType]` 子类（比如 `[EmailAddress]`）时，最常发生这种情况。

向动态表单添加验证

由于 jQuery 非介入式验证会在第一次加载页面时将验证逻辑和参数传递到 jQuery Validate，因此，动态生成的表单不会自动展示验证。你必须指示 jQuery 非介入式验证在创建动态表单后立即对其进行分析。例如，下面的代码展示如何对通过 AJAX 添加的表单设置客户端验证。

```

$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})

```

`$.validator.unobtrusive.parse()` 方法采用 jQuery 选择器作为它的一个参数。此方法指示 jQuery 非介入式验证分析该选择器内表单的 `data-` 属性。这些属性的值随后传递到 jQuery Validate 插件中，以便表单展示所需的客户端验证规则。

向动态控件添加验证

也可以在动态生成各个控件（比如 `<input/>` 和 `<select/>`）时，更新表单上的验证规则。不能将用于这些元素的选择器直接传递到 `parse()` 方法，因为周围表单已进行分析并且不会更新。应当先删除现有的验证数据，然后重新分析整个表单，如下所示：

```

$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator") // Added by jQuery Validate
            .removeData("unobtrusiveValidation"); // Added by jQuery Unobtrusive Validation
        $.validator.unobtrusive.parse(form);
    }
})

```

IClientModelValidator

可为自定义属性创建客户端逻辑，创建 [jQuery 验证](#) 的适配器的 [非介入式验证](#) 将在验证过程中，在客户端上自动为你执行此逻辑。第一步是通过实现 `IClientModelValidator` 接口来控制要添加哪些 `data-` 属性，如下所示：

```

public void AddValidation(ClientModelValidationContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    MergeAttribute(context.Attributes, "data-val", "true");
    MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

    var year = _year.ToString(CultureInfo.InvariantCulture);
    MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
}

```

实现此接口的属性可以将 HTML 属性添加到生成的字段。检查 `ReleaseDate` 元素的输出时，将显示与上一示例类似的 HTML，唯一不同的是，此示例包含一个已在 `IClientModelValidator` 的 `AddValidation` 方法中定义的 `data-val-classicmovie` 属性。

```

<input class="form-control" type="datetime"
      data-val="true"
      data-val-classicmovie="Classic movies must have a release year earlier than 1960."
      data-val-classicmovie-year="1960"
      data-val-required="The ReleaseDate field is required."
      id="ReleaseDate" name="ReleaseDate" value="" />

```

非介入式验证使用 `data-` 属性中的数据来显示错误消息。不过，除非将规则或消息添加到 [jQuery](#) 的 `validator` 对象，否则 [jQuery](#) 并不知道它们的存在。以下示例对此进行了说明，该示例向 [jQuery](#) `validator` 对象添加一个包含自定义客户端验证代码的名为 `classicmovie` 的方法。可在此处查看对 `unobtrusive.adapters.add` 方法的说明

```

$(function () {
    $.validator.addMethod('classicmovie',
        function (value, element, params) {
            // Get element value. Classic genre has value '0'.
            var genre = $(params[0]).val(),
                year = params[1],
                date = new Date(value);
            if (genre && genre.length > 0 && genre[0] === '0') {
                // Since this is a classic movie, invalid if release date is after given year.
                return date.getFullYear() <= year;
            }

            return true;
        });
    $.validator.unobtrusive.adapters.add('classicmovie',
        ['year'],
        function (options) {
            var element = $(options.form).find('select#Genre')[0];
            options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
            options.messages['classicmovie'] = options.message;
        });
});

```

现在, jQuery 包含用于执行自定义 JavaScript 验证的信息, 以及该验证代码返回 `false` 时将要显示的错误消息。

远程验证

远程验证是一项非常不错的功能, 可在需要根据服务器上的数据验证客户端上的数据时使用。例如, 应用可能需要验证某个电子邮件或用户名是否已被使用, 并且它必须为此查询大量数据。为验证一个或几个字段而下载大量数据会占用过多资源。它还有可能暴露敏感信息。一种替代方法是发出往返请求来验证字段。

可以分两步实现远程验证。首先, 必须使用 `[Remote]` 属性为模型添加批注。`[Remote]` 属性采用多个重载, 可用于将客户端 JavaScript 定向到要调用的相应代码。下面的示例指向 `Users` 控制器的 `VerifyEmail` 操作方法。

```

[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; }

```

第二步是按照 `[Remote]` 属性中的定义, 将验证代码放入相应的操作方法。根据 jQuery Validate `remote()` 方法文档:

服务器端响应必须是使用默认错误消息的 JSON 字符串, 对于有效的元素必须为 `"true"`, 对于无效元素则可以为 `"false"`、`undefined` 或 `null`。如果服务器端响应是一个字符串, 例如, `"That name is already taken, try peter123 instead"`, 则此字符串将显示为自定义错误消息, 以取代默认错误消息。

`VerifyEmail()` 方法的定义遵循这些规则, 如下所示。如果电子邮件已被占用, 它会返回验证错误消息; 如果电子邮件可用, 则返回 `true`, 并将结果包装在 `JsonResult` 对象中。然后, 客户端可以使用返回的值, 继续进行下一步操作或根据需要显示错误。

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

现在，当用户输入电子邮件时，视图中的 JavaScript 会发出远程调用，以了解该电子邮件是否已被占用。如果是，则显示错误消息。如果不是，用户就可以像往常一样提交表单。

`[Remote]` 特性的 `AdditionalFields` 属性可用于根据服务器上的数据验证字段组合。例如，如果上面的 `User` 模型具有两个附加属性，名为 `FirstName` 和 `LastName`，你可能想要验证该名称对尚未被现有用户占用。按以下代码所示定义新属性：

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof.LastName)]
public string FirstName { get; set; }

[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof.FirstName)]
public string LastName { get; set; }
```

`AdditionalFields` 可能已显式设置为字符串 `"FirstName"` 和 `"LastName"`，但使用 `nameof` 这样的操作符可简化稍后的重构过程。然后，用于执行验证的操作方法必须采用两个参数，一个用于 `FirstName` 的值，一个用于 `LastName` 的值。

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userRepository.VerifyName(firstName, lastName))
    {
        return Json(data: $"A user named {firstName} {lastName} already exists.");
    }

    return Json(data: true);
}
```

现在，当用户输入名和姓时，JavaScript 会：

- 发出远程调用，以了解该名称对是否已被占用。
- 如果被占用，则显示一条错误消息。
- 如果未被占用，则用户可以提交表单。

如果需要使用 `[Remote]` 特性验证两个或更多附加字段，可将其以逗号分隔的列表形式列出。例如，若要向模型中添加 `MiddleName` 属性，可按以下代码所示设置 `[Remote]` 特性：

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof.FirstName + "," +
nameof.LastName)]
public string MiddleName { get; set; }
```

`AdditionalFields` 与所有属性参数一样，必须是常量表达式。因此，不能使用内插字符串或调用 `string.Join()` 来初始化 `AdditionalFields`。对于添加到 `[Remote]` 特性的每个附加字段，都必须向相应的控制器操作方法另外添加一个参数。

ASP.NET Core MVC 中的视图

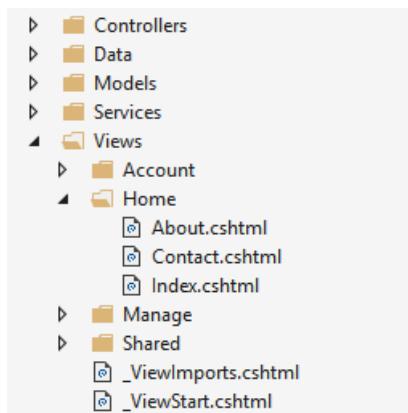
2018/5/14 • 15 min to read • [Edit Online](#)

作者 : Steve Smith 和 Luke Latham

本文档介绍在 ASP.NET Core MVC 应用程序中使用的视图。有关 Razor 页的信息，请参阅 [Razor 页简介](#)。

在“模型-视图-控制器(MVC)”模式中，视图处理应用的数据表示和用户交互。视图是嵌入了 [Razor 标记](#) 的 HTML 模板。Razor 标记一个代码，用于与 HTML 标记交互以生成发送给客户端的网页。

在 ASP.NET Core MVC 中，视图是在 Razor 标记中使用 [C# 编程语言](#) 的 .cshtml 文件。通常，视图文件会分组到以每个应用的 [控制器](#) 命名的文件夹中。此文件夹存储在应用根目录的“Views”文件夹中：



主页控制器由“Views”文件夹内的“Home”文件夹表示。“Home”文件夹包含“关于”、“联系人”和“索引”(主页)网页的视图。用户请求这三个网页中的一个时，主页控制器中的控制器操作决定使用三个视图中的哪一个来生成网页并将其返回给用户。

使用 [布局](#) 提供一致的网页部分并减少代码重复。布局通常包含页眉、导航和菜单元素以及页脚。页眉和页脚通常包含许多元数据元素的样板标记以及脚本和样式资产的链接。布局有助于在视图中避免这种样板标记。

[分部视图](#) 通过管理视图的可重用部分来减少代码重复。例如，分部视图可用于在多个视图中出现的博客网站上的作者简介。作者简介是普通的视图内容，不需要执行代码就能生成网页的内容。可以仅通过模型绑定查看作者简介内容，因此使用这种内容类型的分部视图是理想的选择。

[视图组件](#) 与分部视图的相似之处在于它们可以减少重复性代码，但视图组件还适用于需要在服务器上运行代码才能呈现网页的视图内容。呈现的内容需要数据库交互时(例如网站购物车)，视图组件非常有用。为了生成网页输出，视图组件不局限于模型绑定。

使用视图的好处

视图可帮助在 MVC 应用内建立 [关注点分离 \(SoC\)](#) 设计，方法是分隔用户界面标记与应用的其他部分。采用 SoC 设计可使应用模块化，从而提供以下几个好处：

- 应用组织地更好，因此更易于维护。视图一般按应用功能进行分组。这使得在处理功能时更容易找到相关的视图。
- 应用的若干部分是松散耦合的。可以生成和更新独立于业务逻辑和数据访问组件的应用视图。可以修改应用的视图，而不必更新应用的其他部分。
- 因为视图是独立的单元，所以更容易测试应用的用户界面部分。
- 由于应用组织地更好，因此你不太可能会意外重复用户界面的各个部分。

创建视图

在 Views / [ControllerName] 文件夹中创建特定于控制器的视图。控制器之间共享的视图都将置于 Views/Shared 文件夹。要创建一个视图，请添加新文件，并将其命名为与 .cshtml 文件扩展名相关联的控制器操作的相同名称。要创建与主页控制器中 About 操作相对应的视图，请在 Views/Home 文件夹中创建一个 About.cshtml 文件：

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor 标记以 @ 符号开头。通过将 C# 代码放置在用大括号 ({ ... }) 括住的 Razor 代码块内，运行 C# 语句。有关示例，请参阅上面显示的“About”到 ViewData["Title"] 的分配。只需用 @ 符号来引用值，即可在 HTML 中显示这些值。请参阅上面的 <h2> 和 <h3> 元素的内容。

以上所示的视图内容只是呈现给用户的整个网页中的一部分。其他视图文件中指定了页面布局的其余部分和视图的其他常见方面。要了解详细信息，请参阅[布局主题](#)。

控制器如何指定视图

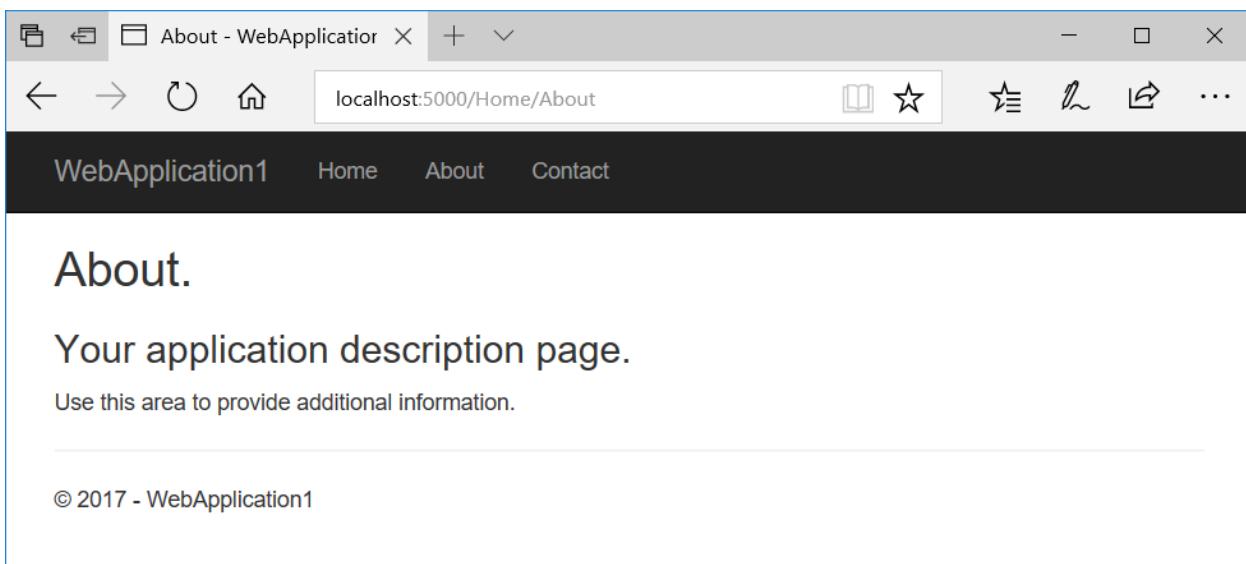
视图通常以 [ViewResult](#) 的形式从操作返回，这是一种 [ActionResult](#)。操作方法可以直接创建并返回 [ViewResult](#)，但通常不会这样做。由于大多数控制器均继承自 [控制器](#)，因此只需使用 [View](#) 帮助程序方法即可返回 [ViewResult](#)：

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

此操作返回时，最后一节显示的 About.cshtml 视图呈现为以下网页：



[View](#) 帮助程序方法有多个重载。可选择指定：

- 要返回的显式视图：

```
return View("Orders");
```

- 要传递给视图的模型：

```
return View(Orders);
```

- 视图和模型：

```
return View("Orders", Orders);
```

视图发现

操作返回一个视图时，会发生称为“视图发现”的过程。此过程基于视图名称确定使用哪个视图文件。

`View` 方法的默认行为 (`return View();`) 旨在返回与其从中调用的操作方法同名的视图。例如，控制器的 `About ActionResult` 方法名称用于搜索名为 `About.cshtml` 的视图文件。运行时首先在 `Views/[ControllerName]` 文件夹中搜索该视图。如果在此处找不到匹配的视图，则会在“Shared”文件夹中搜索该视图。

用 `return View();` 隐式返回 `ViewResult` 还是用 `return View("<ViewName>");` 将视图名称显式传递给 `View` 方法并不重要。在这两种情况下，视图发现都会按以下顺序搜索匹配的视图文件：

1. `Views/[ControllerName]/[ViewName].cshtml`
2. `Views/Shared/[ViewName].cshtml`

可以提供视图文件路径而不提供视图名称。如果使用从应用根目录开始的绝对路径（可选择以“/”或“/”开头），则须指定 `.cshtml` 扩展名：

```
return View("Views/Home/About.cshtml");
```

也可使用相对路径在不同目录中指定视图，而无需指定 `.cshtml` 扩展名。在 `HomeController` 内，可以使用相对路径返回 `Manage` 视图的 `Index` 视图：

```
return View("../Manage/Index");
```

同样，可以用“.”前缀来指示当前的控制器特定目录：

```
return View("./About");
```

[分部视图](#) 和 [视图组件](#) 使用类似（但不完全相同）的发现机制。

可以使用自定义 `IViewLocationExpander` 自定义如何在应用中定位视图的默认约定。

视图发现依赖于按文件名称查找视图文件。如果基础文件系统区分大小写，则视图名称也可能区分大小写。为了各操作系统的兼容性，请在控制器与操作名称之间，关联视图文件夹与文件名称之间匹配大小写。如果在处理区分大小写的文件系统时遇到无法找到视图文件的错误，请确认请求的视图文件与实际视图文件名称之间的大小写是否匹配。

按照组织视图文件结构的最佳做法，反映控制器、操作和视图之间的关系，实现可维护性和清晰度。

向视图传递数据

可使用多种方法将数据传递给视图。最可靠的方法是在视图中指定 **模型** 类型。此模型通常称为 **viewmodel**。将 **.viewmodel** 类型的实例传递给此操作的视图。

使用 **viewmodel** 将数据传递给视图可让视图充分利用强类型检查。强类型化(或强类型)意味着每个变量和常量都有明确定义的类型(例如 `string`、`int` 或 `DateTime`)。在编译时检查视图中使用的类型是否有效。

Visual Studio 和 **Visual Studio Code** 列出了使用 **IntelliSense** 功能的强类型类成员。如果要查看 **viewmodel** 的属性, 请键入 **viewmodel** 的变量名称, 后跟句点 (`.`)。这有助于提高编写代码的速度并降低错误率。

使用 `@model` 指令指定模型。使用带有 `@Model` 的模型:

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

为了将模型提供给视图, 控制器将其作为参数进行传递:

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

没有针对可以提供给视图的模型类型的限制。建议使用普通旧 CLR 对象 (POCO) **viewmodel**, 它几乎没有已定义的行为(方法)。通常, **viewmodel** 类要么存储在“Models”文件夹中, 要么存储在应用根目录处的单独“ViewModels”文件夹中。上例中使用的 **Address** **viewmodel** 是存储在 **Address.cs** 文件中的 POCO **viewmodel**:

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

注意

可随意对 `viewmodel` 类型和业务模型类型使用相同的类。但是，使用单独的模型可使视图独立于应用的业务逻辑和数据访问部分。模型为用户发送给应用的数据使用[模型绑定](#)和[验证](#)时，模型和 `viewmodel` 的分离也会提供安全优势。

弱类型数据 (`ViewData` 和 `ViewBag`)

注意：`ViewBag` 在 Razor 页中不可用。

除了强类型视图，视图还可以访问弱类型(也称为松散类型)的数据集合。与强类型不同，弱类型(或松散类型)意味着不显式声明要使用的数据类型。可以使用弱类型数据的集合将少量数据传入及传出控制器和视图。

传递数据于 ...	示例
控制器和视图	用数据填充下拉列表。
视图和 布局视图	从视图文件设置布局视图中的 <code>< title></code> 元素内容。
分部视图 和视图	基于用户请求的网页显示数据的小组件。

可以通过控制器和视图上的 `ViewData` 或 `ViewBag` 属性来引用此集合。`ViewData` 属性是弱类型对象的字典。

`ViewBag` 属性是 `ViewData` 的包装器，为基础 `ViewData` 集合提供动态属性。

`ViewData` 和 `ViewBag` 在运行时进行动态解析。由于它们不提供编译时类型检查，因此使用这两者通常比使用 `viewmodel` 更容易出错。出于上述原因，一些开发者希望尽量减少或根本不使用 `ViewData` 和 `ViewBag`。

`ViewData`

`ViewData` 是通过 `string` 键访问的 [ViewDataDictionary](#) 对象。字符串数据可以直接存储和使用，而不需要强制转换，但是在提取其他 `ViewData` 对象值时必须将其强制转换为特定类型。可以使用 `ViewData` 将数据从控制器传递到视图，以及在视图(包括[分部视图](#)和[布局](#))内传递数据。

以下是在操作中使用 `ViewData` 设置问候语和地址值的示例：

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"] = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

在视图中处理数据：

```

@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>

```

ViewBag

注意: `ViewBag` 在 Razor 页中不可用。

`ViewBag` 是 `DynamicViewData` 对象, 可提供对存储在 `ViewData` 中的对象的动态访问。`ViewBag` 不需要强制转换, 因此使用起来更加方便。下例演示如何使用与上述 `ViewData` 有相同结果的 `ViewBag`:

```

public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}

```

```

@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>

```

同时使用 `ViewData` 和 `ViewBag`

注意: `ViewBag` 在 Razor 页中不可用。

由于 `ViewData` 和 `ViewBag` 引用相同的基础 `ViewData` 集合, 因此在读取和写入值时, 可以同时使用 `ViewData` 和 `ViewBag`, 并在两者之间进行混合和匹配。

在 `About.cshtml` 视图顶部, 使用 `ViewBag` 设置标题并使用 `ViewData` 设置说明:

```

@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy and mission.";
}

```

读取属性, 但反向使用 `ViewData` 和 `ViewBag`。在 `_Layout.cshtml` 文件中, 使用 `ViewData` 获取标题并使用

`ViewBag` 获取说明：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...

```

请记住，字符串不需要为 `ViewData` 进行强制转换。可以使用 `@ViewData["Title"]` 而不需要强制转换。

可同时使用 `ViewData` 和 `ViewBag` 也可混合和匹配读取及写入属性。呈现以下标记：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About Contoso</title>
    <meta name="description" content="Let us tell you about Contoso's philosophy and mission.">
    ...

```

ViewData 和 ViewBag 之间差异的摘要

`ViewBag` 在 Razor 页中不可用。

- `ViewData`
 - 派生自 `ViewDataDictionary`，因此它有可用的字典属性，如 `ContainsKey`、`Add`、`Remove` 和 `Clear`。
 - 字典中的键是字符串，因此允许有空格。示例：`ViewData["Some Key With Whitespace"]`
 - 任何非 `string` 类型均须在视图中进行强制转换才能使用 `ViewData`。
- `ViewBag`
 - 派生自 `Dynamic ViewData`，因此它可使用点表示法 (`@ViewBag.SomeKey = <value or object>`) 创建动态属性，且无需强制转换。`ViewBag` 的语法使添加到控制器和视图的速度更快。
 - 更易于检查 NULL 值。示例：`@ViewBag.Person?.Name`

何时使用 ViewData 或 ViewBag

`ViewData` 和 `ViewBag` 都是在控制器和视图之间传递少量数据的有效方法。根据偏好选择使用哪种方法。可以混合和匹配 `ViewData` 和 `ViewBag` 对象，但是，使用一致的方法可以更轻松地读取和维护代码。这两种方法都是在运行时进行动态解析的，因此容易造成运行时错误。因而，一些开发团队会避免使用它们。

动态视图

不使用 `@model` 声明模型类型，但有模型实例传递给它们的视图（如 `return View(Address);`）可动态引用实例的属性：

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

此功能提供了灵活性，但不提供编译保护或 IntelliSense。如果属性不存在，则网页生成在运行时会失败。

更多视图功能

[标记帮助程序](#) 可以轻松地将服务器端行为添加到现有的 HTML 标记。使用标记帮助程序可避免在视图内编写自定义代码或帮助程序。标记帮助程序作为属性应用于 HTML 元素，并被无法处理它们的编辑器忽略。这可让你

在各种工具中编辑和呈现视图标记。

通过许多内置 HTML 帮助程序可生成自定义 HTML 标记。通过[视图组件](#)可以处理更复杂的用户界面逻辑。视图组件提供的 SoC 与控制器和视图提供的相同。它们无需使用处理数据(由常见用户界面元素使用)的操作和视图。

与 ASP.NET Core 的许多其他方面一样，视图支持[依赖关系注入](#)，可将服务注入视图。

ASP.NET Core 的 Razor 语法参考

2018/5/17 • 14 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Luke Latham](#)、[Taylor Mullen](#) 和 [Dan Vicarel](#)

Razor 是一种标记语法，用于将基于服务器的代码嵌入网页中。Razor 语法由 Razor 标记、C# 和 HTML 组成。包含 Razor 的文件通常具有 `.cshtml` 文件扩展名。

呈现 HTML

默认 Razor 语言为 HTML。从 Razor 标记呈现 HTML 与从 HTML 文件呈现 HTML 并没有什么不同。服务器会按原样呈现 `.cshtml` Razor 文件中的 HTML 标记。

Razor 语法

Razor 支持 C#，并使用 `@` 符号从 HTML 转换为 C#。Razor 计算 C# 表达式，并将它们呈现在 HTML 输出中。

当 `@` 符号后跟 [Razor 保留关键字](#) 时，它会转换为 Razor 特定标记。否则会转换为纯 C#。

若要对 Razor 标记中的 `@` 符号进行转义，请使用另一个 `@` 符号：

```
<p>@@Username</p>
```

该代码在 HTML 中使用单个 `@` 符号呈现：

```
<p>@Username</p>
```

包含电子邮件地址的 HTML 属性和内容不将 `@` 符号视为转换字符。Razor 分析不会处理以下示例中的电子邮件地址：

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

隐式 Razor 表达式

隐式 Razor 表达式以 `@` 开头，后跟 C# 代码：

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

隐式表达式不能包含空格，但 C# `await` 关键字除外。如果该 C# 语句具有明确的结束标记，则可以混用空格：

```
<p>@await DoSomething("hello", "world")</p>
```

隐式表达式不能包含 C# 泛型，因为括号 (`<>`) 内的字符会被解释为 HTML 标记。以下代码无效：

```
<p>@GenericMethod<int>()</p>
```

上述代码生成与以下错误之一类似的编译器错误：

- "int" 元素未结束。所有元素都必须自结束或具有匹配的结束标记。
- 无法将方法组 "GenericMethod" 转换为非委托类型 "object"。是否希望调用此方法？

泛型方法调用必须包装在显式 Razor 表达式或 Razor 代码块中。

显式 Razor 表达式

显式 Razor 表达式由 @ 符号和平衡圆括号组成。若要呈现上一周的时间，可使用以下 Razor 标记：

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

将计算 @() 括号中的所有内容，并将其呈现到输出中。

前面部分中所述的隐式表达式通常不能包含空格。在下面的代码中，不会从当前时间减去一周：

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

该代码呈现以下 HTML：

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

可以使用显式表达式将文本与表达式结果串联起来：

```
@{  
    var joe = new Person("Joe", 33);  
}  
  
<p>Age@{(joe.Age)}</p>
```

如果不使用显式表达式，`<p>Age@joe.Age</p>` 会被视为电子邮件地址，因此会呈现 `<p>Age@joe.Age</p>`。如果编写为显式表达式，则呈现 `<p>Age33</p>`。

显式表达式可用于从 `.cshtml` 文件中的泛型方法呈现输出。以下标记显示了如何更正之前出现的由 C# 泛型的括号引起的错误。此代码以显式表达式的形式编写：

```
<p>@(GenericMethod<int>())</p>
```

表达式编码

计算结果为字符串的 C# 表达式采用 HTML 编码。计算结果为 `IHtmlContent` 的 C# 表达式直接通过 `IHtmlContent.WriteTo` 呈现。计算结果不为 `IHtmlContent` 的 C# 表达式通过 `ToString` 转换为字符串，并在呈现前进行编码。

```
@("<span>Hello World</span>")
```

该代码呈现以下 HTML：

```
&lt;span&gt;Hello World&lt;/span&gt;
```

该 HTML 在浏览器中显示为：

```
<span>Hello World</span>
```

`HtmlHelper.Raw` 输出不进行编码，但呈现为 HTML 标记。

警告

对未经审查的用户输入使用 `HtmlHelper.Raw` 会带来安全风险。用户输入可能包含恶意的 JavaScript 或其他攻击。审查用户输入比较困难。应避免对用户输入使用 `HtmlHelper.Raw`。

```
@Html.Raw("<span>Hello World</span>")
```

该代码呈现以下 HTML：

```
<span>Hello World</span>
```

Razor 代码块

Razor 代码块以 `@` 开头，并括在 `{}` 中。代码块内的 C# 代码不会呈现，这点与表达式不同。一个视图中的代码块和表达式共享相同的作用域并按顺序进行定义：

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

该代码呈现以下 HTML：

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

隐式转换

代码块中的默认语言为 C#，不过，Razor 页面可以转换回 HTML：

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

带分隔符的显式转换

若要定义应呈现 HTML 的代码块子节，请使用 Razor `<text>` 标记将要呈现的字符括起来：

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

使用此方法可呈现未被 HTML 标记括起来的 HTML。如果没有 HTML 或 Razor 标记，会发生 Razor 运行时错误。

<text> 标记可用于在呈现内容时控制空格：

- 仅呈现 <text> 标记之间的内容。
- <text> 标记之前或之后的空格不会显示在 HTML 输出中。

使用 @ 的显式行转换：

若要在代码块内以 HTML 的形式呈现整个行的其余内容，请使用 @: 语法：

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

如果代码中没有 @:，会生成 Razor 运行时错误。

警告：Razor 文件中多余的 @ 字符可能会导致代码块中后面的语句发生编译器错误。这些编译器错误可能难以理解，因为实际错误发生在报告的错误之前。将多个隐式/显式表达式合并到单个代码块以后，经常会发生此错误。

控制结构

控制结构是对代码块的扩展。代码块的各个方面(转换为标记、内联 C#)同样适用于以下结构：

条件语句 @if、else if、else 和 @switch

@if 控制何时运行代码：

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

else 和 else if 不需要 @ 符号：

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

以下标记展示如何使用 switch 语句：

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

循环语句 @for、@foreach、@while 和 @do while

可以使用循环控制语句呈现模板化 HTML。若要呈现一组人员：

```
@{
    var people = new Person[]
    {
        new Person("Weston", 33),
        new Person("Johnathon", 41),
        ...
    };
}
```

支持以下循环语句：

`@for`

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

复合语句 @using

在 C# 中，`using` 语句用于确保释放对象。在 Razor 中，可使用相同的机制来创建包含附加内容的 HTML 帮助程序。在下面的代码中，HTML 帮助程序使用 `@using` 语句呈现表单标记：

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

可以使用[标记帮助程序](#)执行作用域级别的操作。

@try、catch、finally

异常处理与 C# 类似：

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

@lock

Razor 可以使用 `lock` 语句来保护关键节：

```
@lock (SomeLock)
{
    // Do critical section work
}
```

注释

Razor 支持 C# 和 HTML 注释：

```
@{  
    /* C# comment */  
    // Another C# comment  
}  
<!-- HTML comment -->
```

该代码呈现以下 HTML：

```
<!-- HTML comment -->
```

在呈现网页之前，服务器会删除 Razor 注释。Razor 使用 `@* *@` 来分隔注释。以下代码已被注释禁止，因此服务器不呈现任何标记：

```
@*  
{@  
    /* C# comment */  
    // Another C# comment  
}  
<!-- HTML comment -->  
*@
```

指令

Razor 指令由隐式表达式表示：`@` 符号后跟保留关键字。指令通常用于更改视图分析方式或启用不同的功能。

通过了解 Razor 如何为视图生成代码，更易理解指令的工作原理。

```
@{  
    var quote = "Getting old ain't for wimps! - Anonymous";  
}  
  
<div>Quote of the Day: @quote</div>
```

该代码生成与下面类似的类：

```
public class _Views_Something_cshtml : RazorPage<dynamic>  
{  
    public override async Task ExecuteAsync()  
    {  
        var output = "Getting old ain't for wimps! - Anonymous";  
  
        WriteLiteral("/r/n<div>Quote of the Day: ");  
        Write(output);  
        WriteLiteral("</div>");  
    }  
}
```

本文后面的[查看为视图生成的 Razor C# 类](#)部分说明了如何查看此生成的类。

@using

`@using` 指令用于向生成的视图添加 C# `using` 指令：

```
@using System.IO  
{@  
    var dir = Directory.GetCurrentDirectory();  
}  
<p>@dir</p>
```

@model

@model 指令指定传递到视图的模型类型：

```
@model TypeNameOfModel
```

在使用个人用户帐户创建的 ASP.NET Core MVC 应用中，Views/Account/Login.cshtml 视图包含以下模型声明：

```
@model LoginViewModel
```

生成的类继承自 RazorPage<dynamic>：

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor 公开了 Model 属性，用于访问传递到视图的模型：

```
<div>The Login Email: @Model.Email</div>
```

@model 指令指定此属性的类型。该指令将 RazorPage<T> 中的 T 指定为生成的类，视图便派生自该类。如果未指定 @model 指令，则 Model 属性的类型为 dynamic。模型的值会从控制器传递到视图。有关详细信息，请参阅 [强类型模型和@模型关键字](#)。

@inherits

@inherits 指令对视图继承的类提供完全控制：

```
@inherits TypeNameOfClassToInheritFrom
```

下面的代码是一种自定义 Razor 页面类型：

```
using Microsoft.AspNetCore.Mvc.Razor;  
  
public abstract class CustomRazorPage<TModel> : RazorPage<TModel>  
{  
    public string CustomText { get; } = "Gardyloo! - A Scottish warning yelled from a window before dumping  
    a slop bucket on the street below.";  
}
```

CustomText 显示在视图中：

```
@inherits CustomRazorPage<TModel>  
  
<div>Custom text: @CustomText</div>
```

该代码呈现以下 HTML：

```
<div>Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

`@model` 和 `@inherits` 可在同一视图中使用。`@inherits` 可位于视图导入的 `_ViewImports.cshtml` 文件中：

```
@inherits CustomRazorPage<TModel>
```

下面的代码是一种强类型视图：

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

如果在模型中传递“`rick@contoso.com`”，视图将生成以下 HTML 标记：

```
<div>The Login Email: rick@contoso.com</div>
<div>Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping a slop bucket on the street below.</div>
```

@inject

`@inject` 指令允许 Razor 页面将服务从[服务容器](#)注入到视图。有关详细信息，请参阅[视图中的依赖关系注入](#)。

@functions

`@functions` 指令允许 Razor 页面将 C# 代码块添加到视图中：

```
@functions { // C# Code }
```

例如：

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

该代码生成以下 HTML 标记：

```
<div>From method: Hello</div>
```

以下代码是生成的 Razor C# 类：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998

```

@section

`@section` 指令与[布局](#)结合使用，允许视图将内容呈现在 HTML 页面的不同部分。有关详细信息，请参阅[部分](#)。

标记帮助程序

标记帮助程序有三个相关指令。

指令	函数
<code>@addTagHelper</code>	向视图提供标记帮助程序。
<code>@removeTagHelper</code>	从视图中删除以前添加的标记帮助程序。
<code>@tagHelperPrefix</code>	指定标记前缀，以后用标记帮助程序支持并阐明标记帮助程序的用法。

Razor 保留关键字

Razor 关键字

- `page`(需要 ASP.NET Core 2.0 及更高版本)
- `namespace`
- `functions`
- `inherits`
- `model`
- `section`
- `helper`(ASP.NET Core 当前不支持)

Razor 关键字使用 `@(Razor Keyword)` 进行转义(例如, `@(functions)`)。

C# Razor 关键字

- `case`
- `do`
- `default`
- `for`
- `foreach`

- if
- else
- lock
- switch
- try
- catch
- finally
- using
- while

C# Razor 关键字必须使用 `@@C# Razor Keyword` 进行双转义(例如, `@@case`)。第一个 `@` 对 Razor 分析器转义。第二个 `@` 对 C# 分析器转义。

Razor 不使用的保留关键字

- 类

查看为视图生成的 Razor C# 类

将下面的类添加到 ASP.NET Core MVC 项目：

```
using Microsoft.AspNetCore.Mvc.Razor.Extensions;
using Microsoft.AspNetCore.Razor.Language;

public class CustomTemplateEngine : MvcRazorTemplateEngine
{
    public CustomTemplateEngine(RazorEngine engine, RazorProject project)
        : base(engine, project)
    {
    }

    public override RazorCSharpDocument GenerateCode(RazorCodeDocument codeDocument)
    {
        var csharpDocument = base.GenerateCode(codeDocument);
        var generatedCode = csharpDocument.GeneratedCode;

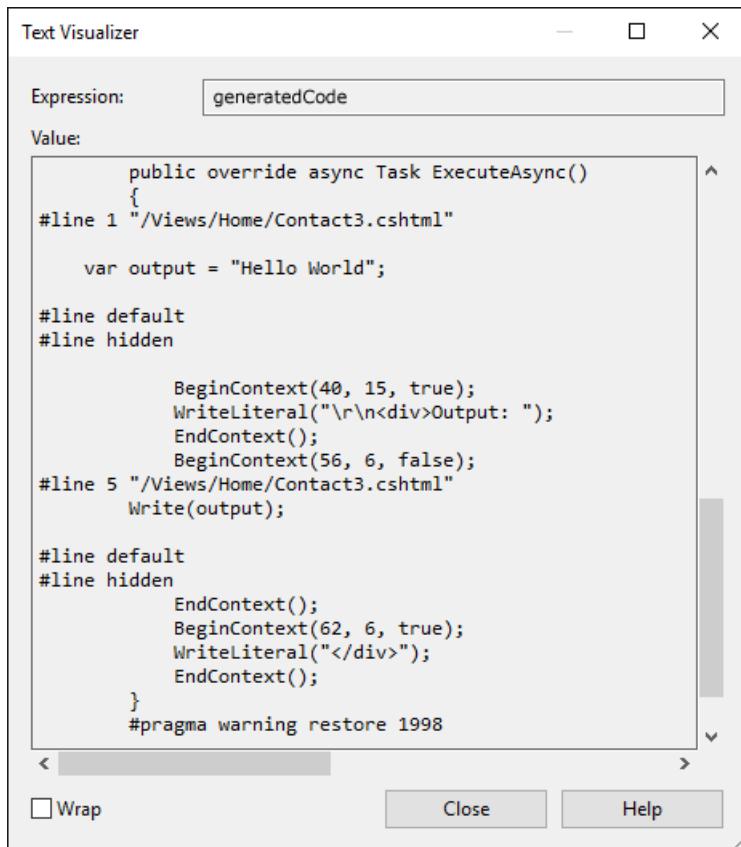
        // Look at generatedCode

        return csharpDocument;
    }
}
```

使用 `CustomTemplateEngine` 类替代 MVC 添加的 `RazorTemplateEngine`：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<RazorTemplateEngine, CustomTemplateEngine>();
}
```

在 `CustomTemplateEngine` 的 `return csharpDocument` 语句上设置断点。当程序执行在断点处停止时，查看 `generatedCode` 的值。



视图查找和区分大小写

Razor 视图引擎为视图执行区分大小写的查找。但是，实际查找取决于基础文件系统：

- 基于文件的源：
 - 在使用不区分大小写的文件系统的操作系统(例如, Windows)上, 物理文件提供程序查找不区分大小写。例如, `return View("Test")` 可匹配 `/Views/Home/Test.cshtml`、`/Views/home/test.cshtml` 以及任何其他大小写变体。
 - 在区分大小写的文件系统(例如, Linux、OSX 以及使用 `EmbeddedFileProvider` 构建的文件系统)上, 查找区分大小写。例如, `return View("Test")` 专门匹配 `/Views/Home/Test.cshtml`。
- 预编译视图:在 ASP.NET Core 2.0 及更高版本中, 预编译视图查找在所有操作系统上均不区分大小写。该行为与 Windows 上物理文件提供程序的行为相同。如果两个预编译视图仅大小写不同, 则查找的结果具有不确定性。

建议开发人员将文件和目录名称的大小写与以下项的大小写匹配：

```
* <span data-ttu-id="c6e82-293">区域、控制器和操作名称。</span><span class="sxs-lookup"><span data-stu-id="c6e82-293">Area, controller, and action names.</span></span>
* <span data-ttu-id="c6e82-294">Razor 页面。</span><span class="sxs-lookup"><span data-stu-id="c6e82-294">Razor Pages.</span></span>
```

匹配大小写可确保无论使用哪种基础文件系统, 部署都能找到其视图。

ASP.NET Core 中的 Razor 视图编译和预编译

2018/5/4 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

调用视图时, Razor 视图在运行时进行编译。ASP.NET Core 1.1.0 及更高版本可以选择性地编译 Razor 视图, 并将其与应用一起部署—这个过程称为预编译。ASP.NET Core 2.x 项目模板默认启用预编译。

重要事项

在 ASP.NET Core 2.0 中执行独立部署 (SCD) 时, Razor 视图预编译当前不可用。2.1 版本发布后, 该功能将可用于 SCD。有关详细信息, 请参阅 [View compilation fails when cross-compiling for Linux on Windows](#)(对 Windows 上的 Linux 进行交叉编译时, 视图编译失败)。

预编译注意事项:

- 预编译视图生成的发布捆绑包更小, 启动速度更快。
- 在预编译视图之后, 无法编辑 Razor 文件。已编辑的视图不会出现在发布捆绑包中。

部署预编译的视图:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果项目面向 .NET Framework, 请添加对 [Microsoft.AspNetCore.Mvc.Razor.ViewCompilation](#) 的包引用:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0"  
PrivateAssets="All" />
```

如果项目面向 .NET Core, 则无需进行任何更改。

默认情况下, ASP.NET Core 2.x 项目模板将 `MvcRazorCompileOnPublish` 隐式设置为 `true`, 这意味着可以从 .csproj 文件安全地删除此节点。如果希望显式设置, 那么将 `MvcRazorCompileOnPublish` 属性设置为 `true` 并没有什么坏处。以下 .csproj 示例突出显示了此设置:

```
<Project Sdk="Microsoft.NET.Sdk.Web">  
  
  <PropertyGroup>  
    <TargetFramework>netcoreapp2.0</TargetFramework>  
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>  
  </PropertyGroup>  
  
  <ItemGroup>  
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />  
  </ItemGroup>  
  
</Project>
```

ASP.NET Core 中的布局

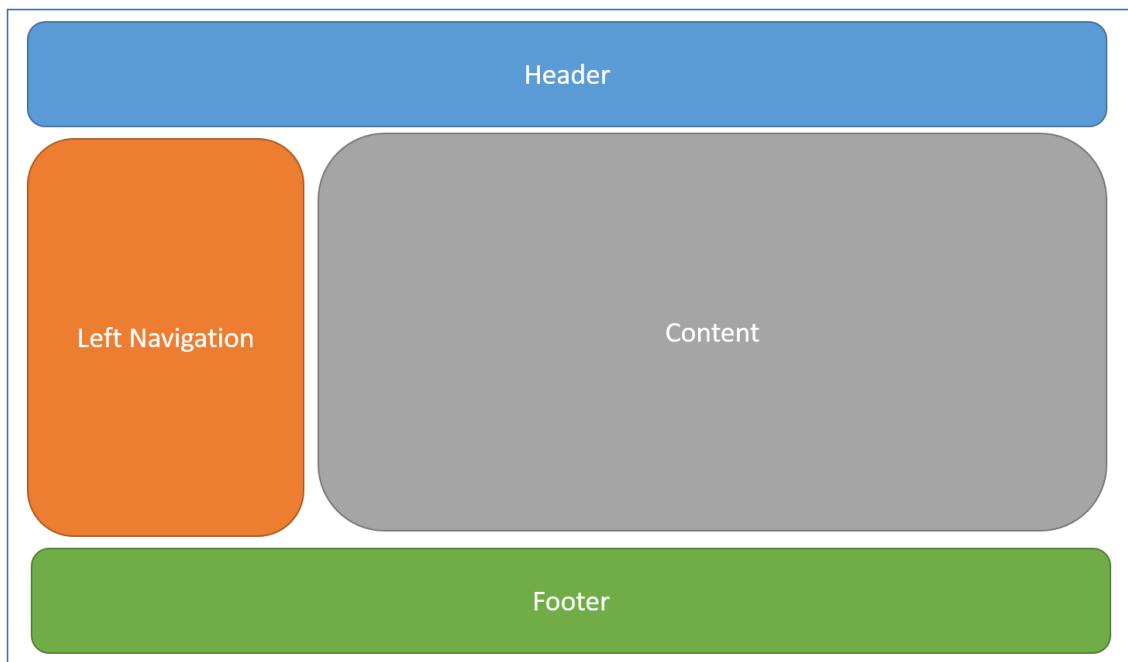
2018/5/14 • 6 min to read • [Edit Online](#)

作者: Steve Smith

视图经常共享可视和编程元素。本文介绍如何在 ASP.NET 应用中呈现视图之前，使用通用布局、共享指令和运行常见代码。

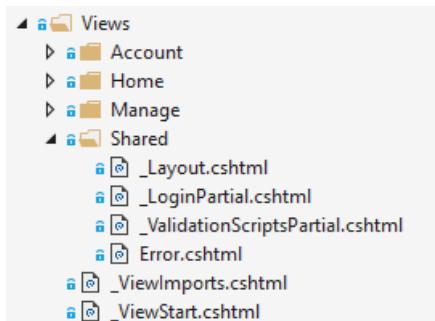
什么是布局

大多数 Web 应用都有一个通用布局，可在页面间切换时为用户提供一致体验。该布局通常包括应用标头、导航或菜单元素以及页脚等常见的用户界面元素。



应用中的许多页面也经常使用脚本和样式表等常用的 HTML 结构。所有这些共享元素均可在布局文件中进行定义，应用内使用的任何视图随后均可引用此文件。布局可减少视图中的重复代码，帮助它们遵循**不要自我重复 (DRY)** 原则。

按照约定，ASP.NET 应用的默认布局名为 `_Layout.cshtml`。Visual Studio ASP.NET Core MVC 项目模板在 `Views/Shared` 文件夹中包含此布局文件：



此布局为应用中的视图定义顶级模板。应用不需要布局，它们可以定义多个布局，并且不同的视图指定不同的布局。

示例 `_Layout.cshtml` :

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">WebApplication1</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
                @await Html.PartialAsync("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2016 - WebApplication1</p>
        </footer>
    </div>

    <environment names="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment names="Staging,Production">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
               asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
               asp-fallback-test="window.jQuery">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
               asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
               asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

```

```
@RenderSection("scripts", required: false)
</body>
</html>
```

指定布局

Razor 视图具有 `Layout` 属性。单个视图通过设置此属性来指定布局：

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

指定的布局可以使用完整路径（例如：`/Views/Shared/_Layout.cshtml`）或部分名称（例如：`_Layout`）。如果提供部分名称，Razor 视图引擎将使用其标准发现过程来搜索布局文件。首先搜索与控制器关联的文件夹，然后搜索 `Shared` 文件夹。此发现过程与用于发现[分部视图](#)的过程相同。

默认情况下，每个布局必须调用 `RenderBody`。无论在何处调用 `RenderBody`，都会呈现视图的内容。

部分

布局可以通过调用 `RenderSection` 来选择引用一个或多个节。节提供一种方法来组织某些页面元素应当放置的位置。对 `RenderSection` 的每次调用均可指定该节是必需的还是可选的。如果找不到所需的节，则会引发异常。单个视图使用 `@section` Razor 语法指定要在节中呈现的内容。如果视图定义某个节，则必须呈现该节（否则会发生错误）。

视图中的示例 `@section` 定义：

```
@section Scripts {
    <script type="text/javascript" src="/scripts/main.js"></script>
}
```

在上面的代码中，验证脚本添加到了视图（包含窗体）上的 `scripts` 节。同一应用程序中的其他视图可能不需要任何其他脚本，因此无需定义脚本节。

视图中定义的节仅在其即时布局页面中可用。不能从部分、视图组件或视图系统的其他部分引用它们。

忽略节

默认情况下，必须由布局页面呈现内容页中的正文和所有节。Razor 视图引擎通过跟踪是否已呈现正文和每个节来强制执行此操作。

要让视图引擎忽略正文或节，请调用 `IgnoreBody` 和 `IgnoreSection` 方法。

必须呈现或忽略 Razor 页面中的正文和每个节。

导入共享指令

视图可以使用 Razor 指令来执行许多操作，例如导入命名空间或执行[依赖关系注入](#)。可在[一个共同的 `_ViewImports.cshtml` 文件](#)中指定由许多视图共享的指令。`_ViewImports` 文件支持以下指令：

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`

- `@model`
- `@inherits`
- `@inject`

该文件不支持函数和节定义等其他 Razor 功能。

示例 `_ViewImports.cshtml` 文件：

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

针对 ASP.NET Core MVC 应用的 `_ViewImports.cshtml` 文件通常放置在 `Views` 文件夹中。

`_ViewImports.cshtml` 文件可以放置在任何文件夹中，在这种情况下，它仅适用于该文件夹及其子文件夹中的视图。由于从根级别开始处理 `_ViewImports` 文件，然后处理视图本身的位置之前的每个文件夹，因此在根级别指定的设置可能会覆盖在文件夹级别。

例如，如果根级别 `_ViewImports.cshtml` 文件指定 `@model` 和 `@addTagHelper`，在该视图的控制器关联文件夹中，另一个 `_ViewImports.cshtml` 文件指定不同的 `@model` 并添加另一个 `@addTagHelper`，该视图将有权访问这两个标记帮助程序，并将使用后一个 `@model`。

如果对一个视图运行多个 `_ViewImports.cshtml` 文件，那么包含在 `ViewImports.cshtml` 文件中的指令的组合行为如下所示：

- `@addTagHelper` `@removeTagHelper`：按顺序全部运行
- `@tagHelperPrefix`：最接近视图的文件会替代任何其他文件
- `@model`：最接近视图的文件会替代任何其他文件
- `@inherits`：最接近视图的文件会替代任何其他文件
- `@using`：全部包括在内；忽略重复项
- `@inject`：针对每个属性，最接近视图的属性会替代具有相同属性名的任何其他属性

在呈现每个视图之前运行代码

如果有需要在呈现每个视图之前运行的代码，应将其置于 `_ViewStart.cshtml` 文件中。按照约定，`_ViewStart.cshtml` 文件位于 `Views` 文件夹中。在呈现每个完整视图（不是布局，也不是分部视图）之前运行 `_ViewStart.cshtml` 中列出的语句。与 `ViewImports.cshtml` 一样，`_ViewStart.cshtml` 是分层的。如果在控制器关联的视图文件夹中定义了 `_ViewStart.cshtml` 文件，则将在 `Views` 文件夹根目录中定义的文件（如有）之后运行该文件。

示例 `_ViewStart.cshtml` 文件：

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

上述文件指定所有视图都将使用 `_Layout.cshtml` 布局。

注意

`_ViewStart.cshtml` 和 `_ViewImports.cshtml` 通常不会放置在 `/Views/Shared` 文件夹中。这些应用级别版本的文件应直接放置在 `/Views` 文件夹中。

ASP.NET Core 中的标记帮助程序

2018/5/14 • 14 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

什么是标记帮助程序？

标记帮助程序使服务器端代码可以在 Razor 文件中参与创建和呈现 HTML 元素。例如，内置 `ImageTagHelper` 可以将版本号追加到映像名称。每当映像发生变化时，服务器都会为映像生成一个新的唯一版本，因此客户端总能获得当前映像（而不是过时的缓存映像）。有多种常见任务（例如创建窗体、链接、加载资产等）的内置标记帮助程序，公共 GitHub 存储库和 NuGet 包中甚至还有更多可用标记帮助程序。标记帮助程序使用 C# 创建，基于元素名称、属性名称或父标记以 HTML 元素为目标。例如，应用 `LabelTagHelper` 属性时，内置 `LabelTagHelper` 可以 HTML `<label>` 元素为目标。如果熟悉 [HTML 帮助程序](#)，则标记帮助程序将减少 Razor 视图中 HTML 和 C# 之间的显式转换。在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。[标记帮助程序与 HTML 帮助程序的比较](#)更详细地介绍了两者之间的差异。

标记帮助程序的功能

HTML 友好开发体验 在多数情况下，使用标记帮助程序的 Razor 标记看起来像是标准 HTML。熟悉 HTML/CSS/JavaScript 的前端设计师，无需学习 C# Razor 语法即可编辑 Razor。

用于创建 HTML 和 Razor 标记的丰富 IntelliSense 环境 这与 HTML 帮助程序形成鲜明对比，HTML 帮助程序是 Razor 视图中标记的曾用服务器端创建方法。[标记帮助程序与 HTML 帮助程序的比较](#)更详细地介绍了两者之间的差异。[标记帮助程序的 IntelliSense 支持](#)解释了 IntelliSense 环境。即使是熟悉 Razor C# 语法的开发人员，使用标记帮助程序也比编写 C# Razor 标记更高效。

使用仅在服务器上可用的信息，可提高生产力，并能生成更稳定、可靠和可维护的代码 例如，过去更新映像时，必须在更改映像时更改映像名称。出于性能原因，要主动缓存映像，而若不更改映像的名称，客户端就可能获得过时的副本。以前，编辑完映像后，必须更改名称，而且需要更新 Web 应用中对该映像的每个引用。这不仅大费周章，还容易出错（可能会漏掉某个引用、意外输入错误的字符串等等）内置 `ImageTagHelper` 可自动执行此操作。`ImageTagHelper` 可将版本号追加到映像名称，这样每当映像出现更改时，服务器都会自动为该映像生成新的唯一版本。客户端总是能获得最新映像。使用 `ImageTagHelper` 实质上是免费获得稳健性而节省劳动力。

大多数内置标记帮助程序以标准 HTML 元素为目标，为该元素提供服务器端属性。例如，`<input>` 用于包含 `asp-for` 特性的“视图/帐户”文件夹中的很多视图。此特性将指定模型属性的名称提取至所呈现的 HTML。以一个具备以下模型的 Razor 视图为例：

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

以下 Razor 标记：

```
<label asp-for="Movie.Title"></label>
```

则会生成以下 HTML：

```
<label for="Movie_Title">Title</label>
```

通过 [LabelTagHelper](#) 中的 `For` 属性，可使用 `asp-for` 特性。请参阅[创作标记帮助程序](#)，获取详细信息。

管理标记帮助程序作用域

标记帮助程序作用域由 `@addTagHelper`、`@removeTagHelper` 和“!”选择退出字符联合控制。

使用 `@addTagHelper` 添加标记帮助程序

如果创建名为 `AuthoringTagHelpers` 的新 ASP.NET Core Web 应用（无身份验证），将向项目添加以下 `Views/_ViewImports.cshtml` 文件：

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

`@addTagHelper` 指令让视图可以使用标记帮助程序。在此示例中，视图文件是 `Views/_ViewImports.cshtml`，“Views”文件夹及其子目录中的所有视图文件都会默认继承它，使得标记帮助程序可用。上面的代码使用通配符语法（“*”），指定程序集 (`Microsoft.AspNetCore.Mvc.TagHelpers`) 中的所有标记帮助程序对于 `Views` 目录或子目录中的所有视图文件可用。`@addTagHelper` 后第一个参数指定要加载的标记帮助程序（我们使用“*”指定加载所有标记帮助程序），第二个参数“`Microsoft.AspNetCore.Mvc.TagHelpers`”指定包含标记帮助程序的程序集。`Microsoft.AspNetCore.Mvc.TagHelpers` 是内置 ASP.NET Core 标记帮助程序的程序集。

要公开此项目中的所有标记帮助程序（将创建名为 `AuthoringTagHelpers` 的程序集），可使用以下内容：

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

如果项目包含具有默认命名空间（`AuthoringTagHelpers.TagHelpers.EmailTagHelper`）的 `EmailTagHelper`，则可提供标记帮助程序的完全限定名称（FQN）：

```
@using AuthoringTagHelpers  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers  
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

若要使用 FQN 将标记帮助程序添加到视图，请首先添加 FQN（

`AuthoringTagHelpers.TagHelpers.EmailTagHelper`），然后添加程序集名称（`AuthoringTagHelpers`）。大多数开发人员更喜欢使用“*”通配符语法。使用通配符语法，可在 FQN 中插入通配符“*”作为后缀。例如，以下任何指令都将引入 `EmailTagHelper`：

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers  
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

如前所述, 将 `@addTagHelper` 指令添加到 Views/_ViewImports.cshtml 文件, 将使标记帮助程序对于 Views 目录及子目录中的所有视图文件可用。如果想选择仅对特定视图公开标记帮助程序, 可在这些视图文件中使用 `@addTagHelper` 指令。

`@removeTagHelper` 删除标记帮助程序

`@removeTagHelper` 与 `@addTagHelper` 具有相同的两个参数, 它会删除之前添加的标记帮助程序。例如, 应用于特定视图的 `@removeTagHelper` 会删除该视图中的指定标记帮助程序。在 Views/Folder/_ViewImports.cshtml 文件中使用 `@removeTagHelper`, 将从 Folder 中的所有视图删除指定的标记帮助程序。

使用 _ViewImports.cshtml 文件控制标记帮助程序作用域

可将 _ViewImports.cshtml 添加到任何视图文件夹, 视图引擎将同时应用该文件和 Views/_ViewImports.cshtml 文件中的指令。如果为 Home 视图添加空的 Views/Home/_ViewImports.cshtml 文件, 则不会发生任何更改, 因为 _ViewImports.cshtml 文件是附加的。添加到 Views/Home/_ViewImports.cshtml 文件(不在默认 Views/_ViewImports.cshtml 文件中)的任何 `@addTagHelper` 指令, 都只会将这些标记帮助程序公开给 Home 文件夹中的视图。

选择退出各个元素

使用标记帮助程序选择退出字符("!"), 可在元素级别禁用标记帮助程序。例如, 使用标记帮助程序选择退出字符在 `` 中禁用 `Email` 验证:

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

须将标记帮助程序选择退出字符应用于开始和结束标记。(将选择退出字符添加到开始标记时, Visual Studio 编辑器会自动为结束标记添加相应字符)。添加选择退出字符后, 元素和标记帮助程序属性不再以独特字体显示。

使用 `@tagHelperPrefix` 阐明标记帮助程序用途

`@tagHelperPrefix` 指令可指定一个标记前缀字符串, 以后用标记帮助程序支持并阐明标记帮助程序用途。例如, 可以将以下标记添加到 Views/_ViewImports.cshtml 文件:

```
@tagHelperPrefix th:
```

在以下代码图像中, 标记帮助程序前缀设置为 `th:`, 所以只有使用前缀 `th:` 的元素才支持标记帮助程序(可使用标记帮助程序的元素以独特字体显示)。`<label>` 和 `<input>` 元素具有标记帮助程序前缀, 可使用标记帮助程序, 而 `` 元素则相反。

```
<div class="form-group">  
    <th:label asp-for="Password" class="col-md-2"></th:label>  
    <div class="col-md-10">  
        <th:input asp-for="Password" class="form-control" />  
        <span asp-validation-for="Password" class="text-danger"></span>  
    </div>  
</div>
```

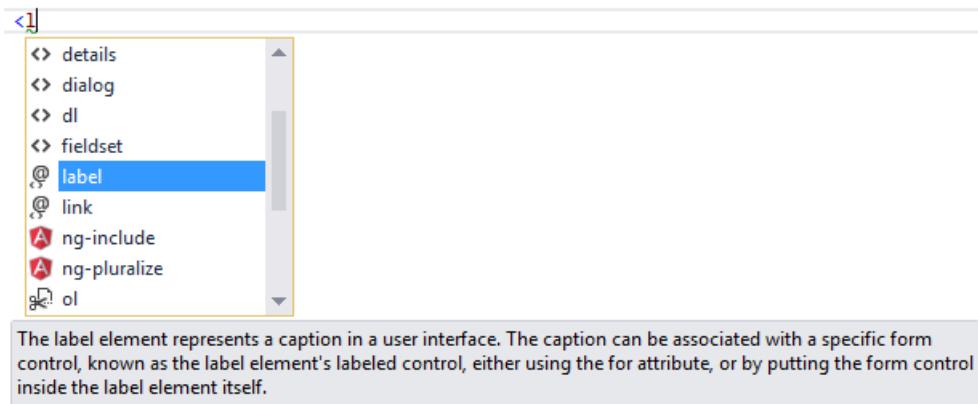
适用于 `@addTagHelper` 的层次结构规则也适用于 `@tagHelperPrefix`。

标记帮助程序的 Intellisense 支持

在 Visual Studio 中创建新的 ASP.NET Web 应用时, 会添加 NuGet

包“Microsoft.AspNetCore.Razor.Tools”。这是添加标记帮助程序工具的包。

请考虑编写 HTML `<label>` 元素。只要在 Visual Studio 编辑器中输入 `<l`，IntelliSense 就会显示匹配的元素：



不仅会获得 HTML 帮助，还会有图标（下方带有“<>”的“@”符号与“”）

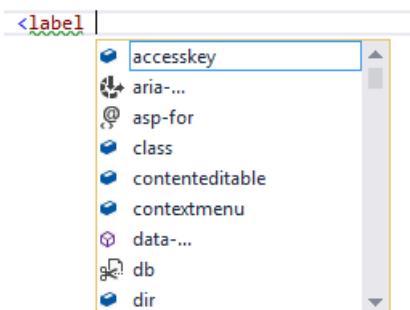


将该元素标识为标记帮助程序的目标。纯 HTML 元素（如 `fieldset`）显示“<>”图标。

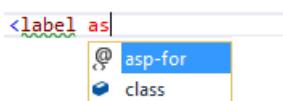
纯 HTML `<label>` 标记以棕色字体显示 HTML 标记（使用默认 Visual Studio 颜色主题时），以红色字体显示属性，并以蓝色字体显示属性值。

`<label class="col-md-2">Email</label>`

输入 `<label>` 后，IntelliSense 会列出可用的 HTML/CSS 属性和以标记帮助程序为目标的属性：



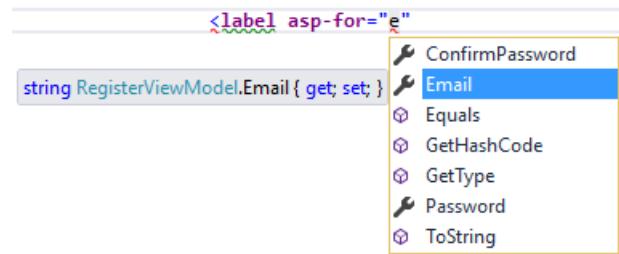
通过 IntelliSense 语句完成功能，按 Tab 键即可用选择的值完成语句：



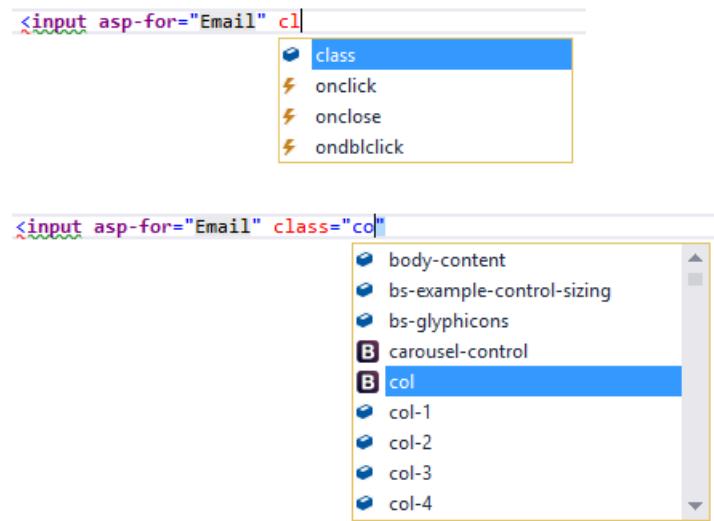
只要输入标记帮助程序属性，标记和属性字体就会更改。如果使用默认的 Visual Studio“蓝色”或“浅色”颜色主题，则字体是粗体紫色。如果使用“深色”主题，则字体为粗体青色。本文档中的图像在使用默认主题时截取的。

`<label asp-for`

可在双引号（""）内输入 Visual Studio CompleteWord 快捷方式（**默认值**为 Ctrl+空格键），即可使用 C#，就像在 C# 类中一样。IntelliSense 会显示页面模型上的所有方法和属性。由于属性类型是 `ModelExpression`，所以这些方法和属性可用。在下图中，我正在编辑 `Register` 视图，所以 `RegisterViewModel` 是可用的。



IntelliSense 会列出页面上模型可用的属性和方法。丰富 IntelliSense 环境可帮助选择 CSS 类：



标记帮助程序与 HTML 帮助程序的比较

标记帮助程序附加到 Razor 视图中的 HTML 元素, [HTML 帮助程序](#)是作为与 Razor 视图中 HTML 交织的方法被调用的。请考虑下列 Razor 标记, 它创建具有 CSS 类“caption”的 HTML 标签：

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

艾特 (@) 符号告诉 Razor 这是代码的开始。接下来的两个参数 (“FirstName”和“First Name:”) 是字符串, 所以 [IntelliSense](#) 无法提供帮助。最后一个参数：

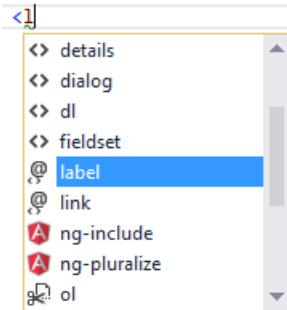
```
new {@class="caption"}
```

是用于表示属性的匿名对象。由于 `class` 是 C# 中的保留关键字, 因此要使用 @ 符号强制 C# 将“@class=”解释为符号(属性名称)。对于前端设计师(熟悉 HTML/CSS/JavaScript 及其他客户端技术, 但不熟悉 C# 和 Razor 的人), 这行代码中大部分内容是陌生的。必须在没有 IntelliSense 帮助的情况下编写完整代码。

使用 [LabelTagHelper](#), 相同标记可以编写为：

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

使用标记帮助程序版本, 只要在 Visual Studio 编辑器中输入 <1, IntelliSense 就会显示匹配的元素：



The `label` element represents a caption in a user interface. The caption can be associated with a specific form control, known as the `label` element's labeled control, either using the `for` attribute, or by putting the form control inside the `label` element itself.

IntelliSense 可帮助编写整行。`LabelTagHelper` 也默认将 `asp-for` 属性值("FirstName")的内容设置为"First Name":即在属性值中每个大写字母前添加一个空格, 将驼峰式大小写的属性转换为由属性名称组成的语句。在下列标记中:

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

生成:

```
<label class="caption" for="FirstName">First Name</label>
```

如果将内容添加到 `<label>`, 则不会使用由驼峰式大小写转换为语句式书写的内容。例如:

```
<label class="caption" asp-for="FirstName">Name First</label>
```

生成:

```
<label class="caption" for="FirstName">Name First</label>
```

以下代码图像显示了由 Visual Studio 2015 包含的旧版 ASP.NET 4.5.x MVC 模板生成的 Views/Account/Register.cshtml Razor 视图的 Form 部分。

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizonal" })
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

Visual Studio 编辑器以灰色背景显示 C# 代码。例如，`AntiForgeryToken` HTML 帮助程序：

```
@Html.AntiForgeryToken()
```

以灰色背景显示。Register 视图中的标记大部分是 C#。将其与使用标记帮助程序的等效方法进行比较：

```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

与 HTML 帮助程序方法相比，此标记更清晰，更容易阅读、编辑和维护。C# 代码会被减少至服务器需要知道的最小值。Visual Studio 编辑器以独特的字体显示标记帮助程序的目标标记。

请考虑 Email 组：

```
<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

每个“asp-”属性都有一个“Email”值，但是“Email”不是字符串。在此上下文中，“Email”是 `RegisterViewModel` 的 C# 模型表达式属性。

Visual Studio 编辑器可帮助编写注册窗体的标记帮助程序方法中的所有标记，而 Visual Studio 不会为 HTML 帮助程序方法中的大多数代码提供帮助。[标记帮助程序的 IntelliSense 支持](#)详细介绍了如何在 Visual Studio 编辑器中使用标记帮助程序。

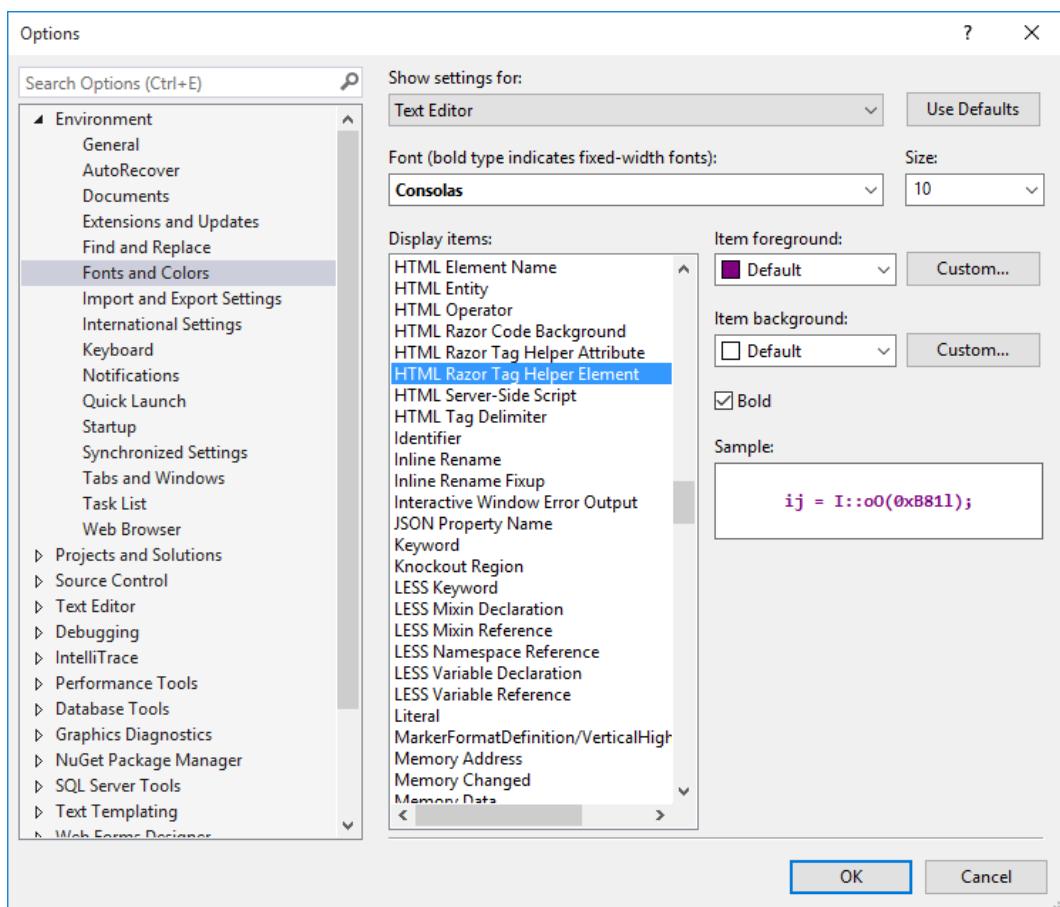
标记帮助程序与 Web 服务器控件的比较

- 标记帮助程序不拥有与其相关的元素：它们只是参与元素和内容的呈现。ASP.NET Web 服务器控件在页面上进行声明和调用。
- [Web 服务器控件](#)具有可观的生命周期，因而难以进行开发和调试。

- 通过 Web 服务器控件，可使用客户端控件向客户端文档对象模型 (DOM) 元素添加功能。标记帮助程序没有 DOM。
- Web 服务器控件包括自动浏览器检测。标记帮助程序不了解浏览器。
- 通常不能撰写 Web 服务器控件时，多个标记帮助程序可作用于同一元素（请参阅[避免标记帮助程序冲突](#)）。
- 标记帮助程序可以修改其作用域内 HTML 元素的标记和内容，但不会直接修改页面上的其他内容。Web 服务器控件的作用域较广，并且可以执行影响页面其他部分的操作，从而可能造成意想不到的副作用。
- Web 服务器控件使用类型转换器将字符串转换为对象。使用标记帮助程序时，本身就用 C# 语言工作，因此无需进行类型转换。
- Web 服务器控件使用 [System.ComponentModel](#) 实现组件和控件的运行时和设计时行为。[System.ComponentModel](#) 包括用于属性和类型转换器的实现、数据源绑定和组件授权的基类和接口。与通常派生自 [TagHelper](#) 的标记帮助程序相比，[TagHelper](#) 基类仅公开两个方法，即 [Process](#) 和 [ProcessAsync](#)。

自定义标记帮助程序元素字体

可以在“工具”>“选项”>“环境”>“字体和颜色”中自定义字体和着色：



其他资源

- [创作标记帮助程序](#)
- [使用窗体](#)
- [GitHub 上的 TagHelperSamples](#) 包含用于处理 [Bootstrap](#) 的标记帮助程序示例。

在 ASP.NET Core 中创作标记帮助程序

2018/5/14 • 18 min to read • [Edit Online](#)

作者: Rick Anderson

[查看或下载示例代码\(如何下载\)](#)

标记帮助程序入门

本教程介绍标记帮助程序编程。[标记帮助程序简介](#)描述了标记帮助程序提供的优势。

标记帮助程序是实现 `ITagHelper` 接口的任何类。但是，在创作标记帮助程序时，通常从 `TagHelper` 派生，这样可以访问 `Process` 方法。

1. 创建一个名为 `AuthoringTagHelpers` 的新 ASP.NET Core 项目。此项目不需要身份验证。
2. 创建一个名为“`TagHelpers`”的文件夹来保存标记帮助程序。“`TagHelpers`”文件夹不是必需的，但它是合理的约定。现在让我们开始编写一些简单的标记帮助程序。

最小的标记帮助程序

在本部分中，你将编写一个更新电子邮件标记的标记帮助程序。例如：

```
<email>Support</email>
```

服务器将使用电子邮件标记帮助程序将该标记转换为以下内容：

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

即，使其成为电子邮件链接的定位标记。如果你正在编写博客引擎，并且需要它将营销、支持和其他联系人的电子邮件全部发送到同一个域，则可能需要执行此操作。

1. 将以下 `EmailTagHelper` 类添加到“`TagHelpers`”文件夹。

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";      // Replaces <email> with <a> tag
        }
    }
}
```

注意：

- 标记帮助程序使用面向根类名称的元素的命名约定(减去类名称的 `TagHelper` 部分)。在此示例中，`EmailTagHelper` 的根名称是 `email`，因此 `<email>` 标记将作为目标名称。此命名约定应适用于

大多数标记帮助程序，稍后将介绍如何重写它。

- `EmailTagHelper` 类派生自 `TagHelper`。 `TagHelper` 类提供编写标记帮助程序的方法和属性。
- 重写的 `Process` 方法控制标记帮助程序在执行时的操作。`TagHelper` 类还提供具有相同参数的异步版本 (`ProcessAsync`)。
- `Process` (和 `ProcessAsync`) 的上下文参数包含与执行当前 HTML 标记相关的信息。
- `Process` (和 `ProcessAsync`) 的输出参数包含监控状态的 HTML 元素，它代表用于生成 HTML 标记和内容的原始源。
- 类名称的后缀是 `TagHelper`，这不是必需的，但被认为是最佳做法约定。可将类声明为：

```
public class Email : TagHelper
```

2. 要使 `EmailTagHelper` 类可用于所有 Razor 视图，请将 `@addTagHelper` 指令添加到 `Views/_ViewImports.cshtml` 文件：[!code-html]

上面的代码使用通配符语法来指定程序集中的所有标记帮助程序都将可用。`@addTagHelper` 之后的第一个字符串指定要加载的标记帮助程序（对所有标记帮助程序使用“*”），第二个字符串“AuthoringTagHelpers”指定标记帮助程序所在的程序集。另请注意，第二行使用通配符语法引入了 ASP.NET Core MVC 标记帮助程序（[标记帮助程序简介](#) 中讨论了这些帮助程序。）要使标记帮助程序可用于 Razor 视图，请使用 `@addTagHelper` 指令。或者，也可以提供标记帮助程序的完全限定的名称 (FQN)，如下所示：

```
@using AuthoringTagHelpers  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers  
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

要使用 FQN 将标记帮助程序添加到视图，请首先添加 FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`)，然后添加程序集名称 (AuthoringTagHelpers)。大多数开发者更喜欢使用通配符语法。[标记帮助程序简介](#) 详细介绍了标记帮助程序的添加和删除方法，以及层次结构和通配符语法。

3. 使用以下更改更新 `Views/Home/Contact.cshtml` 文件中的标记：

```
@{  
    ViewData["Title"] = "Contact";  
}  
<h2>@ViewData["Title"].</h2>  
<h3>@ViewData["Message"]</h3>  
  
<address>  
    One Microsoft Way<br />  
    Redmond, WA 98052<br />  
    <abbr title="Phone">P:</abbr>  
    425.555.0100  
</address>  
  
<address>  
    <strong>Support:</strong><email>Support</email><br />  
    <strong>Marketing:</strong><email>Marketing</email>  
</address>
```

4. 运行应用并使用你喜爱的浏览器来查看 HTML 源，以便验证电子邮件标记是否替换为定位标记（例如，`<a>Support`）。Support 和 Marketing 呈现为链接，但它们不具备使其正常工作的 `href` 属性。此问题将在下一部分得以解决。

SetAttribute 和 SetContent

在本部分中，我们将更新 `EmailTagHelper`，使其能够为电子邮件创建有效的定位标记。我们将对其进行更新以获取 Razor 视图中的信息（采用 `mail-to` 属性的形式）并使用该信息来生成定位点。

使用以下内容更新 `EmailTagHelper` 类：

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";      // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

注意：

- 标记帮助程序采用 Pascal 大小写格式的类和属性名将转换为各自相应的小写短横线格式。因此，要使用 `MailTo` 属性，请使用 `<email mail-to="value"/>` 等效项。
- 最后一行为最小功能标记帮助程序设置已完成的内容。
- 突出显示的行显示了添加属性的语法：

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";      // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

只要属性集合中当前不存在“`href`”属性，该方法就适用于此属性。也可使用 `output.Attributes.Add` 方法将标记帮助程序属性添加到标记属性集合的末尾。

1. 使用以下更改更新 `Views/Home/Contact.cshtml` 文件中的标记：[!code-html]
2. 运行应用并验证它是否生成正确的链接。

注意

如果打算编写电子邮件标记自结束 (`<email mail-to="Rick" />`), 最终输出也将为自结束。要启用只使用开始标记 (`<email mail-to="Rick">`) 来编写标记的功能, 必须用以下内容修饰类:

```
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

鉴于自结束电子邮件标记帮助程序, 输出将为 ``。自结束定位标记不是有效的 HTML, 因此你不想创建这样的标记, 但你可能想要创建一个自结束的标记帮助程序。标记帮助程序在读取标记后设置 `TagMode` 属性的类型。

ProcessAsync

在本部分中, 我们将编写异步电子邮件帮助程序。

1. 将 `EmailTagHelper` 类替换为以下代码:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";                                // Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}
```

注意:

- 此版本使用异步 `ProcessAsync` 方法。异步 `GetChildContentAsync` 返回包含 `TagHelperContent` 的 `Task`。
- 使用 `output` 参数获取 HTML 元素的内容。

2. 对 Views/Home/Contact.cshtml 文件进行以下更改, 以便标记帮助程序能够获取目标电子邮件。

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. 运行应用并验证它是否生成有效的电子邮件链接。

RemoveAll、PreContent.SetHtmlContent 和 PostContent.SetHtmlContent

1. 将以下 `BoldTagHelper` 类添加到“TagHelpers”文件夹。

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

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

注意：

- `[HtmlTargetElement]` 属性传递一个属性参数，该参数指定包含名为“bold”的 HTML 属性的任何 HTML 元素都将匹配，并且该类中的 `Process` 重写方法将会运行。在此示例中，`Process` 方法删除了“bold”属性，并用 `` 围住了包含的标记。
- 因为你不想替换现有的标记内容，所以必须用 `PreContent.SetHtmlContent` 方法编写开头的 `` 标记，并用 `PostContent.SetHtmlContent` 方法编写结尾的 `` 标记。

2. 修改 About.cshtml 视图，以包含 `bold` 属性值。完成的代码如下所示。

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. 运行应用。可以使用你喜爱的浏览器来检查源并验证标记。

上面的 `[HtmlTargetElement]` 属性仅针对提供属性名称“bold”的 HTML 标记。标记帮助程序未修改 `<bold>` 元素。

4. 标注出 `[HtmlTargetElement]` 属性行，它将默认为目标 `<bold>` 标记，也就是 `<bold>` 格式的 HTML 标记。请记住，默认的命名约定会将类名称 `BoldTagHelper` 与 `<bold>` 标记相匹配。

5. 运行应用并验证 `<bold>` 标记是否由标记帮助程序进行处理。

用多个 `[HtmlTargetElement]` 属性修饰类会导致目标出现逻辑 OR。例如，使用下面的代码时，系统将匹配出 `bold` 标记或 `bold` 属性。

```
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

将多个属性添加到同一语句时，运行时会将其视为逻辑 AND。例如，在下面的代码中，HTML 元素必须命名为“bold”并具有名为“bold”的属性（`<bold bold />`）才能匹配。

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

也可使用 `[HtmlTargetElement]` 更改目标元素的名称。例如，如果你希望 `BoldTagHelper` 以 `<MyBold>` 标记为目标，则可使用以下属性：

```
[HtmlTargetElement("MyBold")]
```

将模型传递到标记帮助程序

1. 添加“Models”文件夹。
2. 将以下 `WebsiteContext` 类添加到“模型”文件夹：

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. 将以下 `WebsiteInformationTagHelper` 类添加到“TagHelpers”文件夹。

```

using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "section";
            output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li></ul>");
            output.TagMode = TagMode.StartTagAndEndTag;
        }
    }
}

```

注意：

- 如前所述，标记帮助程序会将标记帮助程序采用 Pascal 大小写格式的 C# 类名和属性转换为**小写短横线格式**。因此，要在 Razor 中使用 `WebsiteInformationTagHelper`，请编写 `<website-information />`。
- 未显式标识具有 `[HtmlTargetElement]` 属性的目标元素，因此 `website-information` 的默认值将成为目标元素。如果应用了以下属性（请注意，它虽不是短横线格式，但却与类名相匹配）：

```
[HtmlTargetElement("WebsiteInformation")]
```

小写短横线格式标记 `<website-information />` 不匹配。若要使用 `[HtmlTargetElement]` 属性，请使用短横线格式，如下所示：

```
[HtmlTargetElement("Website-Information")]
```

- 自结束的元素没有任何内容。在此示例中，Razor 标记将使用自结束标记，但标记帮助程序将创建 `section` 元素（这不是自结束元素，并且你将在 `section` 元素中编写内容）。因此，需要将 `TagMode` 设置为 `StartTagAndEndTag` 以写入输出。或者，可以标注出行设置 `TagMode` 并用结束标记编写标记。（本教程后面将提供示例标记。）
- 下一行中的 `$`（美元符号）使用**内插字符串**：

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

- 将以下标记添加到 About.cshtml 视图。突出显示的标记显示 Web 站点信息。

```
@using AuthoringTagHelpers.Models  
{@  
    ViewData["Title"] = "About";  
}  
<h2>@ViewData["Title"].</h2>  
<h3>@ViewData["Message"]</h3>  
  
<p bold>Use this area to provide additional information.</p>  
  
<bold> Is this bold?</bold>  
  
<h3> web site info </h3>  
<website-information info="new WebsiteContext {  
    Version = new Version(1, 3),  
    CopyrightYear = 1638,  
    Approved = true,  
    TagsToShow = 131 }" />
```

注意

在 Razor 中，标记如下所示：

```
<website-information info="new WebsiteContext {  
    Version = new Version(1, 3),  
    CopyrightYear = 1638,  
    Approved = true,  
    TagsToShow = 131 }" />
```

Razor 知道 `info` 属性是一个类，而不是字符串，并且你想要编写 C# 代码。编写任何非字符串标记帮助程序属性时，都不应使用 `@` 字符。

- 运行应用，并导航到“关于”视图查看 Web 站点信息。

注意

可使用带有结束标记的以下标记，并在标记帮助程序中删除带有 `TagMode.StartTagAndEndTag` 的行：

```
<website-information info="new WebsiteContext {  
    Version = new Version(1, 3),  
    CopyrightYear = 1638,  
    Approved = true,  
    TagsToShow = 131 }" >  
</website-information>
```

条件标记帮助程序

条件标记帮助程序在传递 `true` 值时呈现输出。

- 将以下 `ConditionTagHelper` 类添加到“TagHelpers”文件夹。

```

using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}

```

2. 将 Views/Home/Index.cshtml 文件的内容替换为以下标记：

```

@using AuthoringTagHelpers.Models
@model WebsiteContext

 @{
     ViewData["Title"] = "Home Page";
 }

<div>
    <h3>Information about our website (outdated):</h3>
    <website-information info=@Model />
    <div condition="@Model.Approved">
        <p>
            This website has <strong surround="em"> @Model.Approved </strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>

```

3. 将 Home 控制器中的 Index 方法替换为以下代码：

```

public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}

```

4. 运行应用并浏览到主页。条件 div 中的标记不会呈现。将查询字符串 ?approved=true 追加到 URL（例如，<http://localhost:1235/Home/Index?approved=true>）。approved 设置为 true，并将显示条件标记。

注意

使用 `nameof` 运算符将属性指定为目标，而不是像使用 `bold` 标记帮助程序那样指定字符串：

```
[HtmlTargetElement(Attributes = nameof(Condition))]
// [HtmlTargetElement(Attributes = "condition")]
public class ConditionTagHelper : TagHelper
{
    public bool Condition { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        if (!Condition)
        {
            output.SuppressOutput();
        }
    }
}
```

如果代码被重构，`nameof` 运算符将保护它（可能需要将名称更改为 `RedCondition`）。

避免标记帮助程序冲突

在本部分中，你将编写一对自动链接标记帮助程序。第一个标记帮助程序会将包含以 HTTP 开头的 URL 的标记替换为包含相同 URL 的 HTML 定位标记（从而产生指向 URL 的链接）。第二个标记帮助程序也会对以 WWW 开头的 URL 执行相同的操作。

由于这两个帮助程序密切相关，并且你将来可能会重构它们，因此可将其保存在同一文件中。

- 将以下 `AutoLinkerHttpTagHelper` 类添加到“TagHelpers”文件夹。

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(?:https?:\/\/)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version}
    }
}
```

注意

`AutoLinkerHttpTagHelper` 类以 `p` 元素为目标，并使用 [正则表达式](#) 来创建定位点。

- 将以下标记添加到 Views/Home/Contact.cshtml 文件的末尾：

```

@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>

```

3. 运行应用并验证标记帮助程序是否正确呈现定位点。

4. 更新 `AutoLinker` 类以包含 `AutoLinkerWwwTagHelper`，这会将 `www` 文本转换为还包含原始 `www` 文本的定位标记。更新后的代码在下方突出显示：

```

[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(?:https?:\/\/)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>"); // http link version)
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\b(www\.)\S+\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>"); // www version
    }
}

```

5. 运行应用。请注意 `www` 文本呈现为链接，但 `HTTP` 文本不是。如果将中断点放在这两个类中，可以看到 `HTTP` 标记帮助程序类首先运行。问题是，标记帮助程序输出已缓存，当运行 `WWW` 标记帮助程序时，它会覆盖 `HTTP` 标记帮助程序的缓存输出。本教程稍后将介绍如何控制标记帮助程序中的运行顺序。我们将用以下方法修复代码：

```

public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(?:https?:\/\/)(\S+)\b",
            "<a target=_blank" href=\"$0\"">$0</a>"); // http link version
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(www\.)(\S+)\b",
            "<a target=_blank" href="http://$0\"">$0</a>"); // www version
    }
}

```

注意

在自动链接标记帮助程序的第一版中，使用以下代码获取了目标的内容：

```
var childContent = await output.GetChildContentAsync();
```

也就是说，使用传递到 `ProcessAsync` 方法的 `TagHelperOutput` 调用 `GetChildContentAsync`。如前所述，由于输出已缓存，因此将调用最后运行的标记帮助程序。使用以下代码解决了这个问题：

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

上面的代码检查内容是否已修改，如果已修改，则从输出缓冲区获取内容。

6. 运行应用并验证这两个链接是否按预期工作。虽然它可能显示自动链接器标记帮助程序是正确且完整的，但它有一个细微的问题。如果首先运行 WWW 标记帮助程序，则 www 链接不正确。通过添加 `Order` 重载更新代码来控制标记运行的顺序。`Order` 属性确定相对于面向同一元素的其他标记帮助程序的执行顺序。默认顺序值为零，并首先执行具有较低值的实例。

```

public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get { return int.MinValue; }
    }
}

```

上面的代码可以保证 HTTP 标记帮助程序在 WWW 标记帮助程序之前运行。将 `Order` 更改为 `MaxValue` 并验证为 WWW 标记生成的标记是否不正确。

检查和检索子内容

标记帮助程序提供多个属性来检索内容。

- 可将 `GetChildContentAsync` 的结果追加到 `output.Content`。
- 可使用 `GetContent` 检查 `GetChildContentAsync` 的结果。
- 如果修改 `output.Content`，则不会执行或呈现 `TagHelper` 主体，除非像自动链接器示例中那样调用 `GetChildContentAsync`：

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find URLs in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\b(?:https?:\/\/)(\S+)\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>"); // http link version}
    }
}
```

- 对 `GetChildContentAsync` 的多次调用返回相同的值，且不重新执行 `TagHelper` 主体，除非传入一个指示不使用缓存结果的 `false` 参数。

ASP.NET Core 表单中的标记帮助程序

2018/5/14 • 20 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Dave Paquette](#) 和 [Jerrie Pelser](#)

本文档演示如何使用表单和表单中常用的 HTML 元素。HTML `Form` 元素提供 Web 应用用于向服务器回发数据的主要机制。本文档的大部分内容介绍标记帮助程序及其如何帮助高效创建可靠的 HTML 表单。建议在阅读本文档前先阅读[标记帮助程序简介](#)。

在很多情况下，HTML 帮助程序为特定标记帮助程序提供了一种替代方法，但标记帮助程序不会替代 HTML 帮助程序，且并非每个 HTML 帮助程序都有对应的标记帮助程序，认识到这点也很重要。如果存在 HTML 帮助程序替代项，文中会提到。

表单标记帮助程序

[表单](#)标记帮助程序：

- 为 MVC 控制器操作或命名路由生成 HTML `<FORM>` `action` 属性值
- 生成隐藏的[请求验证令牌](#)，防止跨站点请求伪造(在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时)
- 提供 `asp-route-<Parameter Name>` 属性，其中 `<Parameter Name>` 添加到路由值。`Html.BeginForm` 和 `Html.BeginRouteForm` 的 `routeValues` 参数提供类似的功能。
- 具有 HTML 帮助程序替代项 `Html.BeginForm` 和 `Html.BeginRouteForm`

示例：

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

上述表单标记帮助程序生成以下 HTML：

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

MVC 运行时通过表单标记帮助程序属性 `asp-controller` 和 `asp-action` 生成 `action` 属性值。表单标记帮助程序还会生成隐藏的[请求验证令牌](#)，防止跨站点请求伪造(在 HTTP Post 操作方法中与 `[ValidateAntiForgeryToken]` 属性配合使用时)。保护纯 HTML 表单免受跨站点请求伪造的影响很难，但表单标记帮助程序可提供此服务。

使用命名路由

`asp-route` 标记帮助程序属性还可为 HTML `action` 属性生成标记。具有名为 `register` 的[路由](#)的应用可将以下标记用于注册页：

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Views/Account 文件夹中的许多视图(在新建使用个人用户帐户身份验证的 Web 应用时生成)包含 `asp-route-returnurl` 属性:

```
<form asp-controller="Account" asp-action="Login"
    asp-route-returnurl="@ViewData["ReturnUrl"]"
    method="post" class="form-horizontal" role="form">
```

注意

使用内置模板时, `returnUrl` 仅会在用户尝试访问授权资源, 但未验证身份或未获得授权的情况下自动填充。如果尝试执行未经授权的访问, 安全中间件会使用 `returnUrl` 集将用户重定向至登录页。

输入标记帮助程序

输入标记帮助程序将 HTML `<input>` 元素绑定到 Razor 视图中的模型表达式。

语法:

```
<input asp-for="<Expression Name>" />
```

输入标记帮助程序:

- 为 `asp-for` 属性中指定的表达式名称生成 `id` 和 `name` HTML 属性。
`asp-for="Property1.Property2"` 与 `m => m.Property1.Property2` 相等。表达式的名称用于 `asp-for` 属性值。有关其他信息, 请参阅[表达式名称](#)部分。
- 根据模型类型和应用于模型属性的[数据注释](#)特性设置 HTML `type` 特性值
- 如果已经指定, 不会覆盖 HTML `type` 属性值
- 通过应用于模型属性的[数据注释](#)特性生成 HTML5 验证特性
- 具有与 `Html.TextBoxFor` 和 `Html.EditorFor` 重叠的 HTML 帮助程序功能。有关详细信息, 请参阅[输入标记帮助程序的 HTML 帮助程序替代项](#)部分。
- 提供强类型化。如果属性的名称更改, 但未更新标记帮助程序, 则会收到类似如下内容的错误:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.
```

Type expected

```
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

`Input` 标记帮助程序根据 .NET 类型设置 HTML `type` 属性。下表列出一些常见的 .NET 类型和生成的 HTML 类型(并未列出每个 .NET 类型)。

.NET 类型	输入类型
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte	type="number"
Int	type="number"
Single、Double	type="number"

下表显示输入标记帮助程序会映射到特定输入类型的一些常见数据注释属性(并未列出每个验证属性)：

特性	输入类型
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

示例：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email: <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>

```

上述代码生成以下 HTML：

```

<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
           data-val-email="The Email Address field is not a valid email address."
           data-val-required="The Email Address field is required."
           id="Email" name="Email" value="" /> <br>
    Password:
    <input type="password" data-val="true"
           data-val-required="The Password field is required."
           id="Password" name="Password" /><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>

```

应用于 `Email` 和 `Password` 属性的数据注释在模型中生成元数据。输入标记帮助程序使用模型元数据并生成 HTML5 `data-val-*` 属性（请参阅[模型验证](#)）。这些属性描述要附加到输入字段的验证程序。这样可以提供非介入式 HTML5 和 [jQuery](#) 验证。非介入式属性具有格式 `data-val-rule="Error Message"`，其中规则是验证规则的名称（例如 `data-val-required`、`data-val-email`、`data-val-maxlength` 等）。如果在属性中提供错误消息，则该错误消息会作为 `data-val-rule` 属性的值显示。还有表单 `data-val-ruleName-argumentName="argumentValue"` 的属性，这些属性提供有关规则的其他详细信息，例如，`data-val-maxlength-max="1024"`。

输入标记帮助程序的 HTML 帮助程序替代项

`Html.TextBox`、`Html.TextBoxFor`、`Html.Editor` 和 `Html.EditorFor` 与输入标记帮助程序的功能存在重叠。输入标记帮助程序会自动设置 `type` 属性；而 `Html.TextBox` 和 `Html.TextBoxFor` 不会。`Html.Editor` 和 `Html.EditorFor` 处理集合、复杂对象和模板；而输入标记帮助程序不会。输入标记帮助程序、`Html.EditorFor` 和 `Html.TextBoxFor` 是强类型（使用 lambda 表达式）；而 `Html.TextBox` 和 `Html.Editor` 不是（使用表达式名称）。

HtmlAttributes

`@Html.Editor()` 和 `@Html.EditorFor()` 在执行其默认模板时使用名为 `htmlAttributes` 的特殊 `ViewDataDictionary` 条目。此行为可选择使用 `additional ViewData` 参数增强。键“`htmlAttributes`”区分大小写。键“`htmlAttributes`”的处理方式与传递到输入帮助程序的 `htmlAttributes` 对象（例如 `@Html.TextBox()`）的处理方式类似。

```

@Html.EditorFor(model => model.YourProperty,
new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })

```

表达式名称

`asp-for` 属性值是 `ModelExpression`，并且是 lambda 表达式的右侧。因此，`asp-for="Property1"` 在生成的代码中变成 `m => m.Property1`，这也是无需使用 `Model` 前缀的原因。可使用“@”字符作为内联表达式的开头并移到 `m` 之前：

```
@{
    var joe = "Joe";
}
<input asp-for="@joe" />
```

生成以下 HTML：

```
<input type="text" id="joe" name="joe" value="Joe" />
```

使用集合属性时，`asp-for="CollectionProperty[23].Member"` 在 `i` 具有值 `23` 时生成与 `asp-for="CollectionProperty[i].Member"` 相同的名称。

在 ASP.NET Core MVC 计算 `ModelExpression` 的值时，它会检查多个源，包括 `ModelState`。以 `<input type="text" asp-for="@Name" />` 为例。计算出的 `value` 属性是第一个非 null 值，属于：

- 带有“Name”键的 `ModelState` 条目。
- `Model.Name` 表达式的结果。

导航子属性

还可使用视图模型的属性路径导航到子属性。设想一个包含子 `Address` 属性的更复杂的模型类。

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

在视图中，绑定到 `Address.AddressLine1`：

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email: <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

为 `Address.AddressLine1` 生成以下 HTML：

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

表达式名称和集合

包含 `Colors` 数组的模型示例:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

操作方法:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

以下 Razor 显示如何访问特定 `Color` 元素:

```
@model Person
 @{
     var index = (int)ViewData["index"];
 }

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

Views/Shared/EditorTemplates/String.cshtml 模板:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

使用 `List<T>` 的示例:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

以下 Razor 演示如何循环访问集合:

```

@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>

```

Views/Shared/EditorTemplates/ToDoItem.cshtml 模板：

```

@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

<@

    This template replaces the following Razor which evaluates the indexer three
times.
<td>
    <label asp-for="@Model[i].Name"></label>
    @Html.DisplayFor(model => model[i].Name)
</td>
<td>
    <input asp-for="@Model[i].IsDone" />
</td>
*@

```

注意

始终使用 `for` (而不要使用 `foreach`) 循环访问列表。计算 LINQ 表达式中的索引器会耗费大量资源，应尽量减少此类计算。

注意

上述带有注释的示例代码演示如何将 lambda 表达式替换为 `@` 运算符来访问列表中的每个 `ToDoItem`。

文本区标记帮助程序

`Textarea Tag Helper` 标记帮助程序类似于输入标记帮助程序。

- 通过模型为 `<textarea>` 元素生成 `id` 和 `name` 属性以及数据验证属性。

- 提供强类型化。
- HTML 帮助程序替代项: `Html.TextAreaFor`

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

生成以下 HTML:

```
<form method="post" action="/Demo/RegisterTextArea">
    <textarea data-val="true"
              data-val-maxlength="The field Description must be a string or array type with a
              maximum length of 1024."
              data-val-maxlength-max="1024"
              data-val-minlength="The field Description must be a string or array type with a
              minimum length of 5."
              data-val-minlength-min="5"
              id="Description" name="Description">
    </textarea>
    <button type="submit">Test</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

标签标记帮助程序

- 在 `元素` 中为表达式名称生成标签描述和 `for` 属性
- HTML 帮助程序替代项: `Html.LabelFor`。

`Label Tag Helper` 通过纯 HTML 标签元素提供如下优势:

- 可自动从 `Display` 属性中获取描述性标签值。预期的显示名称可能会随时间变化, `Display` 属性和标签标记帮助程序的组合会在其被使用的所有位置应用 `Display`。
- 源代码中的标记更少
- 模型属性的强类型化。

示例:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

为 `<label>` 元素生成以下 HTML：

```
<label for="Email">Email Address</label>
```

标签标记帮助程序生成“Email”的 `for` 属性值，即与 `<input>` 元素关联的 ID。标记帮助程序生成一致的 `id` 和 `for` 元素，方便将其正确关联。本示例中的描述来自 `Display` 属性。如果模型不包含 `Display` 特性，描述将为表达式的属性名称。

验证标记帮助程序

有两个验证标记帮助程序。`Validation Message Tag Helper`（为模型中的单个属性显示验证消息）和 `Validation Summary Tag Helper`（显示验证错误的摘要）。`Input Tag Helper` 根据模型类的数据注释属性将 HTML5 客户端验证属性添加到输入元素中。同时在服务器上执行验证。验证标记帮助程序会在发生验证错误时显示这些错误消息。

验证消息标记帮助程序

- 将 `HTML5` `data-valmsg-for="property"` 属性添加到 `span` 元素中，该元素会附加指定模型属性的输入字段中的验证错误消息。`jQuery` 会在发生客户端验证错误时在 `` 元素中显示错误消息。
- 还会在服务器上执行验证。客户端可能已禁用 JavaScript，一些验证仅可在服务器端执行。
- HTML 帮助程序替代项：`Html.ValidationMessageFor`

`Validation Message Tag Helper` 与 `HTML span` 元素中的 `asp-validation-for` 属性配合使用。

```
<span asp-validation-for="Email"></span>
```

验证消息标记帮助程序会生成以下 HTML：

```
<span class="field-validation-valid"
      data-valmsg-for="Email"
      data-valmsg-replace="true"></span>
```

对于同一属性，通常在 `Input` 标记帮助程序后使用 `Validation Message Tag Helper`。这样做可在导致错误的输入附近显示所有验证错误消息。

注意

必须拥有包含正确的 JavaScript 和 `jQuery` 脚本引用的视图才能执行客户端验证。有关详细信息，请参阅[模型验证](#)。

发生服务器端验证错误时（例如，禁用自定义服务器端验证或客户端验证时），MVC 会将该错误消息作为 `` 元素的主体。

```
<span class="field-validation-error" data-valmsg-for="Email"
      data-valmsg-replace="true">
    The Email Address field is required.
</span>
```

验证摘要标记帮助程序

- 针对具有 `asp-validation-summary` 属性的 `<div>` 元素
- HTML 帮助程序替代项：`@Html.ValidationSummary`

`Validation Summary Tag Helper` 用于显示验证消息的摘要。`asp-validation-summary` 属性值可以是以下任意值：

ASP-VALIDATION-SUMMARY	显示的验证消息
<code>ValidationSummary.All</code>	属性和模型级别
<code>ValidationSummary.ModelOnly</code>	模型
<code>ValidationSummary.None</code>	无

示例

在以下示例中，数据模型使用 `DataAnnotation` 属性修饰，在 `<input>` 元素中生成验证错误消息。验证标记帮助程序会在发生验证错误时显示错误消息：

```

using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}

```

```

@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email: <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>

```

生成的 HTML(如果模型有效)：

```

<form action="/DemoReg/Register" method="post">
    <div class="validation-summary-valid" data-valmsg-summary="true">
        <ul><li style="display:none"></li></ul></div>
    Email: <input name="Email" id="Email" type="email" value="" data-val-required="The Email field is required." data-val-email="The Email field is not a valid email address." data-val="true"> <br>
    <span class="field-validation-valid" data-valmsg-replace="true" data-valmsg-for="Email"></span><br />
    Password: <input name="Password" id="Password" type="password" data-val-required="The Password field is required." data-val="true"><br />
    <span class="field-validation-valid" data-valmsg-replace="true" data-valmsg-for="Password"></span><br />
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>

```

选择标记帮助程序

- 为模型属性生成 `select` 元素和关联的 `option` 元素。
- 具有 HTML 帮助程序替代项 `Html.DropDownListFor` 和 `Html.ListBoxFor`

`Select Tag Helper` `asp-for` 为 `select` 元素指定模型属性名称, `asp-items` 指定 `option` 元素。例如:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

示例:

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

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" }
        };
    }
}
```

Index 方法初始化 `CountryViewModel`，设置选定的国家/地区并将其传递到 Index 视图。

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTTP POST Index 方法显示选定内容：

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

Index 视图：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

生成以下 HTML(选择“CA”时)：

```
<form method="post" action="/">
    <select id="Country" name="Country">
        <option value="MX">Mexico</option>
        <option selected="selected" value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

注意

不建议将 `ViewBag` 或 `ViewData` 与选择标记帮助程序配合使用。视图模型在提供 MVC 元数据方面更可靠且通常不容易出现问题。

`asp-for` 属性值是特殊情况，它不要求提供 `Model` 前缀，但其他标记帮助程序属性需要该前缀（例如 `asp-items`）

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

枚举绑定

通常可方便地将 `<select>` 与 `enum` 属性配合使用并通过 `enum` 值生成 `SelectListItem` 元素。

示例：

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

`GetEnumSelectList` 方法为枚举生成 `SelectList` 对象。

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()" >
    </select>
    <br /><button type="submit">Register</button>
</form>
```

可使用 `Display` 属性修饰枚举器列表，以获取更丰富的 UI：

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

生成以下 HTML：

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is required." id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

选项组

如果视图模型包含一个或多个 `SelectListGroup` 对象，则会生成 HTML `<optgroup>` 元素。

`CountryViewModelGroup` 将 `SelectListItem` 元素分组为“North America”组和“Europe”组：

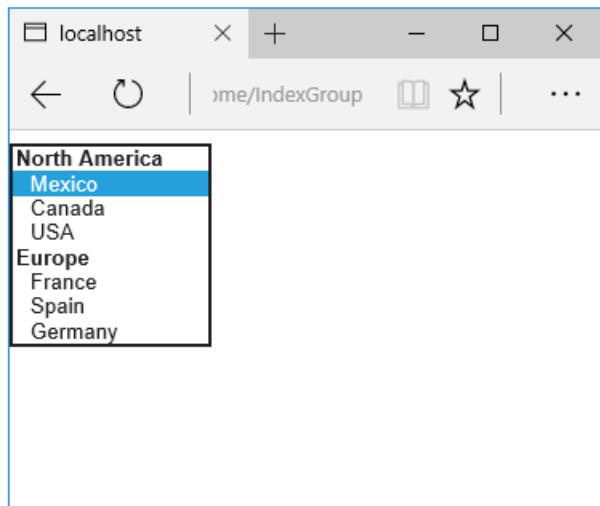
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
        };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
}
```

两个组如下所示：



生成的 HTML：

```
<form method="post" action="/Home/IndexGroup">
    <select id="Country" name="Country">
        <optgroup label="North America">
            <option value="MEX">Mexico</option>
            <option value="CAN">Canada</option>
            <option value="US">USA</option>
        </optgroup>
        <optgroup label="Europe">
            <option value="FR">France</option>
            <option value="ES">Spain</option>
            <option value="DE">Germany</option>
        </optgroup>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

多重选择

如果 `asp-for` 属性中指定的属性为 `IEnumerable`，选择标记帮助程序会自动生成 `multiple = "multiple"` 属性。例如，如果给定以下模型：

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

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel : IEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain" },
            new SelectListItem { Value = "DE", Text = "Germany" }
        };
    }
}
```

及以下视图：

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

则会生成以下 HTML：

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
        multiple="multiple"
        name="CountryCodes"><option value="MX">Mexico</option>
    <option value="CA">Canada</option>
    <option value="US">USA</option>
    <option value="FR">France</option>
    <option value="ES">Spain</option>
    <option value="DE">Germany</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

无选定内容

如果发现自己在多个页面中使用“未指定”选项，可创建模板用于消除重复的 HTML：

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

Views/Shared/EditorTemplates/CountryViewModel.cshtml 模板：

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

添加 HTML `<option>` 元素并不局限于无选定内容用例。例如，以下视图和操作方法会生成与上述代码类似的 HTML：

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

根据当前的 `Country` 值选择正确的 `<option>` 元素(包含 `selected="selected"` 属性)。

```
<form method="post" action="/Home/IndexEmpty">
    <select id="Country" name="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA" selected="selected">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
</form>
```

其他资源

- [标记帮助程序](#)
- [HTML Form 元素](#)
- [请求验证令牌](#)
- [模型绑定](#)
- [模型验证](#)
- [IAttributeAdapter 接口](#)
- [本文档的代码片段](#)

ASP.NET Core 内置标记帮助程序

2018/3/19 • 1 min to read • [Edit Online](#)

作者: [Peter Kellner](#)

ASP.NET Core 包括了许多内置标记帮助程序以用于提高生产力。本部分对内置标记帮助程序进行了概述。

注意

有些内置标记帮助程序未在此处讨论，因为它们由 [Razor](#) 视图引擎在内部使用。这包括针对扩展到网站根路径的 ~ 字符所适用的标记帮助程序。

内置 ASP.NET Core 标记帮助程序

[定位点标记帮助程序](#)

[缓存标记帮助程序](#)

[分布式缓存标记帮助程序](#)

[环境标记帮助程序](#)

[表单标记帮助程序](#)

[图像标记帮助程序](#)

[输入标记帮助程序](#)

[标签标记帮助程序](#)

[部分标记帮助程序](#)

[选择标记帮助程序](#)

[文本区标记帮助程序](#)

[验证消息标记帮助程序](#)

[验证摘要标记帮助程序](#)

其他资源

- [客户端开发](#)
- [标记帮助程序](#)

ASP.NET Core 中的定位点标记帮助程序

2018/5/8 • 7 min to read • [Edit Online](#)

作者: [Peter Kellner](#) 和 [Scott Addie](#)

[查看或下载示例代码](#) ([如何下载](#))

[定位点标记帮助程序](#) 可通过添加新属性来增强标准的 HTML 定位点 (`<a ... >`) 标记。按照约定，属性名称将使用前缀 `asp-`。`asp-` 属性的值决定呈现的定位点元素的 `href` 属性值。

本文档中的示例均使用 SpeakerController:

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

public class SpeakerController : Controller
{
    private List<Speaker> Speakers =
        new List<Speaker>
    {
        new Speaker {SpeakerId = 10},
        new Speaker {SpeakerId = 11},
        new Speaker {SpeakerId = 12}
    };

    [Route("Speaker/{id:int}")]
    public IActionResult Detail(int id) =>
        View(Speakers.FirstOrDefault(a => a.SpeakerId == id));

    [Route("/Speaker/Evaluations",
        Name = "speakerevals")]
    public IActionResult Evaluations() => View();

    [Route("/Speaker/EvaluationsCurrent",
        Name = "speakerevalscurrent")]
    public IActionResult Evaluations(
        int speakerId,
        bool currentYear) => View();

    public IActionResult Index() => View(Speakers);
}

public class Speaker
{
    public int SpeakerId { get; set; }
}
```

`asp-` 属性的清单如下所示。

asp-controller

`asp-controller` 属性可分配用于生成 URL 的控制器。下面的标记列出了所有发言人:

```
<a asp-controller="Speaker"
    asp-action="Index">All Speakers</a>
```

生成的 HTML：

```
<a href="/Speaker">All Speakers</a>
```

如果指定了 `asp-controller` 属性，而未指定 `asp-action` 属性，则默认的 `asp-action` 值为与当前正在执行的视图关联的控制器操作。如果前面的标记中省略了 `asp-action`，并在 `HomeController` 的索引视图 (/Home) 中使用了定位点标记帮助程序，则生成的 HTML 为：

```
<a href="/Home">All Speakers</a>
```

asp-action

`asp-action` 属性值表示生成的 `href` 属性中包含的控制器操作名称。下面的标记可将生成的 `href` 属性值设置为发言人评估页：

```
<a asp-controller="Speaker"  
    asp-action="Evaluations">Speaker Evaluations</a>
```

生成的 HTML：

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

如果未指定 `asp-controller` 属性，则使用默认控制器，该控制器调用执行当前视图的视图。

如果 `asp-action` 属性值为 `Index`，则不向 URL 追加任何操作，从而导致调用默认的 `Index` 操作。
`asp-controller` 引用的控制器中必须存在指定的(或默认的)操作。

asp-route-{value}

`asp-route-{value}` 属性可实现通配符路由前缀。占用 `{value}` 占位符的所有值都解释为潜在的路由参数。如果找不到默认路由，则将此路由前缀作为请求参数和值追加到生成的 `href` 属性。否则，将在路由模板中替换它。

考虑以下控制器操作：

```
public IActionResult AnchorTagHelper(int id)  
{  
    var speaker = new Speaker  
    {  
        SpeakerId = id  
    };  
  
    return View(speaker);  
}
```

在 `Startup.Configure` 中定义默认路由模板：

```

app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "areaRoute",
                    template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

```

MVC 视图使用操作提供的模型，如下所示：

```

@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-id="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>

```

默认路由的 `{id?}` 占位符得以匹配。生成的 HTML：

```
<a href="/Speaker/Detail/12">SpeakerId: 12</a>
```

假设路由前缀不属于自己匹配路由模板的一部分，如下面的 MVC 视图所示：

```

@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-speakerid="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>

```

生成以下 HTML，因为匹配的路由中未找到 `speakerid`：

```
<a href="/Speaker/Detail?speakerid=12">SpeakerId: 12</a>
```

如果 `asp-controller` 或 `asp-action` 均未指定，则会执行与 `asp-route` 属性中相同的默认处理。

asp-route

`asp-route` 属性用于创建直接链接到命名路由的 URL。使用[路由属性](#)，路由可以按 `SpeakerController` 中所示进行命名并用于其 `Evaluations` 操作：

```

[Route("/Speaker/Evaluations",
      Name = "speakerevals")]
public IActionResult Evaluations() => View();

```

在下列标记中，`asp-route` 属性引用命名路由：

```
<a asp-route="speakerevals">Speaker Evaluations</a>
```

定位点标记帮助程序使用 URL /Speaker/Evaluations 生成直接指向该控制器操作的路由。生成的 HTML：

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

如果除了 `asp-route`，还指定了 `asp-controller` 或 `asp-action`，则可能不会生成预期的路由。为了避免发生路由冲突，不应将 `asp-route` 与 `asp-controller` 和 `asp-action` 属性结合使用。

asp-all-route-data

`asp-all-route-data` 属性支持创建键值对字典。键是参数名称，值是参数值。

在下面的示例中，将对字典进行初始化并将其传递给 Razor 视图。或者，也可以使用模型传入数据。

```
@{  
    var parms = new Dictionary<string, string>  
    {  
        { "speakerId", "11" },  
        { "currentYear", "true" }  
    };  
  
<a asp-route="speakerevalscurrent"  
    asp-all-route-data="parms">Speaker Evaluations</a>
```

前面的代码生成以下 HTML：

```
<a href="/Speaker/EvaluationsCurrent?speakerId=11&currentYear=true">Speaker Evaluations</a>
```

平展 `asp-all-route-data` 字典，以生成满足重载 `Evaluations` 操作要求的查询字符串：

```
[Route("/Speaker/EvaluationsCurrent",  
      Name = "speakerevalscurrent")]  
public IActionResult Evaluations(  
    int speakerId,  
    bool currentYear) => View();
```

如果字典中的任何键匹配路由参数，则将根据需要在路由中替换这些值。其他不匹配的值作为请求参数生成。

asp-fragment

`asp-fragment` 属性可定义要追加到 URL 的 URL 片段。定位点标记帮助程序添加哈希字符 (#)。请考虑以下标记：

```
<a asp-controller="Speaker"  
    asp-action="Evaluations"  
    asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>
```

生成的 HTML：

```
<a href="/Speaker/Evaluations#SpeakerEvaluations">Speaker Evaluations</a>
```

生成客户端应用时，哈希标记很有用。它们可用于在 JavaScript 中轻松地执行标记和搜索等操作。

asp-area

`asp-area` 属性可设置用来设置相应路由的区域名称。以下示例展示了区域属性如何导致重新映射路由。如果将 `asp-area` 设置为“Blogs”，则会为此定位点标记的关联控制器和视图的路由添加目录 Areas/Blogs 作为前缀。

- <项目名称>
 - **wwwroot**
 - **区域**
 - **博客**
 - **控制器**
 - HomeController.cs
 - **视图**
 - **主文件夹**
 - AboutBlog.cshtml
 - Index.cshtml
 - _ViewStart.cshtml
 - **控制器**

鉴于上述目录层次结构，引用 `AboutBlog.cshtml` 文件的标记是：

```
<a asp-area="Blogs"
    asp-controller="Home"
    asp-action="AboutBlog">About Blog</a>
```

生成的 HTML：

```
<a href="/Blogs/Home/AboutBlog">About Blog</a>
```

提示

若要使区域在 MVC 应用中正常工作，路由模板必须包含对该区域（如果存在）的引用。该模板由 `Startup.Configure` 中 `routes.MapRoute` 方法调用的第二个参数表示：[!code-csharp]

asp-protocol

`asp-protocol` 属性用于在 URL 中指定协议（比如 `https`）。例如：

```
<a asp-protocol="https"
    asp-controller="Home"
    asp-action="About">About</a>
```

生成的 HTML：

```
<a href="https://localhost/Home/About">About</a>
```

示例中的主机名为 `localhost`，但在生成 URL 时，定位点标记帮助程序会使用网站的公共域。

asp-host

`asp-host` 属性用于在 URL 中指定主机名。例如:

```
<a asp-protocol="https"
    asp-host="microsoft.com"
    asp-controller="Home"
    asp-action="About">About</a>
```

生成的 HTML:

```
<a href="https://microsoft.com/Home/About">About</a>
```

asp-page

`asp-page` 属性适用于 Razor 页面。使用它向特定页设置定位点标记的 `href` 属性值。通过在页面名称前面使用正斜杠 (“/”) 作为前缀，可创建 URL。

下列示例指向与会者 Razor 页面:

```
<a asp-page="/Attendee">All Attendees</a>
```

生成的 HTML:

```
<a href="/Attendee">All Attendees</a>
```

`asp-page` 属性与 `asp-route`、`asp-controller` 和 `asp-action` 属性互斥。但是，`asp-page` 可与 `asp-route-{value}` 结合使用以控制路由，如以下标记所示:

```
<a asp-page="/Attendee"
    asp-route-attendeeid="10">View Attendee</a>
```

生成的 HTML:

```
<a href="/Attendee?attendeeid=10">View Attendee</a>
```

asp-page-handler

`asp-page-handler` 属性适用于 Razor 页面。它用于链接到特定的页处理程序。

请考虑以下页处理程序:

```
public void OnGetProfile(int attendeeId)
{
    ViewData["AttendeeId"] = attendeeId;

    // code omitted for brevity
}
```

页模型的关联标记链接到 `OnGetProfile` 页处理程序。注意，`asp-page-handler` 属性值中省略了页处理程序方法名称的 `On<Verb>` 前缀。如果这是异步方法，还会省略 `Async` 后缀。

```
<a asp-page="/Attendee"
    asp-page-handler="Profile"
    asp-route-attendeeid="12">Attendee Profile</a>
```

生成的 HTML：

```
<a href="/Attendee?attendeeid=12&handler=Profile">Attendee Profile</a>
```

其他资源

- [区域](#)
- [Razor 页面简介](#)

ASP.NET Core MVC 中的缓存标记帮助程序

2018/5/8 • 5 min to read • [Edit Online](#)

作者: Peter Kellner

缓存标记帮助程序通过将其内容缓存到内部 ASP.NET Core 缓存提供程序中，可极大地提高 ASP.NET Core 应用的性能。

Razor 视图引擎将默认的 `expires-after` 设置为 20 分钟。

以下 Razor 标记将缓存日期/时间：

```
<cache>@DateTime.Now</cache>
```

针对包含 `CacheTagHelper` 页面的第一个请求将显示当前的日期/时间。其他请求将显示已缓存的值，直到缓存过期(默认 20 分钟)或被内存压力逐出。

可使用以下属性设置缓存持续时间：

缓存标记帮助程序属性

enabled

属性类型	有效值
boolean	"true"(默认值)
	"false"

确定是否缓存了缓存标记帮助程序所包含的内容。默认值为 `true`。如果设置为 `false`，则此缓存标记帮助程序对于呈现的输出没有缓存效果。

示例:

```
<cache enabled="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-on

属性类型	示例值
DateTimeOffset	"@new DateTime(2025,1,29,17,02,0)"

设置一个绝对到期日期。以下示例将在 2025 年 1 月 29 日下午 5:02 之前缓存缓存标记帮助程序的内容。

示例:

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

属性类型	示例值
TimeSpan	"@TimeSpan.FromSeconds(120)"

设置从第一个请求时间到缓存内容的时间长度。

示例:

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-sliding

属性类型	示例值
TimeSpan	"@TimeSpan.FromSeconds(60)"

设置缓存条目在未被访问时应被逐出的时间。

示例:

```
<cache expires-sliding="@TimeSpan.FromSeconds(60)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-header

属性类型	示例值
String	"User-Agent"
	"User-Agent,content-encoding"

接受单个标头值或逗号分隔的标头值列表，在更改时触发缓存刷新。以下示例监视标头值 `User-Agent`。该示例将缓存提供给 Web 服务器的每个不同 `User-Agent` 的内容。

示例:

```
<cache vary-by-header="User-Agent">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-query

属性类型	示例值
String	"Make"
	"Make,Model"

接受单个标头值或逗号分隔的标头值列表，当标头值更改时触发缓存刷新。以下示例查看 `Make` 和 `Model` 的值。

示例：

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

属性类型	示例值
String	"Make"
	"Make,Model"

接受单个标头值或逗号分隔的标头值列表，当路由数据参数值更改时触发缓存刷新。示例：

Startup.cs

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

Index.cshtml

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

属性类型	示例值
String	".AspNetCore.Identity.Application"
	".AspNetCore.Identity.Application,HairColor"

接受单个标头值或逗号分隔的标头值列表，当标头值更改时触发缓存刷新。以下示例查看与 ASP.NET Identity 相关联的 cookie。当用户经过身份验证时，要设置的请求 cookie 将触发缓存刷新。

示例：

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-user

属性类型	示例值
Boolean	"true"
	"false"(默认值)

指定当已登录用户(或上下文主体)更改时是否应该重置缓存。当前用户也称为请求上下文主体，可通过引用

`@User.Identity.Name` 在 Razor 视图中查看。

以下示例查看当前登录的用户。

示例:

```
<cache vary-by-user="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

通过登录和注销周期，使用此属性将内容维护在缓存中。使用 `vary-by-user="true"` 时，登录和注销操作会使经过身份验证的用户的缓存无效。缓存无效，因为登录时会生成一个新的唯一 cookie 值。当 cookie 不存在或已过期时，则维持缓存以呈现匿名状态。这意味着如果没有用户登录，将维持缓存。

vary-by

属性类型	示例值
String	"@Model"

允许自定义缓存的数据。当属性的字符串值引用的对象发生更改时，会更新缓存标记帮助程序的内容。通常将模型值的字符串串联分配给此属性。从效果上看，这意味着更新任何已连接的值都会使缓存无效。

以下示例假定视图的控制器方法将两个路由参数 `myParam1` 和 `myParam2` 的整数值相加，并将其作为单个模型属性返回。当此总和更改时，会再次呈现并缓存缓存标记帮助程序的内容。

示例:

操作:

```
public IActionResult Index(string myParam1, string myParam2, string myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

Index.cshtml

```
<cache vary-by="@Model"">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

属性类型	示例值
CacheItemPriority	"High"
	"Low"
	"NeverRemove"
	"Normal"

为内置缓存提供程序提供缓存逐出指导。在内存压力下，Web 服务器将首先逐出 `Low` 缓存条目。

示例：

```
<cache priority="High">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

`priority` 属性并不能保证特定级别的缓存保留。`CacheItemPriority` 仅供参考。将此属性设置为 `NeverRemove` 并不能保证缓存将始终保留。有关详细信息，请参阅[其他资源](#)。

缓存标记帮助程序依赖于[内存缓存服务](#)。如果尚未添加该服务，缓存标记帮助程序将添加。

其他资源

- [内存中缓存](#)
- [标识简介](#)

ASP.NET Core 中的分布式缓存标记帮助程序

2018/5/8 • 2 min to read • [Edit Online](#)

作者: Peter Kellner

分布式缓存标记帮助程序将其内容缓存到分布式缓存源, 从而大幅提高 ASP.NET Core 应用的性能。

分布式缓存标记帮助程序与缓存标记帮助程序继承自相同的基类。与缓存标记帮助程序相关的所有属性也将适用于分布式标记帮助程序。

分布式缓存标记帮助程序遵循被称为“构造函数注入”的显式依赖关系原则。具体而言, 将 `IDistributedCache` 接口容器传递给分布式缓存标记帮助程序的构造函数。如果未在 `ConfigureServices` 中创建 `IDistributedCache` 的特定具体实现(通常在 `startup.cs` 中找到), 则分布式缓存标记帮助程序将使用与基本缓存标记帮助程序所用相同的内存中提供程序来存储缓存数据。

分布式缓存标记帮助程序属性

enabled expires-on expires-after expires-sliding vary-by-header vary-by-query vary-by-route vary-by-cookie vary-by-user vary-by priority

有关定义, 请参阅缓存标记帮助程序。分布式缓存标记帮助程序与缓存标记帮助程序继承自相同的类, 因此其属性都是缓存标记帮助程序中的常见属性。

name(必需)

属性类型	示例值
字符串	"my-distributed-cache-unique-key-101"

必需的 `name` 属性用作为分布式缓存标记帮助程序的每个实例存储的缓存的键。分布式缓存标记帮助程序与基础缓存标记帮助程序不同:后者基于 Razor 页面名称和 Razor 页面中标记帮助程序的位置, 向每个缓存标记帮助程序实例分配一个键;前者的键仅基于属性 `name`

用法示例:

```
<distributed-cache name="my-distributed-cache-unique-key-101">
    Time Inside Cache Tag Helper: @DateTime.Now
</distributed-cache>
```

分布式缓存标记帮助程序 `IDistributedCache` 实现

ASP.NET Core 中内置了 `IDistributedCache` 的两个实现。一个基于 Sql Server, 另一个基于 Redis。有关这些实现的详细信息, 请参阅下方引用的名为“使用分布式缓存”的资源。这两个实现都涉及在 ASP.NET Core 的 `startup.cs` 中设置 `IDistributedCache` 的实例。

没有专门与使用 `IDistributedCache` 的任何特定实现相关的标记属性。

其他资源

- [ASP.NET Core MVC 中的缓存标记帮助程序](#)

- [在 ASP.NET Core 依赖注入](#)
- [使用 ASP.NET Core 中分布式缓存](#)
- [缓存在内存中 ASP.NET 核心](#)
- [在 ASP.NET Core 上的标识简介](#)

ASP.NET Core 中的环境标记帮助程序

2018/2/8 • 1 min to read • [Edit Online](#)

作者: [Peter Kellner](#) 和 [Hisham Bin Ateya](#)

环境标记帮助程序根据当前宿主环境, 有条件地呈现其包含的内容。其单一属性 `names` 是一个以逗号分隔的环境名称列表, 如果其中的任何名称与当前环境匹配, 就会触发已包含内容的呈现。

环境标记帮助程序属性

名称

采用单个宿主环境名称或以逗号分隔的宿主环境名称列表, 用于触发已包含内容的呈现。

这些值与从 ASP.NET Core 静态属性 `HostingEnvironment.EnvironmentName` 返回的当前值进行比较。此值为以下值之一:**Staging**、**Development** 或 **Production**。比较不区分大小写。

有效的 `environment` 标记帮助程序示例为:

```
<environment names="Staging,Production">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

include 和 exclude 属性

ASP.NET Core 2.x 添加了 `include` & `exclude` 属性。这些属性基于已包括或已排除的宿主环境名称, 控制已包含内容的呈现。

include ASP.NET Core 2.0 及更高版本

`include` 属性的行为与 ASP.NET Core 1.0 中 `names` 属性的行为类似。

```
<environment include="Staging,Production">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

exclude ASP.NET Core 2.0 及更高版本

相反, `exclude` 属性允许 `EnvironmentTagHelper` 为指定名称以外的所有宿主环境名称呈现已包含内容。

```
<environment exclude="Development">
  <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

其他资源

- [在 ASP.NET Core 中使用多个环境](#)
- [在 ASP.NET Core 依赖注入](#)

ASP.NET Core 中的图像标记帮助程序

2018/5/8 • 1 min to read • [Edit Online](#)

作者:Peter Kellner

图像标记帮助程序可增强 `img` (``) 标记。它需要 `src` 标记以及 `boolean` 属性 `asp-append-version`。

如果图像源 (`src`) 是主机 Web 服务器上的静态文件，则会向图像源追加一个唯一缓存清除字符串作为查询参数。这可确保，如果主机 Web 服务器上的文件发生更改，将生成包含已更新请求参数的唯一请求 URL。缓存清除字符串是一个唯一值，表示静态图像文件的哈希。

如果图像源 (`src`) 不是静态文件(例如远程 URL, 或者服务器上不存在该文件)，则生成的 `` 标记的 `src` 属性不带缓存清除查询字符串参数。

图像标记帮助程序属性

asp-append-version

与 `src` 属性一起指定时，会调用图像标记帮助程序。

有效的 `img` 标记帮助程序示例为：

```
<img src "~/images/asplogo.png"
      asp-append-version="true" />
```

如果目录 `..wwwroot/images/asplogo.png` 中存在静态文件，生成的 html 与下面类似(哈希有所不同)：

```

```

分配给参数 `v` 的值是磁盘上的文件的哈希值。如果 Web 服务器无法获取对所引用的静态文件的读取访问权限，则不会向 `src` 属性添加 `v` 参数。

src

若要激活图像标记帮助程序，`` 元素需要有 `src` 属性。

注意

图像标记帮助程序使用本地 Web 服务器上的 `Cache` 提供程序来存储给定文件的已计算 `Sha512`。如果再次请求该文件，则不需要重新计算 `Sha512`。当计算该文件的 `Sha512` 时，附加到该文件的文件观察程序会让 `Cache` 失效。

其他资源

- [缓存在内存中 ASP.NET 核心](#)

ASP.NET Core 中的部分标记帮助程序

2018/5/14 • 3 min to read • [Edit Online](#)

作者: Scott Addie

注意

ASP.NET Core 2.1 是预览版，不建议用于生产环境。

[查看或下载示例代码\(如何下载\)](#)

概述

Partial 标记帮助程序用于在 Razor 页面和 MVC 应用中呈现**分部视图**。请考虑：

- 需要 ASP.NET Core 2.1 或更高版本。
- 是 [HTML 帮助程序语法](#)的替代方法。
- 以异步方式呈现分部视图。

用于呈现分部视图的 HTML 帮助程序选项包括：

- `@await Html.PartialAsync`
- `@await Html.RenderPartialAsync`
- `@Html.Partial`
- `@Html.RenderPartial`

本文档中的示例均使用产品模型：

```
namespace TagHelpersBuiltIn.Models
{
    public class Product
    {
        public int Number { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }
    }
}
```

以下是分部标记帮助程序属性的清单。

name

需要 `name` 属性。它指示要呈现的分部视图的名称或路径。提供分部视图名称时，会启动**视图发现进程**。提供显式路径时，将绕过该进程。

以下标记使用显式路径，指示要从共享文件夹加载 `_ProductPartial.cshtml`。使用 `for` 属性，将模型传递给分部视图进行绑定。

```
<partial name="Shared/_ProductPartial.cshtml"
    for="Product" />
```

for

`for` 属性分配要根据当前模型评估的 [ModelExpression](#)。`ModelExpression` 推断 `@Model.` 语法。例如，可使用 `for="Product"` 而非 `for="@Model.Product"`。通过使用 `@` 符号定义内联表达式来替代此默认推理行为。`for` 属性不能与 `model` 属性一起使用。

以下标记加载 `_ProductPartial.cshtml`：

```
<partial name="_ProductPartial"
    for="Product" />
```

分部视图绑定到关联页模型的 `Product` 属性：

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using TagHelpersBuiltIn.Models;

namespace TagHelpersBuiltIn.Pages
{
    public class ProductModel : PageModel
    {
        public Product Product { get; set; }

        public void OnGet()
        {
            Product = new Product
            {
                Number = 1,
                Name = "Test product",
                Description = "This is a test product"
            };
        }
    }
}
```

model

`model` 属性分配模型实例，以传递到分部视图。`model` 属性不能与 `for` 属性一起使用。

在以下标记中，实例化新的 `Product` 对象并将其传递给 `model` 属性进行绑定：

```
<partial name="_ProductPartial"
    model='new Product { Number = 1, Name = "Test product", Description = "This is a test" }' />
```

view-data

`view-data` 属性分配 [ViewDataDictionary](#)，以传递到分部视图。以下标记使整个 `ViewData` 集合可访问分部视图：

```
@{  
    ViewData["IsNumberReadOnly"] = true;  
}  
  
<partial name="_ProductViewDataPartial"  
    for="Product"  
    view-data="@ViewData" />
```

在前面的代码中，`IsNumberReadOnly` 键值设置为 `true` 并添加到 `ViewData` 集合中。因此，在以下分部视图中可访问 `ViewData["IsNumberReadOnly"]`：

```
@model TagHelpersBuiltIn.Models.Product  
  
<div class="form-group">  
    <label asp-for="Number"></label>  
    @if ((bool)ViewData["IsNumberReadOnly"])  
    {  
        <input asp-for="Number" type="number" class="form-control" readonly />  
    }  
    else  
    {  
        <input asp-for="Number" type="number" class="form-control" />  
    }  
</div>  
<div class="form-group">  
    <label asp-for="Name"></label>  
    <input asp-for="Name" type="text" class="form-control" />  
</div>  
<div class="form-group">  
    <label asp-for="Description"></label>  
    <textarea asp-for="Description" rows="4" cols="50" class="form-control"></textarea>  
</div>
```

在此示例中，`ViewData["IsNumberReadOnly"]` 的值确定 `Number` 字段是否显示为只读。

其他资源

- [部分视图](#)
- [弱类型数据\(ViewData 和 ViewBag\)](#)

ASP.NET Core 中的分部视图

2018/5/14 • 5 min to read • [Edit Online](#)

作者: [Steve Smith](#)、[Maher JENDOUBI](#)、[Rick Anderson](#) 和 [Scott Sauber](#)

ASP.NET Core MVC 支持分部视图，如果拥有想要在不同视图之间共享的网页的可重用部分，分部视图十分有用。

[查看或下载示例代码\(如何下载\)](#)

什么是分部视图？

分部视图是指在其他视图内呈现的视图。通过执行分部视图生成的 HTML 输出在调用视图(或父视图)中呈现。和视图一样，分部视图也使用 `.cshtml` 文件扩展名。

应在何时使用分部视图？

分部视图是将大型视图分解为较小组件的有效方法。它们可减少视图内容的重复并使视图元素得以重复使用。常见布局元素应在 `_Layout.cshtml` 中指定。非布局可重用内容可封装到分部视图中。

如果拥有由多个逻辑部分组成的复杂页面，将每个部分用作其自己的分部视图十分有用。页面的每个部分都可独立于页面的其余部分进行查看，且页面本身的视图也会变得更加简单，因为它仅包含整体页面结构以及用于呈现分部视图的调用。

提示: 在视图中遵守 **不要自我重复原则**。

声明分部视图

分部视图的创建方式与任何其他视图的创建方式类似:在 `Views` 文件夹内创建 `.cshtml` 文件。分部视图和常规视图之间没有语义差异，仅呈现方式不同。可拥有直接从控制器的 `ViewResult` 返回的视图，并可将同一视图用作分部视图。视图和分部视图的主要呈现方式差异在于分部视图不运行 `_ViewStart.cshtml`(而视图运行—有关 `_ViewStart.cshtml` 的详细信息，请参阅 [布局](#))。

引用分部视图

在视图页中，有多种方法可呈现分部视图。最佳做法是使用 `Html.PartialAsync`，它会返回 `IHtmlString` 并可通过为调用添加 `@` 前缀进行引用:

```
@await Html.PartialAsync("AuthorPartial")
```

可使用 `RenderPartialAsync` 呈现分部视图。此方法不返回结果;它将呈现的输出直接流式传输到响应。因为它不返回结果，所以必须在 Razor 代码块内调用它:

```
@{
    await Html.RenderPartialAsync("AuthorPartial");
}
```

由于它直接流式传输结果，`RenderPartialAsync` 在某些情况下可能会表现更佳。但是，建议使用 `PartialAsync`。

虽然存在 `Html.PartialAsync` (`Html.Partial`) 和 `Html.RenderPartialAsync` (`Html.RenderPartial`) 的同步等效

项，但不建议使用同步等效项，因为可能会出现死锁的情况。同步方法在将来版本中不可用。

注意

如果视图需要执行代码，建议模式为使用[视图组件](#)，而不要使用分部视图。

分部视图发现

引用分部视图时，可通过多种方式引用其位置：

```
// Uses a view in current folder with this name  
// If none is found, searches the Shared folder  
@await Html.PartialAsync("ViewName")  
  
// A view with this name must be in the same folder  
@await Html.PartialAsync("ViewName.cshtml")  
  
// Locate the view based on the application root  
// Paths that start with "/" or "~/ refer to the application root  
@await Html.PartialAsync("~/Views/Folder/ViewName.cshtml")  
@await Html.PartialAsync("/Views/Folder/ViewName.cshtml")  
  
// Locate the view using relative paths  
@await Html.PartialAsync("../Account/LoginPartial.cshtml")
```

可在不同视图文件夹中拥有具有相同名称的不同分部视图。按名称(不带文件扩展名)引用视图时，每个文件夹中的视图都会使用与其位于同一文件夹中的分部视图。还可指定要使用的默认分部视图，将其放在 *Shared* 文件夹中。任何没有属于自己的分部视图的视图则可使用共享分部视图。可设置默认分部视图(位于 *Shared* 中)，该视图被与父视图位于同一文件夹并具有相同名称的分部视图替代。

分部视图可链接。也就是说，分部视图可调用其他分部视图(只要未创建循环)。在每个视图或分部视图内，相对路径始终相对于该视图，而不相对于根视图或父视图。

注意

如果在分部视图中声明 [Razor section](#)，它不会对其父对象可见；它会局限于分部视图。

通过分部视图访问数据

实例化分部视图时，它会获得父视图的 `ViewData` 字典的副本。在分部视图内对数据所做的更新不会保存到父视图中。在分部视图中更改的 `ViewData` 会在分部视图返回时丢失。

可将 `ViewDataDictionary` 的实例传递到分部视图：

```
@await Html.PartialAsync("PartialName", custom ViewData)
```

还可将模型传入分部视图。该模型可以是页面的视图模型或自定义对象。可将模型传递到 `PartialAsync` 或 `RenderPartialAsync`：

```
@await Html.PartialAsync("PartialName", viewModel)
```

可将 `ViewDataDictionary` 的实例和视图模型传递到分部视图：

```
@await Html.PartialAsync("ArticleSection", section,
    new ViewDataDictionary(this.ViewData) { { "index", index } })
```

以下标记显示包含两个分部视图的 `Views/Articles/Read.cshtml` 视图。第二个分部视图将模型和 `ViewData` 传入分部视图。如果使用下方突出显示的 `ViewDataDictionary` 的构造函数重载，则可在传递新 `ViewData` 字典的同时保留现有的 `ViewData`：

```
@using Microsoft.AspNetCore.Mvc.ViewFeatures
@using PartialViewsSample.ViewModels
@model Article

<h2>@Model.Title</h2>
@*Pass the authors name to Views\Shared\AuthorPartial.cshtml*@
@await Html.PartialAsync("AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@*Loop over the Sections and pass in a section and additional ViewData
   to the strongly typed Views\Articles\ArticleSection.cshtml partial view.*@
@{ var index = 0;
    @foreach (var section in Model.Sections)
    {
        @await Html.PartialAsync("ArticleSection", section,
            new ViewDataDictionary(this.ViewData) { { "index", index } })
        index++;
    }
}
```

`Views/Shared/AuthorPartial:`

```
@model string
<div>
    <h3>@Model</h3>
    This partial view came from /Views/Shared/AuthorPartial.cshtml.<br />
</div>
```

`ArticleSection` 分部视图：

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"] </h3>
<div>
    @Model.Content
</div>
```

在运行时，分部视图在父视图中呈现，而父视图本身在共享的 `_Layout.cshtml` 内呈现

The screenshot shows a web browser window titled "PartialViewsSample" with the URL "localhost:3501/articles". The page content is a partial view of the Gettysburg Address by Abraham Lincoln. The title "The Gettysburg Address" is displayed, followed by the author's name "Abraham Lincoln". A note indicates the view came from "/Views/Shared/AuthorPartial.cshtml" and was created on "11/19/1863 12:00:00 AM". The main text is divided into three sections: "Section One Index: 0", "Section Two Index: 1", and "Section Three Index: 2". Each section contains a portion of the speech.

The Gettysburg Address

Abraham Lincoln

This partial view came from /Views/Shared/AuthorPartial.cshtml.
11/19/1863 12:00:00 AM

Section One Index: 0

Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Section Two Index: 1

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

Section Three Index: 2

But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- this ground. The brave men, living and dead, who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to

在 ASP.NET Core 中将依赖项注入到视图

2018/5/14 • 4 min to read • [Edit Online](#)

作者 : Steve Smith

ASP.NET Core 支持将[依赖关系注入](#)到视图。这对于视图特定服务很有用，例如仅为填充视图元素所需的本地化或数据。应尽量在控制器和视图之间保持[问题分离](#)。视图显示的大部分数据应该从控制器传入。

[查看或下载示例代码\(如何下载\)](#)

简单示例

可使用 `@inject` 指令将服务注入到视图中。可将 `@inject` 看作向视图添加属性，并用 DI 填充该属性。

`@inject` 的语法: `@inject <type> <name>`

操作中的 `@inject` 示例:

```
 @using System.Threading.Tasks
 @using ViewInjectSample.Model
 @using ViewInjectSample.Model.Services
 @model IEnumerable<ToDoItem>
 @inject StatisticsService StatsService
 <!DOCTYPE html>
 <html>
 <head>
   <title>To Do Items</title>
 </head>
 <body>
   <div>
     <h1>To Do Items</h1>
     <ul>
       <li>Total Items: @StatsService.GetCount()</li>
       <li>Completed: @StatsService.GetCompletedCount()</li>
       <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
     </ul>
     <table>
       <thead>
         <tr>
           <th>Name</th>
           <th>Priority</th>
           <th>Is Done?</th>
         </tr>
       </thead>
       <tbody>
         @foreach (var item in Model)
         {
           <tr>
             <td>@item.Name</td>
             <td>@item.Priority</td>
             <td>@item.IsDone</td>
           </tr>
         }
       </tbody>
     </table>
   </div>
 </body>
 </html>
```

此视图显示 `ToDoItem` 实例的列表，以及显示总体统计信息的摘要。摘要从已注入的 `StatisticsService` 中填充。在 `Startup.cs` 的 `ConfigureServices` 中为依赖关系注入注册此服务：

```
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
    services.AddTransient<StatisticsService>();
    services.AddTransient<ProfileOptionsService>();
```

`StatisticsService` 通过存储库访问 `ToDoItem` 实例集并执行某些计算：

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

示例存储库使用内存中集合。建议不将上示实现(对内存中的所有数据进行操作)用于远程访问的大型数据集。

该示例显示绑定到视图的模型数据以及注入到视图中的服务：

To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name Priority Is Done?

Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

填充查找数据

视图注入可用于填充 UI 元素(如下拉列表)中的选项。请考虑这样的用户个人资料窗体，其中包含用于指定性别、状态和其他首选项的选项。使用标准 MVC 方法呈现这样的窗体，需让控制器为每组选项请求数据访问服务，然后用要绑定的每组选项填充模型或 `ViewBag`。

另一种方法是将服务直接注入视图以获取选项。这最大限度地减少了控制器所需的代码量，将此视图元素构造逻辑移入视图本身。显示个人资料编辑窗体的控制器操作只需要传递个人资料实例的窗体：

```
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers
{
    public class ProfileController : Controller
    {
        [Route("Profile")]
        public IActionResult Index()
        {
            // TODO: look up profile based on logged-in user
            var profile = new Profile()
            {
                Name = "Steve",
                FavColor = "Blue",
                Gender = "Male",
                State = new State("Ohio", "OH")
            };
            return View(profile);
        }
    }
}
```

用于更新这些首选项的 HTML 窗体包括三个属性的下拉列表：

Update Profile

Name:

Gender:

State:

Fav. Color:

这些列表由已注入视图的服务填充：

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
        Options.ListGenders().Select(g =>
            new SelectListItem() { Text = g, Value = g }))<br/>
    State: @Html.DropDownListFor(m => m.State.Code,
        Options.ListStates().Select(s =>
            new SelectListItem() { Text = s.Name, Value = s.Code}))<br />
    Fav. Color: @Html.DropDownList("FavColor",
        Options.ListColors().Select(c =>
            new SelectListItem() { Text = c, Value = c }))</div>
</body>
</html>
```

ProfileOptionsService 是 UI 级别的服务，旨在准确提供此窗体所需的数据：

```
using System.Collections.Generic;

namespace ViewInjectSample.Model.Services
{
    public class ProfileOptionsService
    {
        public List<string> ListGenders()
        {
            // keeping this simple
            return new List<string>() {"Female", "Male"};
        }

        public List<State> ListStates()
        {
            // a few states from USA
            return new List<State>()
            {
                new State("Alabama", "AL"),
                new State("Alaska", "AK"),
                new State("Ohio", "OH")
            };
        }

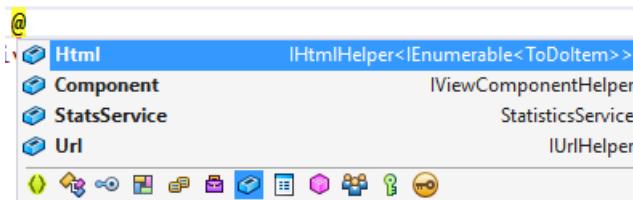
        public List<string> ListColors()
        {
            return new List<string>() { "Blue", "Green", "Red", "Yellow" };
        }
    }
}
```

提示

请记得在 `Startup.cs` 的 `ConfigureServices` 方法中注册要通过依赖关系注入请求的类型。

替代服务

除了注入新的服务之外，此方法也可用于替代以前在页面上注入的服务。下图显示了第一个示例中使用的页面上的所有可用字段：



如你所见，包括默认字段 `Html`、`Component` 和 `Url`（以及我们注入的 `StatsService`）。如果想用自己的 HTML 帮助程序替换默认的 HTML 帮助程序，可使用 `@inject` 轻松完成：

```
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
    <title>My Helper</title>
</head>
<body>
    <div>
        Test: @Html.Value
    </div>
</body>
</html>
```

如果想扩展现有的服务，用自己的方法继承或包装现有的实现时，只需使用此方法。

请参阅

- Simon Timms 的博客：[在视图中查找数据](#)

ASP.NET Core 中的视图组件

2018/5/14 • 10 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

[查看或下载示例代码\(如何下载\)](#)

视图组件简介

作为 ASP.NET Core MVC 的新功能, 视图组件与分部视图类似, 但它们的功能更加强大。视图组件不使用模型绑定, 并且仅依赖调用时提供的数据。视图组件:

- 呈现一个区块而不是整个响应。
- 包括控制器和视图间发现的相同关注点分离和可测试性优势。
- 可以有参数和业务逻辑。
- 通常从布局页调用。

视图组件可用于具有可重用呈现逻辑(对分部视图来说过于复杂)的任何位置, 例如:

- 动态导航菜单
- 标记云(查询数据库的位置)
- 登录面板
- 购物车
- 最近发布的文章
- 典型博客上的边栏内容
- 一个登录面板, 呈现在每页上并显示注销或登录链接, 具体取决于用户的登录状态

视图组件由两部分组成:类(通常派生自 [ViewComponent](#))及其返回的结果(通常为视图)。与控制器一样, 视图组件也可以是 POCO, 但大多数开发人员都希望利用派生自 [ViewComponent](#) 的可用方法和属性。

创建视图组件

本部分包含创建视图组件的高级别要求。本文后续部分将详细检查每个步骤并创建视图组件。

视图组件类

可通过以下任一方法创建视图组件类:

- 从 [ViewComponent](#) 派生
- 使用 [\[ViewComponent\]](#) 属性修饰类, 或者从具有 [\[ViewComponent\]](#) 属性的类派生
- 创建名称以 [ViewComponent](#) 后缀结尾的类

与控制器一样, 视图组件必须是公共、非嵌套和非抽象的类。视图组件名称是删除了“[ViewComponent](#)”后缀的类名。也可以使用 [ViewComponentAttribute.Name](#) 属性显式指定它。

视图组件类:

- 完全支持构造函数[依赖关系注入](#)
- 不参与控制器生命周期, 这意味着不能在视图组件中使用[筛选器](#)

视图组件方法

视图组件以返回 [InvokeAsync](#) 的 [IViewComponentResult](#) 方法定义其逻辑。参数直接来自视图组件的调用, 而不

是来自模型绑定。视图组件从不直接处理请求。通常，视图组件通过调用 `View` 方法来初始化模型并将其传递到视图。总之，视图组件方法：

- 定义返回 `IViewComponentResult` 的 `InvokeAsync` 方法
- 一般通过调用 `ViewComponent` `View` 方法来初始化模型并将其传递到视图
- 参数来自调用方法，而不是 HTTP，没有模型绑定
- 不可直接作为 HTTP 终结点访问，它们从代码调用（通常在视图中）。视图组件从不处理请求
- 在签名上重载，而不是当前 HTTP 请求的任何详细信息

视图搜索路径

运行时在以下路径中搜索视图：

- `Views/<controller_name>/Components/<view_component_name>/<view_name>`
- `Views/Shared/Components/<view_component_name>/<view_name>`

视图组件的默认视图名称为“默认”，这意味着视图文件通常命名为“Default.cshtml”。可以在创建视图组件结果或调用 `View` 方法时指定不同的视图名称。

建议将视图文件命名为“Default.cshtml”并使用

`Views/Shared/Components/<view_component_name>/<view_name>` 路径。此示例中使用的 `PriorityList` 视图组件对视图组件视图使用 `Views/Shared/Components/PriorityList/Default.cshtml`。

调用视图组件

要使用视图组件，请在视图中调用以下内容：

```
@Component.InvokeAsync("Name of view component", <anonymous type containing parameters>)
```

参数将传递给 `InvokeAsync` 方法。本文中开发的 `PriorityList` 视图组件是从 `Views/Todo/Index.cshtml` 视图文件中调用的。在下例中，使用两个参数调用 `InvokeAsync` 方法：

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

调用视图组件作为标记帮助程序

对于 ASP.NET Core 1.1 及更高版本，可以调用视图组件作为[标记帮助程序](#)：

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

标记帮助程序采用 Pascal 大小写格式的类和方法参数将转换为各自相应的小写短横线格式。要调用视图组件的标记帮助程序使用 `<vc></vc>` 元素。按如下方式指定视图组件：

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

注意：为了将视图组件用作标记帮助程序，必须使用 `@addTagHelper` 指令注册包含视图组件的程序集。例如，如果视图组件位于名为“MyWebApp”的程序集中，请将以下指令添加到 `_ViewImports.cshtml` 文件：

```
@addTagHelper *, MyWebApp
```

可将视图组件作为标记帮助程序注册到任何引用视图组件的文件。要详细了解如何注册标记帮助程序，请参阅[管理标记帮助程序作用域](#)。

本教程中使用的 `InvokeAsync` 方法：

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

在标记帮助程序标记中：

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

在以上示例中，`PriorityList` 视图组件变为 `priority-list`。视图组件的参数作为小写短横线格式的属性进行传递。

从控制器直接调用视图组件

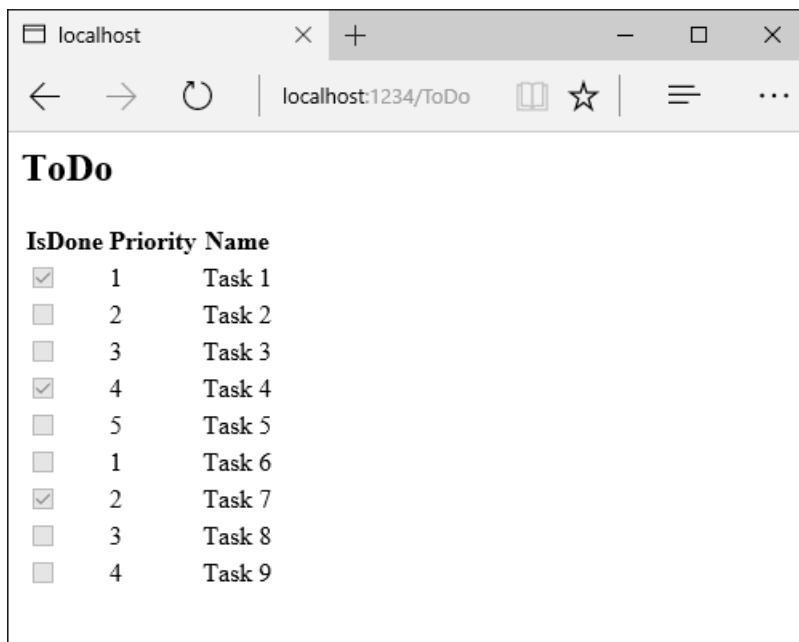
视图组件通常从视图调用，但你可以直接从控制器方法调用它们。尽管视图组件不定义控制器等终结点，但你可以轻松实现返回 `ViewComponentResult` 内容的控制器操作。

在此示例中，视图组件直接从控制器调用：

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

演练：创建简单的视图组件

[下载、生成和测试起始代码](#)。它是一个带有 `Todo` 控制器的简单项目，该控制器显示 Todo 项的列表。



添加 `ViewComponent` 类

创建一个 `ViewComponents` 文件夹并添加以下 `PriorityListViewComponent` 类：

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListViewComponent : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityListViewComponent(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
            int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }

        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                x.Priority <= maxPriority).ToListAsync();
        }
    }
}

```

代码说明：

- 视图组件类可以包含在项目的任意文件夹中。
- 因为类名 PriorityListViewComponent 以后缀 ViewComponent 结尾，所以运行时将在从视图引用类组件时使用字符串“PriorityList”。我稍后将进行详细解释。
- [ViewComponent] 属性可以更改用于引用视图组件的名称。例如，我们可以将类命名为 xyz 并应用 ViewComponent 属性：

```

[ViewComponent(Name = "PriorityList")]
public class XYZ : ViewComponent

```

- 上面的 [ViewComponent] 属性通知视图组件选择器在查找与组件相关联的视图时使用名称 PriorityList，以及在从视图引用类组件时使用字符串“PriorityList”。我稍后将进行详细解释。
- 组件使用 [依赖关系注入](#) 以使数据上下文可用。
- InvokeAsync 公开可以从视图调用的方法，且可以采用任意数量的参数。
- InvokeAsync 方法返回满足 isDone 和 maxPriority 参数的 ToDo 项集。

创建视图组件 Razor 视图

- 创建 Views/Shared/Components 文件夹。此文件夹必须命名为 Components。
- 创建 Views/Shared/Components/PriorityList 文件夹。此文件夹名称必须与视图组件类的名称或类名去掉后缀（如果遵照约定并在类名中使用了“ViewComponent”后缀）的名称相匹配。如果使用了 ViewComponent 属性，则类名称需要匹配指定的属性。
- 创建 Views/Shared/Components/PriorityList/Default.cshtml Razor 视图：[!code-cshtml]

Razor 视图获取并显示 `TodoItem` 列表。如果视图组件 `InvokeAsync` 方法不传递视图名称(如示例中所示), 则按照约定使用“默认”作为视图名称。在本教程后面部分, 我将演示如何传递视图名称。要替代特定控制器的默认样式, 请将视图添加到控制器特定的视图文件夹(例如 `Views/Todo/Components/PriorityList/Default.cshtml`)。

如果视图组件是控制器特定的, 则可将其添加到控制器特定的文件夹 (`Views/Todo/Components/PriorityList/Default.cshtml`)。

- 将包含优先级列表组件调用的 `div` 添加到 `Views/Todo/index.cshtml` 文件底部:

```
</table>
<div>
    @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
</div>
```

标记 `@await Component.InvokeAsync` 显示调用视图组件的语法。第一个参数是要调用的组件的名称。后续参数将传递给该组件。`InvokeAsync` 可以采用任意数量的参数。

测试应用。下图显示 ToDo 列表和优先级项:

The screenshot shows a web browser window titled "localhost" at "localhost:1235". The main content area has a heading "ToDo". Below it is a table with columns "IsDone", "Priority", and "Name". The data is as follows:

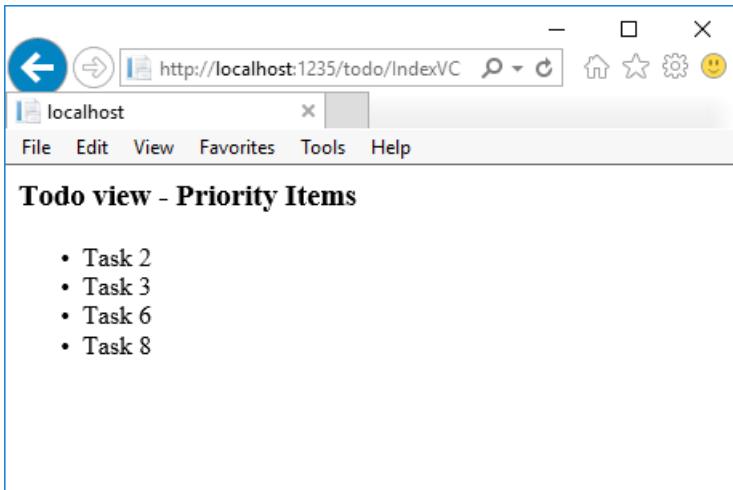
IsDone	Priority	Name
✓	1	Task 1
✗	2	Task 2
✗	3	Task 3
✓	4	Task 4
✗	5	Task 5
✗	1	Task 6
✓	2	Task 7
✗	3	Task 8
✗	4	Task 9

Below the table, there is a section titled "Todo view - Priority Items" containing the following list:

- Task 2
- Task 6

也可直接从控制器调用视图组件:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```



指定视图名称

在某些情况下，复杂的视图组件可能需要指定非默认视图。以下代码显示如何从 `InvokeAsync` 方法指定“PVC”视图。更新 `PriorityListViewComponent` 类中的 `InvokeAsync` 方法。

```
public async Task<IViewComponentResult> InvokeAsync(
    int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

将 `Views/Shared/Components/PriorityList/Default.cshtml` 文件复制到名为 `Views/Shared/Components/PriorityList/PVC.cshtml` 的视图。添加标题以指示正在使用 PVC 视图。

```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

更新 `Views/TodoList/Index.cshtml`:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

运行应用并验证 PVC 视图。

IsDone	Priority	Name
<input checked="" type="checkbox"/>	1	Task 1
<input type="checkbox"/>	2	Task 2
<input type="checkbox"/>	3	Task 3
<input checked="" type="checkbox"/>	4	Task 4
<input type="checkbox"/>	5	Task 5
<input type="checkbox"/>	1	Task 6
<input checked="" type="checkbox"/>	2	Task 7
<input type="checkbox"/>	3	Task 8
<input type="checkbox"/>	4	Task 9

PVC Named Priority Component View

- Task 1
- Task 4
- Task 7

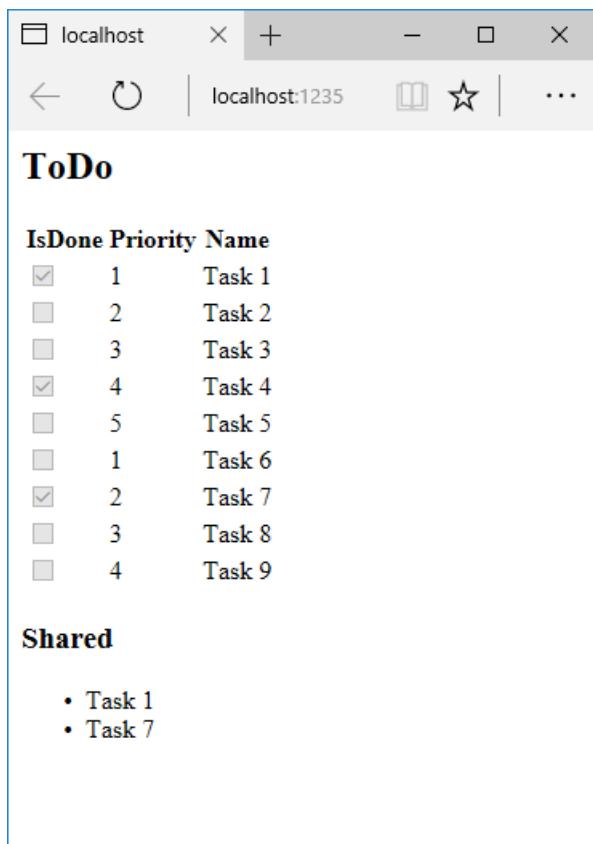
如果不呈现 PVC 视图, 请验证是否调用优先级为 4 或更高的视图组件。

检查视图路径

- 将优先级参数更改为 3 或更低, 从而不返回优先级视图。
- 将 Views/Todo/Components/PriorityList/Default.cshtml 暂时重命名为 1Default.cshtml。
- 测试应用, 你将收到以下错误:

```
An unhandled exception occurred while processing the request.  
InvalidOperationException: The view 'Components/PriorityList/Default' wasn't found. The following  
locations were searched:  
/Views/Todo/Components/PriorityList/Default.cshtml  
/Views/Shared/Components/PriorityList/Default.cshtml  
EnsureSuccessful
```

- 将 Views/Todo/Components/PriorityList/1Default.cshtml 复制到 Views/Shared/Components/PriorityList/Default.cshtml。
- 将一些标记添加到共享 Todo 视图组件视图以指示视图来自“Shared”文件夹。
- 测试“共享”组件视图。



避免魔幻字符串

若要确保编译时的安全性，可以用类名替换硬编码的视图组件名称。创建没有“ViewComponent”后缀的视图组件：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityList : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityList(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
            int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }

        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

将 `using` 语句添加到 Razor 视图文件，并使用 `nameof` 运算符：

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>
<!-- Markup removed for brevity. -->

<div>

    @await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
</div>
```

其他资源

- [视图中的依赖关系注入](#)

在 ASP.NET Core MVC 中使用控制器处理请求

2018/5/14 • 6 min to read • [Edit Online](#)

作者: [Steve Smith](#) 和 [Scott Addie](#)

控制器、操作和操作结果是开发人员如何使用 ASP.NET Core MVC 生成应用的一个基本组成部分。

什么是控制器？

控制器用于对一组操作进行定义和分组。操作(或操作方法)是控制器上一种用来处理请求的方法。控制器按逻辑将类似的操作集合到一起。通过这种操作的聚合，可以共同应用路由、缓存和授权等通用规则集。请求通过[路由](#)映射到操作。

根据惯例，控制器类：

- 驻留在项目的根级别“Controllers”文件夹中
- 继承自 `Microsoft.AspNetCore.Mvc.Controller`

控制器是一个可实例化的类，其中下列条件至少某一个为 true：

- 类名称带有“Controller”后缀
- 该类继承自带有“Controller”后缀的类
- 使用 `[Controller]` 属性修饰该类

控制器类不可含有关联的 `[NonController]` 属性。

控制器应遵循 [Explicit Dependencies Principle](#)(显式依赖关系原则)。以下几种方法可以实现此原则。如果多个控制器操作需要相同的服务，请考虑使用[构造函数注入](#)来请求这些依赖关系。如果该服务仅需要一个操作方法，请考虑使用[操作注入](#)来请求依赖关系。

在“模型-视图-控制器”模式中，控制器负责请求的初始处理和模型的实例化操作。通常情况下，应在模型中执行业务决策。

控制器获取模型处理的结果(如果有)，并返回正确的视图及其关联的视图数据或 API 调用的结果。请参阅 [ASP.NET Core MVC 概述](#) 以及 [ASP.NET Core MVC 和 Visual Studio 入门](#) 了解详细信息。

控制器是一个 UI 级别的抽象。它的职责是确保请求的数据有效，并选择应当返回的视图(或 API 的结果)。在构造良好的应用程序中，它不会直接包括数据访问或业务逻辑。相反，控制器会委托给处理这些责任的服务。

定义操作

控制器上的公共方法(除了那些使用 `[NonAction]` 属性装饰的方法)均为操作。操作上的参数会绑定到请求数据，并使用[模型绑定](#)进行验证。所有模型绑定的内容都会执行模型验证。`ModelState.IsValid` 属性值指示模型绑定和验证是否成功。

操作方法应包含用于将请求映射到某个业务关注点的逻辑。业务关注点通常应当表示为控制器通过[依赖关系注入](#)来访问的服务。然后，操作将业务操作的结果映射到应用程序状态。

操作可以返回任何内容，但是经常返回生成响应的 `IActionResult` (或异步方法的 `Task<IActionResult>`) 的实例。操作方法负责选择响应的类型。操作结果会做出响应。

控制器帮助程序方法

控制器通常继承自[控制器](#)(尽管没有要求)。派生自 `Controller` 会提供对三个帮助程序方法类别的访问：

1. 导致空响应正文的方法

没有包含 `Content-Type` HTTP 响应标头，因为响应正文缺少要描述的内容。

该类别中有两种结果类型：重定向和 HTTP 状态代码。

• HTTP 状态代码

此类型返回 HTTP 状态代码。此类型的几种帮助程序方法是 `BadRequest`、`NotFound` 和 `Ok`。例如，
`return BadRequest();` 执行时生成 400 状态代码。重载 `BadRequest``、`NotFound` 和 `Ok` 等方法时，它们不再符合 HTTP 状态代码响应方的资格，因为正在进行内容协商。

• 重定向

此类型（使用 `Redirect``、`LocalRedirect``、`RedirectToAction` 或 `RedirectToRoute`）返回一个到操作或目标的重定向。例如，`return RedirectToAction("Complete", new {id = 123});` 重定向到 `Complete`，传递一个匿名对象。

重定向结果类型与 HTTP 状态代码类型的不同之处主要在于 `Location` HTTP 响应标头的添加。

2. 导致含有预定义内容类型的非空响应正文的方法

此类别中的大多数帮助程序方法都包含一个 `ContentType` 属性，通过它可以设置 `Content-Type` 响应标头来描述响应正文。

此类别中有两种结果类型：[视图](#) 和 [已格式化的响应](#)。

• 视图

此类型返回一个使用模型呈现 HTML 的视图。例如，`return View(customer);` 将模型传递给视图以进行数据绑定。

• 已格式化的响应

此类型返回 JSON 或类似的数据交换格式，从而以特定方式表示某个对象。例如，`return Json(customer);` 将提供的对象串行化为 JSON 格式。

此类型的其他常见方法包括 `File`、`PhysicalFile` 和 `VirtualFile`。例如，
`return PhysicalFile(customerFilePath, "text/xml");` 返回由 `Content-Type` 响应标头值“text/xml”所描述的 XML 文件。

3. 导致在与客户端协商的内容类型中格式化为非空响应正文的方法

此类别更为熟知的说法是“内容协商”。每当操作返回 `ObjectResult` 类型或除 `IActionResult` 实现之外的任何实现时，会应用[内容协商](#)。返回非 `IActionResult` 实现（例如 `object`）的操作也会返回已格式化的响应。

此类型的一些帮助程序方法包括 `BadRequest`、`CreatedAtRoute` 和 `Ok`。这些方法的示例分别为
`return BadRequest(modelState);`、`return CreatedAtRoute("routename", values, newobject);` 和 `return Ok(value);`。
请注意，`BadRequest` 和 `Ok` 仅在传递了值的时候才执行内容协商；在没有传递值的情况下，它们充当 HTTP 状态码结果类型。另一方面，`CreatedAtRoute` 方法始终执行内容协商，因为它的重载均要求传递一个值。

横切关注点

应用程序通常会共享其部分工作流程。示例包括需要身份验证才能访问购物车的应用，或者在某些页面上缓存数据的应用。要在某个操作方法之前或之后执行逻辑，请使用筛选器。在横切关注点上使用[筛选器](#)可以减少重复，使它们遵循[不要自我重复 \(DRY\) 原则](#)。

可在控制器或操作级别上应用大多数筛选器属性（例如 `[Authorize]`），具体取决于所需的粒度级别。

错误处理和响应缓存通常是横切关注点：

- [处理错误](#)
- [响应缓存](#)

使用筛选器或自定义[中间件](#)可处理许多横切关注点。

在 ASP.NET Core 中路由到控制器操作

2018/5/14 • 37 min to read • [Edit Online](#)

作者: [Ryan Nowak](#) 和 [Rick Anderson](#)

ASP.NET Core MVC 使用路由中间件来匹配传入请求的 URL 并将它们映射到操作。路由在启动代码或属性中定义。路由描述应如何将 URL 路径与操作相匹配。它还用于在响应中生成送出的 URL(用于链接)。

操作既支持传统路由，也支持属性路由。通过在控制器或操作上放置路由可实现属性路由。有关详细信息，请参阅[混合路由](#)。

本文档将介绍 MVC 与路由之间的交互，以及典型的 MVC 应用如何使用各种路由功能。有关高级路由的详细信息，请参阅[路由](#)。

设置路由中间件

在 `Configure` 方法中，可能会看到与下面类似的代码：

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

在对 `UseMvc` 的调用中，`MapRoute` 用于创建单个路由，亦称 `default` 路由。大多数 MVC 应用使用带有模板的路由，与 `default` 路由类似。

路由模板 `"{controller=Home}/{action=Index}/{id?}"` 可以匹配诸如 `/Products/Details/5` 之类的 URL 路径，并通过对路径进行标记来提取路由值 `{ controller = Products, action = Details, id = 5 }`。MVC 将尝试查找名为 `ProductsController` 的控制器并运行 `Details` 操作：

```
public class ProductsController : Controller
{
    public IActionResult Details(int id) { ... }
}
```

请注意，在此示例中，当调用此操作时，模型绑定会使用值 `id = 5` 将 `id` 参数设置为 `5`。有关更多详细信息，请参阅[模型绑定](#)。

使用 `default` 路由：

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

路由模板：

- `{controller=Home}` 将 `Home` 定义为默认 `controller`
- `{action=Index}` 将 `Index` 定义为默认 `action`
- `{id?}` 将 `id` 定义为可选参数

默认路由参数和可选路由参数不必包含在 URL 路径中进行匹配。有关路由模板语法的详细说明，请参

阅读[路由模板参考](#)。

"{controller=Home}/{action=Index}/{id?}" 可以匹配 URL 路径 / 并生成路由值 { controller = Home, action = Index }。 controller 和 action 的值使用默认值, id 不生成值, 因为 URL 路径中没有相应的段。MVC 使用这些路由值选择 HomeController 和 Index 操作:

```
public class HomeController : Controller
{
    public IActionResult Index() { ... }
}
```

通过使用此控制器定义和路由模板, 将对以下任意 URL 路径执行 HomeController.Index 操作:

- /Home/Index/17
- /Home/Index
- /Home
- /

简便方法 `UseMvcWithDefaultRoute` :

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

可用于替换:

```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

`UseMvc` 和 `UseMvcWithDefaultRoute` 可向中间件管道添加 `RouterMiddleware` 的实例。MVC 不直接与中间件交互, 而是使用路由来处理请求。MVC 通过 `MvcRouteHandler` 实例连接到路由。`UseMvc` 内的代码与下面类似:

```
var routes = new RouteBuilder(app);

// Add connection to MVC, will be hooked up by calls to MapRoute.
routes.DefaultHandler = new MvcRouteHandler(...);

// Execute callback to register routes.
// routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");

// Create route collection and add the middleware.
app.UseRouter(routes.Build());
```

`UseMvc` 不直接定义任何路由, 它向 `attribute` 路由的路由集合添加占位符。重载 `UseMvc(Action<IRouteBuilder>)` 则允许用户添加自己的路由, 并且还支持属性路由。`UseMvc` 及其所有变体都会为属性路由添加占位符:无论如何配置 `UseMvc`, 属性路由始终可用。`UseMvcWithDefaultRoute` 定义默认路由并支持属性路由。[属性路由](#)部分提供了有关属性路由的更多详细信息。

传统路由

`default` 路由:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

是一种**传统路由**。将这种样式称为**传统路由**的原因在于，它为 URL 路径设立了一个约定：

- 第一个路径段映射到控制器名称
- 第二段映射到操作名称。
- 第三段用于可选 `id` (用于映射到模型实体)

使用此 `default` 路由时，URL 路径 `/Products/List` 映射到 `ProductsController.List` 操作，
`/Blog/Article/17` 映射到 `BlogController.Article`。此映射仅基于控制器和操作名称，而不基于命名空间、源文件位置或方法参数。

提示

使用默认路由进行传统路由时，可快速生成应用程序，无需为所定义的每项操作提供一个新的 URL 模式。对于包含 CRUD 样式操作的应用程序，通过保持各控制器间 URL 的一致性，可帮助简化代码，使 UI 更易预测。

警告

路由模板将 `id` 定义为可选参数，意味着无需在 URL 中提供 ID 也可执行操作。从 URL 中省略 `id` 通常会导致模型绑定将它设置为 `0`，进而导致在数据库中找不到与 `id == 0` 匹配的实体。属性路由可以提供细化控制，使某些操作需要 ID，某些操作不需要 ID。按照惯例，当可选参数(比如 `id`)有可能在正确的用法中出现时，本文档将涵盖这些参数。

多个路由

通过添加对 `MapRoute` 的多次调用，可以在 `UseMvc` 内添加多个路由。这样做可以定义多个约定，或添加专用于特定操作的传统路由，比如：

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

此处的 `blog` 路由是一个**专用的传统路由**，这表示它使用传统路由系统，但专用于特定的操作。由于 `controller` 和 `action` 不会在路由模板中作为参数显示，它们只能有默认值，因此，此路由将始终映射到 `BlogController.Article` 操作。

路由集合中的路由会进行排序，并按添加顺序进行处理。因此，在此示例中，将先尝试 `blog` 路由，再尝试 `default` 路由。

注意

专用传统路由通常使用 catch-all 路由参数(比如 `{*article}`)来捕获 URL 路径的剩余部分。这会使某个路由变得“太贪婪”，也就是说，它会匹配用户想要使用其他路由来匹配的 URL。将“贪婪的”路由放在路由表中靠后的位置可解决此问题。

回退

在处理请求时，MVC 将验证路由值能否用于在应用程序中查找控制器和操作。如果路由值与任何操作

都不匹配，则将该路由视为不匹配，并尝试下一个路由。这称为回退，其目的是简化传统路由重叠的情况。

区分操作

当通过路由匹配到两项操作时，MVC 必须进行区分，以选择“最佳”候选项，否则会引发异常。例如：

```
public class ProductsController : Controller
{
    public IActionResult Edit(int id) { ... }

    [HttpPost]
    public IActionResult Edit(int id, Product product) { ... }
}
```

此控制器定义了两项操作，这两项操作均与 URL 路径 `/Products/Edit/17` 和路由数据 `{ controller = Products, action = Edit, id = 17 }` 匹配。这是 MVC 控制器的典型模式，其中 `Edit(int)` 显示用于编辑产品的表单，`Edit(int, Product)` 处理已发布的表单。为此，MVC 需要在请求为 HTTP `POST` 时选择 `Edit(int, Product)`，在 Http 谓词为任何其他内容时选择 `Edit(int)`。

`HttpPostAttribute` (`[HttpPost]`) 是 `IActionConstraint` 的实现，它仅允许执行当 Http 谓词为 `POST` 时选择的操作。`IActionConstraint` 的存在使 `Edit(int, Product)` 成为比 `Edit(int)` “更好”的匹配项，因此会先尝试 `Edit(int, Product)`。

只需在特殊化方案中编写自定义 `IActionConstraint` 实现，但务必了解 `HttpPostAttribute` 等属性的角色 — 为其他 Http 谓词定义了类似的属性。在传统路由中，当操作属于 `show form -> submit form` 工作流时通常使用相同的操作名称。在阅读[了解 IActionConstraint](#)部分后，此模式的便利性将变得更加明显。

如果匹配多个路由，但 MVC 找不到“最佳”路由，则会引发 `AmbiguousActionException`。

路由名称

以下示例中的字符串 `"blog"` 和 `"default"` 都是路由名称：

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

路由名称为路由提供一个逻辑名称，以便使用命名路由来生成 URL。路由排序会使 URL 生成复杂化，而这极大地简化了 URL 创建。路由名称必须在应用程序范围内唯一。

路由名称不影响请求的 URL 匹配或处理；它们仅用于 URL 生成。[路由](#) 提供了有关 URL 生成（包括 MVC 特定帮助程序中的 URL 生成）的更多详细信息。

属性路由

属性路由使用一组属性将操作直接映射到路由模板。在下面的示例中，`Configure` 方法使用 `app.UseMvc();`，不传递任何路由。`HomeController` 将匹配一组 URL，这组 URL 与默认路由 `{controller=Home}/{action=Index}/{id?}` 匹配的 URL 类似：

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult Index()
    {
        return View();
    }
    [Route("Home/About")]
    public IActionResult About()
    {
        return View();
    }
    [Route("Home/Contact")]
    public IActionResult Contact()
    {
        return View();
    }
}
```

将针对任意 URL 路径 `/`、`/Home` 或 `/Home/Index` 执行 `HomeController.Index()` 操作。

注意

此示例重点介绍属性路由与传统路由之间的主要编程差异。属性路由需要更多输入来指定路由；传统的默认路由处理路由的方式则更简洁。但是，属性路由允许（并需要）精确控制应用于每项操作的路由模板。

使用属性路由时，控制器名称和操作名称对于操作的选择没有影响。此示例匹配的 URL 与上一示例相同。

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    public IActionResult MyIndex()
    {
        return View("Index");
    }
    [Route("Home/About")]
    public IActionResult MyAbout()
    {
        return View("About");
    }
    [Route("Home/Contact")]
    public IActionResult MyContact()
    {
        return View("Contact");
    }
}
```

注意

上述路由模板未定义 `action`、`area` 和 `controller` 的路由参数。事实上，属性路由中不允许使用这些路由参数。由于路由模板已与某项操作关联，因此，分析 URL 中的操作名称毫无意义。

使用 `Http[Verb]` 属性的属性路由

属性路由还可以使用 `[Http[Verb]]` 属性，比如 `[HttpPostAttribute]`。所有这些属性都可采用路由模板。此示例展示与同一路由模板匹配的两项操作：

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
    // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
    // ...
}
```

对于诸如 `/products` 之类的 URL 路径，当 Http 谓词为 `GET` 时将执行 `ProductsApi.ListProducts` 操作，当 Http 谓词为 `POST` 时将执行 `ProductsApi.CreateProduct`。属性路由首先将 URL 与路由属性定义的路由模板集进行匹配。一旦某个路由模板匹配，就会应用 `IActionConstraint` 约束来确定可以执行的操作。

提示

生成 REST API 时，很少会在操作方法上使用 `[Route(...)]`。建议使用更特定的 `Http*Verb*Attributes` 来明确 API 所支持的操作。REST API 的客户端需要知道映射到特定逻辑操作的路径和 Http 谓词。

由于属性路由适用于特定操作，因此，使参数变成路由模板定义中的必需参数很简单。在此示例中，`id` 是 URL 路径中的必需参数。

```
public class ProductsApiController : Controller
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

将针对诸如 `/products/3` (而非 `/products`) 之类的 URL 路径执行 `ProductsApi.GetProduct(int)` 操作。请参阅[路由](#)了解路由模板和相关选项的完整说明。

路由名称

以下代码定义 `Products_List` 的路由名称：

```
public class ProductsApiController : Controller
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

可以使用路由名称基于特定路由生成 URL。路由名称不影响路由的 URL 匹配行为，仅用于生成 URL。路由名称必须在应用程序范围内唯一。

注意

这一点与传统的默认路由相反，后者将 `id` 参数定义为可选参数 (`{id?}`)。这种精确指定 API 的功能可带来一些好处，比如允许将 `/products` 和 `/products/5` 分派到不同的操作。

合并路由

若要使属性路由减少重复，可将控制器上的路由属性与各个操作上的路由属性合并。控制器上定义的所有路由模板均作为操作上路由模板的前缀。在控制器上放置路由属性会使控制器中的所有操作都使用属性路由。

```
[Route("products")]
public class ProductsApiController : Controller
{
    [HttpGet]
    public IActionResult ListProducts() { ... }

    [HttpGet("{id}")]
    public ActionResult GetProduct(int id) { ... }
}
```

在此示例中，URL 路径 `/products` 可以匹配 `ProductsApi.ListProducts`，URL 路径 `/products/5` 可以匹配 `ProductsApi.GetProduct(int)`。这两项操作仅匹配 HTTP `GET`，因为它们用 `HttpGetAttribute` 修饰。

应用于操作的以 `/` 开头的路由模板不与应用于控制器的路由模板合并。此示例匹配一组与默认路由类似的 URL 路径。

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]  
        // Combines to define the route template "Home"
    [Route("Index")]  
        // Combines to define the route template "Home/Index"
    [Route("/")]  
        // Doesn't combine, defines the route template ""
    public IActionResult Index()
    {
        ViewData["Message"] = "Home index";
        var url = Url.Action("Index", "Home");
        ViewData["Message"] = "Home index" + "var url = Url.Action; = " + url;
        return View();
    }

    [Route("About")]  
        // Combines to define the route template "Home/About"
    public IActionResult About()
    {
        return View();
    }
}
```

对属性路由排序

与按照已定义顺序执行的传统路由相反，属性路由会生成树，并同时匹配所有路由。其行为就像路由条目是以理想排序方式放置的一样；最特定的路由有机会比较一般的路由先执行。

例如，像 `blog/search/{topic}` 这样的路由比像 `blog/{*article}` 这样的路由更特定。从逻辑上讲，`blog/search/{topic}` 路由默认情况下先“运行”，因为这是唯一合理的排序。使用传统路由时，开发人员负责按所需顺序放置路由。

属性路由可以使用框架提供的所有路由属性的 `Order` 属性来配置顺序。路由按 `Order` 属性的升序进行处理。默认顺序为 `0`。使用 `Order = -1` 设置的路由比未设置顺序的路由先运行。使用 `Order = 1` 设置的路由在默认路由排序后运行。

提示

避免依赖 `Order`。如果 URL 空间需要有显式顺序值才能正确进行路由，则同样可能使客户端混淆不清。属性路由通常选择与 URL 匹配的正确路由。如果用于 URL 生成的默认顺序不起作用，使用路由名称作为替代项通常比应用 `Order` 属性更简单。

路由模板中的标记替换([controller]、[action]、[area])

为方便起见，属性路由支持标记替换，方法是将标记用大括号(`[]`)括起来。标记 `[action]`、`[area]` 和 `[controller]` 将替换为定义了路由的操作中的操作名称值、区域名称值和控制器名称值。在此示例中，操作可以与注释中所述的 URL 路径匹配：

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")]
    // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

标记替换发生在属性路由生成的最后一步。上述示例的行为方式将与以下代码相同：

```
public class ProductsController : Controller
{
    [HttpGet("[controller]/[action]")]
    // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("[controller]/[action]/{id}")]
    // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

属性路由还可以与继承结合使用。与标记替换结合使用时尤为强大。

```
[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
    [HttpGet] // Matches '/api/Products'
    public IActionResult List() { ... }

    [HttpPut("{id}")]
    // Matches '/api/Products/{id}'
    public IActionResult Edit(int id) { ... }
}
```

标记替换也适用于属性路由定义的路由名称。

`[Route("[controller]/[action]", Name="[controller]_[action]")]` 将为每项操作生成一个唯一的路由名

称。

若要匹配文本标记替换分隔符 [或]，可通过重复该字符([或])对其进行转义。

多个路由

属性路由支持定义多个访问同一操作的路由。此操作最常用于模拟默认传统路由的行为，如以下示例所示：

```
[Route("[controller]")]
public class ProductsController : Controller
{
    [Route("")]  
        // Matches 'Products'  
    [Route("Index")] // Matches 'Products/Index'  
    public IActionResult Index()
}
```

在控制器上放置多个路由属性意味着，每个路由属性将与操作方法上的每个路由属性合并。

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
    [HttpPost("Buy")]  
        // Matches 'Products/Buy' and 'Store/Buy'  
    [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'  
    public IActionResult Buy()
}
```

当在某个操作上放置多个路由属性(可实现 `IActionConstraint`)时，每个操作约束将与定义它的属性中的路由模板合并。

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpPut("Buy")]  
        // Matches PUT 'api/Products/Buy'  
    [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'  
    public IActionResult Buy()
}
```

提示

在操作上使用多个路由可能看起来很强大，但更建议使应用程序的 URL 空间保持简洁且定义完善。仅在需要时，例如为了支持现有客户端，才在操作上使用多个路由。

指定属性路由的可选参数、默认值和约束

属性路由支持使用与传统路由相同的内联语法，来指定可选参数、默认值和约束。

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
    // ...
}
```

有关路由模板语法的详细说明，请参阅[路由模板参考](#)。

使用 `IRouteTemplateProvider` 的自定义路由属性

该框架中提供的所有路由属性(`[Route(...)]`、`[HttpGet(...)]`等)都可实现 `IRouteTemplateProvider` 接

口。当应用启动时，MVC 会查找控制器类和操作方法上的属性，并使用可实现 `IRouteTemplateProvider` 的属性生成一组初始路由。

用户可以实现 `IRouteTemplateProvider` 来定义自己的路由属性。每个 `IRouteTemplateProvider` 都允许定义一个包含自定义路由模板、顺序和名称的路由：

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";

    public int? Order { get; set; }

    public string Name { get; set; }
}
```

应用 `[MyApiController]` 时，上述示例中的属性会自动将 `Template` 设置为 `"api/[controller]"`。

使用应用程序模型自定义属性路由

应用程序模型是一个在启动时创建的对象模型，MVC 可使用其中的所有元数据来路由和执行操作。应用程序模型包含从路由属性收集（通过 `IRouteTemplateProvider`）的所有数据。可通过编写约定在启动时修改应用程序模型，以便自定义路由的行为方式。此部分通过一个简单的示例说明了如何使用应用程序模型自定义路由。

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;
using System.Text;
public class NamespaceRoutingConvention : IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
            selector.AttributeRouteModel != null);
        if (hasRouteAttributes)
        {
            // This controller manually defined some routes, so treat this
            // as an override and not apply the convention here.
            return;
        }

        // Use the namespace and controller name to infer a route for the controller.
        //
        // Example:
        //
        // controller.ControllerTypeInfo -> "My.Application.Admin.UsersController"
        // baseNamespace -> "My.Application"
        //
        // template => "Admin/[controller]"
        //
        // This makes your routes roughly line up with the folder structure of your project.
        //

        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
            namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}

```

混合路由：属性路由与传统路由

MVC 应用程序可以混合使用传统路由与属性路由。通常将传统路由用于为浏览器处理 HTML 页面的控制器，将属性路由用于处理 REST API 的控制器。

操作既支持传统路由，也支持属性路由。通过在控制器或操作上放置路由可实现属性路由。不能通过传统路由访问定义属性路由的操作，反之亦然。控制器上的任何路由属性都会使控制器中的所有操作使用属性路由。

注意

这两种路由系统的区别在于 URL 与路由模板匹配后所应用的过程。在传统路由中，将使用匹配项中的路由值，从包含所有传统路由操作的查找表中选择操作和控制器。在属性路由中，每个模板都与某项操作关联，无需进行进一步的查找。

URL 生成

MVC 应用程序可以使用路由的 URL 生成功能，生成指向操作的 URL 链接。生成 URL 可消除硬编码 URL，使代码更稳定、更易维护。此部分重点介绍 MVC 提供的 URL 生成功能，并且仅涵盖 URL 生成工作原理的基础知识。有关 URL 生成的详细说明，请参阅[路由](#)。

`IUrlHelper` 接口用于生成 URL，是 MVC 与路由之间的基础结构的基础部分。在控制器、视图和视图组件中，可通过 `Url` 属性找到 `IUrlHelper` 的实例。

在此示例中，将通过 `Controller.Url` 属性使用 `IUrlHelper` 接口来生成指向另一项操作的 URL。

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return Content($"Go check out {url}, it's really great.");
    }

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

如果应用程序使用的是传统默认路由，则 `url` 变量的值将为 URL 路径字符串

`/UrlGeneration/Destination`。此 URL 路径由路由创建，方法是将当前请求中的路由值（环境值）与传递到 `Url.Action` 的值合并，并将这些值替换到路由模板中：

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

路由模板中的每个路由参数都会通过将名称与这些值和环境值匹配，来替换掉原来的值。没有值的路由参数如果有默认值，则可使用默认值；如果本身是可选参数（比如此示例中的 `id`），则可直接跳过。如果任何所需路由参数没有对应的值，URL 生成将失败。如果某个路由的 URL 生成失败，则尝试下一个路由，直到尝试所有路由或找到匹配项为止。

上面的 `Url.Action` 示例假定使用传统路由，但 URL 生成功能的工作方式与属性路由相似，只不过概念不同。在传统路由中，路由值用于扩展模板，`controller` 和 `action` 的路由值通常出现在该模板中——这种做法可行是因为通过路由匹配的 URL 遵守某项约定。在属性路由中，`controller` 和 `action` 的路由值不能出现在模板中，它们用于查找要使用的模板。

此示例使用属性路由：

```
// In Startup class
public void Configure(IApplicationBuilder app)
{
    app.UseMvc();
}
```

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination"); // Generates /custom/url/to/destination
        return Content($"Go check out {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination() {
        return View();
    }
}
```

MVC 生成一个包含所有属性路由操作的查找表，并匹配 `controller` 和 `action` 的值，以选择要用于生成 URL 的路由模板。在上述示例中，生成了 `custom/url/to/destination`。

根据操作名称生成 URL

`Url.Action` (`IUrlHelper` . `Action`) 以及所有相关重载都基于这样一种想法：用户想通过指定控制器名称和操作名称来指定要链接的内容。

注意

使用 `Url.Action` 时，将为用户指定 `controller` 和 `action` 的当前路由值，`controller` 和 `action` 的值是环境值和值的一部分。`Url.Action` 方法始终使用 `action` 和 `controller` 的当前值，并将生成将路由到当前操作的 URL 路径。

路由尝试使用环境值中的值来填充生成 URL 时未提供的信息。通过使用路由（比如 `{a}/{b}/{c}/{d}`）和环境值 `{ a = Alice, b = Bob, c = Carol, d = David }`，路由就具有足够的信息来生成 URL，而无需任何附加值，因为所有路由参数都有值。如果添加了值 `{ d = Donovan }`，则会忽略值 `{ d = David }`，生成的 URL 路径将为 `Alice/Bob/Carol/Donovan`。

警告

URL 路径是分层的。在上述示例中，如果添加了值 `{ c = Cheryl }`，则会忽略 `{ c = Carol, d = David }` 这两个值。在这种情况下，`d` 不再具有任何值，URL 生成将失败。用户需要指定 `c` 和 `d` 所需的值。使用默认路由 (`{controller}/{action}/{id?}`) 时可能会遇到此问题，但在实际操作中很少遇到此行为，因为 `Url.Action` 始终显式指定 `controller` 和 `action` 值。

较长的 `Url.Action` 重载还采用附加路由值对象，为 `controller` 和 `action` 以外的路由参数提供值。此重载最常与 `id` 结合使用，比如 `Url.Action("Buy", "Products", new { id = 17 })`。按照惯例，路由值对象通常是匿名类型的对象，但它也可以是 `IDictionary<>` 或普通旧.NET 对象。任何与路由参数不匹配的附加路由值都放在查询字符串中。

```
using Microsoft.AspNetCore.Mvc;

public class TestController : Controller
{
    public IActionResult Index()
    {
        // Generates /Products/Buy/17?color=red
        var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
        return Content(url);
    }
}
```

提示

若要创建绝对 URL, 请使用采用 `protocol` 的重载:

```
Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme)
```

根据路由生成 URL

上面的代码演示了如何通过传入控制器和操作名称来生成 URL。`IUrlHelper` 还提供 `Url.RouteUrl` 系列的方法。这些方法类似于 `Url.Action`, 但它们不会将 `action` 和 `controller` 的当前值复制到路由值。最常见的用法是指定一个路由名称, 以使用特定路由来生成 URL, 通常不指定控制器或操作名称。

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route"); // Generates /custom/url/to/destination
        return Content($"See {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination", Name = "Destination_Route")]
    public IActionResult Destination()
    {
        return View();
    }
}
```

在 HTML 中生成 URL

`IHtmlHelper` 提供 `HtmlHelper` 方法 `Html.BeginForm` 和 `Html.ActionLink`, 可分别生成 `<form>` 和 `<a>` 元素。这些方法使用 `Url.Action` 方法来生成 URL, 并且采用相似的参数。`HtmlHelper` 的配套 `Url.RouteUrl` 为 `Html.BeginRouteForm` 和 `Html.RouteLink`, 两者具有相似的功能。

`TagHelper` 通过 `form TagHelper` 和 `<a> TagHelper` 生成 URL。两者均通过 `IUrlHelper` 来实现。有关详细信息, 请参阅[使用表单](#)。

在视图内, 可通过 `Url` 属性将 `IUrlHelper` 用于前文未涵盖的任何临时 URL 生成。

在操作结果中生成 URL

以上示例展示了如何在控制器中使用 `IUrlHelper`, 不过, 控制器中最常见的用法是将 URL 生成为操作结果的一部分。

`ControllerBase` 和 `Controller` 基类为操作结果提供简便的方法来引用另一项操作。一种典型用法是在接受用户输入后进行重定向。

```
public Task<IActionResult> Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        return RedirectToAction("Index");
    }
}
```

操作结果工厂方法遵循与 `IUrlHelper` 上的方法类似的模式。

专用传统路由的特殊情况

传统路由可以使用一种特殊的路由定义，称为**专用传统路由**。在下面的示例中，名为 `blog` 的路由是一种**专用传统路由**。

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

使用这些路由定义，`Url.Action("Index", "Home")` 将通过 `default` 路由生成 URL 路径 `/`，但是，为什么会这样？用户可能认为使用 `blog` 路由值 `{ controller = Home, action = Index }` 就足以生成 URL，且结果为 `/blog?action=Index&controller=Home`。

专用传统路由依赖于不具有相应路由参数的默认值的特殊行为，以防止路由在 URL 生成过程中“太贪婪”。在此例中，默认值是为 `{ controller = Blog, action = Article }`，`controller` 和 `action` 均未显示为路由参数。当路由执行 URL 生成时，提供的值必须与默认值匹配。使用 `blog` 的 URL 生成将失败，因为值 `{ controller = Home, action = Index }` 与 `{ controller = Blog, action = Article }` 不匹配。然后，路由回退，尝试使用 `default`，并最终成功。

区域

区域是一种 MVC 功能，用于将相关功能整理到一个组中，作为单独的路由命名空间（用于控制器操作）和文件夹结构（用于视图）。通过使用区域，应用程序可以有多个名称相同的控制器，只要它们具有不同的区域。通过向 `controller` 和 `action` 添加另一个路由参数 `area`，可使用区域为路由创建层次结构。此部分将讨论路由如何与区域交互；有关如何将区域与视图结合使用的详细信息，请参阅[区域](#)。

下面的示例将 MVC 配置为使用默认传统路由和**区域路由**（用于名为 `Blog` 的区域）：

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

与 URL 路径（比如 `/Manage/Users/AddUser`）匹配时，第一个路由将生成路由值 `{ area = Blog, controller = Users, action = AddUser }`。`area` 路由值由 `area` 的默认值生成，事实上，通过 `MapAreaRoute` 创建的路由等效于以下路由：

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog" });
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

`MapAreaRoute` 通过为使用所提供的区域名称(本例中为 `Blog`)的 `area` 提供默认值和约束, 来创建路由。默认值确保路由始终生成 `{ area = Blog, ... }`, 约束要求在生成 URL 时使用值 `{ area = Blog, ... }`。

提示

传统路由依赖于顺序。一般情况下, 具有区域的路由应放在路由表中靠前的位置, 因为它们比没有区域的路由更特定。

在上面的示例中, 路由值将与以下操作匹配:

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

`AreaAttribute` 用于将控制器表示为某个区域的一部分, 比方说, 此控制器位于 `Blog` 区域中。没有 `[Area]` 属性的控制器不是任何区域的成员, 在路由提供 `area` 路由值时不匹配。在下面的示例中, 只有所列出的第一个控制器才能与路由值 `{ area = Blog, controller = Users, action = AddUser }` 匹配。

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

注意

出于完整性考虑，此处显示了每个控制器的命名空间，否则，控制器会发生命名冲突并生成编译器错误。类命名空间对 MVC 的路由没有影响。

前两个控制器是区域成员，仅在 `area` 路由值提供其各自的区域名称时匹配。第三个控制器不是任何区域的成员，只能在路由没有为 `area` 提供任何值时匹配。

注意

就不匹配任何值而言，缺少 `area` 值相当于 `area` 的值为 NULL 或空字符串。

在某个区域内执行某项操作时，`area` 的路由值将以环境值的形式提供，以便路由用于生成 URL。这意味着默认情况下，区域在 URL 生成中具有粘性，如以下示例所示。

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("duck_route", "Duck",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default", "Manage/{controller=Home}/{action=Index}/{id?}");
});
```

```

using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area
            var url = Url.Action("Index", "Home");
            // returns /Manage
            return Content(url);
        }

        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area
            var url = Url.Action("Index", "Home", new { area = "" });
            // returns /Manage/Home/Index
            return Content(url);
        }
    }
}

```

了解 `IActionConstraint`

注意

此部分深入介绍框架内部结构以及 MVC 如何选择要执行的操作。典型的应用程序不需要自定义

`IActionConstraint`

即使不熟悉 `IActionConstraint`，也可能已经用过该接口。`[HttpGet]` 属性和类似的 `[Http-VERB]` 属性可实现 `IActionConstraint` 来限制操作方法的执行。

```

public class ProductsController : Controller
{
    [HttpGet]
    public IActionResult Edit() { }

    public IActionResult Edit(...) { }
}

```

假定使用默认传统路由，URL 路径 `/Products/Edit` 将生成值 `{ controller = Products, action = Edit }`，这将匹配此处所示的两项操作。在 `IActionConstraint` 术语中，我们会说，这两项操作都视为候选项，因为它们都与该路由数据匹配。

当 `HttpGetAttribute` 执行时，它认为 `Edit()` 是 `GET` 的匹配项，而不是任何其他 `Http` 谓词的匹配项。`Edit(...)` 操作未定义任何约束，因此将匹配任何 `Http` 谓词。因此，假定 `Http` 谓词为 `POST`，则仅 `Edit(...)` 匹配。不过，对于 `GET`，这两项操作仍然都能匹配，只是具有 `IActionConstraint` 的操作始终被认为比没有该接口的操作更匹配。因此，由于 `Edit()` 具有 `[HttpGet]`，则认为它更特定，在两项操作都能匹配的情况下将选择它。

从概念上讲，`IActionConstraint` 是一种重载形式，但它并不重载具有相同名称的方法，而在匹配相同 URL 的操作之间重载。属性路由也使用 `IActionConstraint`，这可能会导致将不同控制器中的操作都视为候选项。

实现 `IActionConstraint`

实现 `IActionConstraint` 最简单的方法是创建派生自 `System.Attribute` 的类，并将其置于操作和控制器上。MVC 将自动发现任何应用为属性的 `IActionConstraint`。可使用应用程序模型应用约束，这可能是最灵活的一种方法，因为它允许对其应用方式进行元编程。

在下面的示例中，约束基于路由数据中的国家/地区代码选择操作。[GitHub 上的完整示例](#)。

```
public class CountrySpecificAttribute : Attribute, IActionConstraint
{
    private readonly string _countryCode;

    public CountrySpecificAttribute(string countryCode)
    {
        _countryCode = countryCode;
    }

    public int Order
    {
        get
        {
            return 0;
        }
    }

    public bool Accept(ActionConstraintContext context)
    {
        return string.Equals(
            context.RouteContext.RouteData.Values["country"].ToString(),
            _countryCode,
            StringComparison.OrdinalIgnoreCase);
    }
}
```

用户负责实现 `Accept` 方法，并为要执行的约束选择“顺序”。在此例中，当 `country` 路由值匹配时，`Accept` 方法返回 `true` 以表示该操作是匹配项。它与 `RouteValueAttribute` 的不同之处在于，它允许回退到非属性化操作。通过该示例可以了解到，如果定义 `en-US` 操作，则像 `fr-FR` 这样的国家/地区代码将回退到一个未应用 `[CountrySpecific(...)]` 的较通用的控制器。

`Order` 属性决定约束所属的阶段。操作约束基于 `Order` 分组运行。例如，该框架提供的所有 HTTP 方法属性均使用相同的 `Order` 值，以便在相同的阶段运行。用户可以按需设置阶段数来实现所需的策略。

提示

若要确定 `Order` 的值，请考虑是否应在 HTTP 方法前应用约束。数值较低的先运行。

ASP.NET Core 中的文件上传

2018/5/8 • 9 min to read • [Edit Online](#)

作者: Steve Smith

ASP.NET MVC 操作支持使用简单的模型绑定(针对较小文件)或流式处理(针对较大文件)上传一个或多个文件。

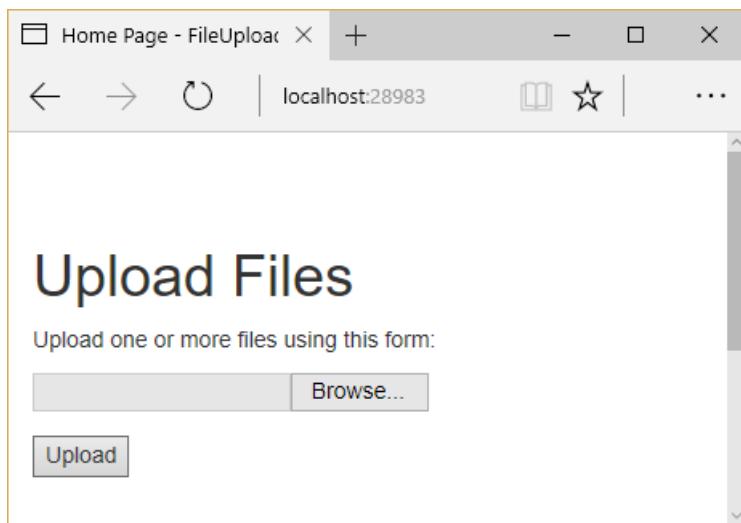
[查看或下载 GitHub 中的示例](#)

使用模型绑定上传小文件

要上传小文件, 可使用多部分 HTML 窗体或使用 JavaScript 构造 POST 请求。下方显示使用 Razor(支持上传多个文件)的示例窗体:

```
<form method="post" enctype="multipart/form-data" asp-controller="UploadFiles" asp-action="Index">
    <div class="form-group">
        <div class="col-md-10">
            <p>Upload one or more files using this form:</p>
            <input type="file" name="files" multiple />
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-10">
            <input type="submit" value="Upload" />
        </div>
    </div>
</form>
```

为支持文件上传, HTML 窗体必须指定 `multipart/form-data` 的 `enctype`。上面显示的 `files` 输入元素支持上传多个文件。忽略此输入元素上的 `multiple` 属性, 只允许上传单个文件。上述标记在浏览器中呈现为:



上传到服务器的单个文件可使用 `IFormFile` 接口通过 [模型绑定](#) 进行访问。`IFormFile` 具有以下结构:

```
public interface IFormFile
{
    string ContentType { get; }
    string ContentDisposition { get; }
    IHeaderDictionary Headers { get; }
    long Length { get; }
    string Name { get; }
    string FileName { get; }
    Stream OpenReadStream();
    void CopyTo(Stream target);
    Task CopyToAsync(Stream target, CancellationToken cancellationToken = null);
}
```

警告

切勿依赖或信任未经验证的 `FileName` 属性。`FileName` 属性应仅用于显示目的。

使用模型绑定和 `IFormFile` 接口上传文件时，操作方法可接受单个 `IFormFile` 或代表多个文件的 `IEnumerable<IFormFile>`（或 `List<IFormFile>`）。以下示例循环访问一个或多个上传的文件、将其保存到本地文件系统，并返回上传的文件总数和大小。

警告：下面的代码使用 `GetTempFileName`，该类会引发 `IOException` 如果超过 65535 个文件创建而不会删除以前的临时文件。实际的应用程序应删除临时文件或使用 `GetTempPath` 和 `GetRandomFileName` 创建临时文件的名称。65535 个文件限制为每个服务器，因此在服务器上的另一个应用程序可以使用 65535 的所有文件。

```
[HttpPost("UploadFiles")]
public async Task<IActionResult> Post(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    // full path to file in temp location
    var filePath = Path.GetTempFileName();

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            using (var stream = new FileStream(filePath, FileMode.Create))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size, filePath });
}
```

使用 `IFormFile` 技术上传的文件在处理之前会缓存在内存中或 Web 服务器的磁盘中。在操作方法中，`IFormFile` 内容可作为流访问。除了本地文件系统之外，还可将文件流式传输到 [Azure Blob 存储](#) 或 [实体框架](#)。

要使用实体框架将二进制文件数据存储在数据库中，请在实体上定义类型为 `byte[]` 属性：

```
public class ApplicationUser : IdentityUser
{
    public byte[] AvatarImage { get; set; }
}
```

指定类型为 `IFormFile` 的 viewmodel 属性：

```
public class RegisterViewModel
{
    // other properties omitted

    public IFormFile AvatarImage { get; set; }
}
```

注意

`IFormFile` 可直接用作操作方法参数或 viewmodel 属性，如上所示。

将 `IFormFile` 复制到流并将其保存到字节数组中：

```
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {
            UserName = model.Email,
            Email = model.Email
        };
        using (var memoryStream = new MemoryStream())
        {
            await model.AvatarImage.CopyToAsync(memoryStream);
            user.AvatarImage = memoryStream.ToArray();
        }
        // additional logic omitted

        // Don't rely on or trust the model.AvatarImage.FileName property
        // without validation.
    }
}
```

注意

在关系数据库中存储二进制数据时要格外小心，因为它可能对性能产生不利影响。

使用流式处理上传大文件

如果文件上传的大小或频率会导致应用出现资源问题，请考虑使用流式处理上传文件，而不是像如上所示的模型绑定方法那样全部缓冲文件。尽管使用 `IFormFile` 和模型绑定是更为简单的一种解决方案，但流式处理需要大量步骤才能正确实现。

注意

任何超过 64KB 的单个缓冲文件会从 RAM 移动到服务器磁盘上的临时文件中。文件上传所用的资源（磁盘、RAM）取决于并发文件上传的数量和大小。流式处理与性能没有太大的关系，而是与规模有关。如果尝试缓冲过多上传，站点就会在内存或磁盘空间不足时崩溃。

以下示例演示如何通过 JavaScript/Angular 来流式传输到控制器操作。使用自定义筛选器属性生成文件的防伪令牌

牌，并在 HTTP 头中(而不是在请求正文中)传递该令牌。由于操作方法直接处理上传的数据，所以其他筛选器会禁用模型绑定。在该操作中，使用 `MultipartReader` 读取窗体的内容，它会读取每个单独的 `MultipartSection`，从而根据需要处理文件或存储内容。读取所有节之后，该操作会执行自己的模型绑定。

初始操作加载窗体并将防伪令牌保存在 Cookie 中(通过 `GenerateAntiforgeryTokenCookieForAjax` 属性)：

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

该属性使用 ASP.NET Core 的内置[防伪](#)支持来设置包含请求令牌的 Cookie：

```
public class GenerateAntiforgeryTokenCookieForAjaxAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // We can send the request token as a JavaScript-readable cookie,
        // and Angular will use it by default.
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);
        context.HttpContext.Response.Cookies.Append(
            "XSRF-TOKEN",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }
}
```

Angular 会在名为 `X-XSRF-TOKEN` 的请求标头中自动传递防伪令牌。ASP.NET Core MVC 应用配置为在 `Startup.cs` 的配置中引用此标头：

```
public void ConfigureServices(IServiceCollection services)
{
    // Angular's default header name for sending the XSRF token.
    services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");

    services.AddMvc();
}
```

如下所示的 `DisableFormValueModelBinding` 属性用于禁用针对 `Upload` 操作方法的模型绑定。

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

由于已禁用模型绑定，因此 `Upload` 操作方法不接受参数。它直接使用 `ControllerBase` 的 `Request` 属性。

`MultipartReader` 用于读取每个节。该文件以 GUID 文件名保存，并且键/值数据存储在 `KeyValueAccumulator` 中。读取所有节之后，系统会使用 `KeyValueAccumulator` 的内容将窗体数据绑定到模型类型。

完整的 `Upload` 方法如下所示：

警告：下面的代码使用 `GetTempFileName`，该类会引发 `IOException` 如果超过 65535 个文件创建而不会删除以前的临时文件。实际的应用程序应删除临时文件或使用 `GetTempPath` 和 `GetRandomFileName` 创建临时文件的名称。65535 个文件限制为每个服务器，因此在服务器上的另一个应用程序可以使用 65535 的所有文件。

```
// 1. Disable the form value model binding here to take control of handling
// potentially large files.
// 2. Typically antiforgery tokens are sent in request body, but since we
// do not want to read the request body early, the tokens are made to be
// sent via headers. The antiforgery token filter first looks for tokens
// in the request header and then falls back to reading the body.
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Upload()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        return BadRequest($"Expected a multipart request, but got {Request.ContentType}");
    }

    // Used to accumulate all the form url encoded key value pairs in the
    // request.
    var formAccumulator = new KeyValueAccumulator();
    string targetFilePath = null;

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();
    while (section != null)
    {
        ContentDispositionHeaderValue contentDisposition;
        var hasContentDispositionHeader = ContentDispositionHeaderValue.TryParse(section.ContentDisposition,
            out contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper.HasFileContentDisposition(contentDisposition))
            {
                targetFilePath = Path.GetTempFileName();
                using (var targetStream = System.IO.File.Create(targetFilePath))
                {
                    await section.Body.CopyToAsync(targetStream);

                    _logger.LogInformation($"Copied the uploaded file '{targetFilePath}'");
                }
            }
            else if (MultipartRequestHelper.HasFormDataContentDisposition(contentDisposition))
            {
                // Content-Disposition: form-data; name="key"
                //
                // value

                // Do not limit the key name length here because the
                // multipart headers length limit is already in effect.
                var key = HeaderUtilities.RemoveQuotes(contentDisposition.Name);
                var encoding = GetEncoding(section);
                using (var streamReader = new StreamReader(
                    section.Body,
                    encoding))
            }
        }
    }
}
```

```

        encoding,
        detectEncodingFromByteOrderMarks: true,
        bufferSize: 1024,
        leaveOpen: true))
    {
        // The value length limit is enforced by MultipartBodyLengthLimit
        var value = await streamReader.ReadToEndAsync();
        if (String.Equals(value, "undefined", StringComparison.OrdinalIgnoreCase))
        {
            value = String.Empty;
        }
        formAccumulator.Append(key, value);

        if (formAccumulator.ValueCount > _defaultFormOptions.ValueCountLimit)
        {
            throw new InvalidDataException($"Form key count limit
{_defaultFormOptions.ValueCountLimit} exceeded.");
        }
    }
}

// Drains any remaining section body that has not been consumed and
// reads the headers for the next section.
section = await reader.ReadNextSectionAsync();
}

// Bind form data to a model
var user = new User();
var formValueProvider = new FormValueProvider(
    BindingSource.Form,
    new FormCollection(formAccumulator.GetResults()),
    CultureInfo.CurrentCulture);

var bindingSuccessful = await TryUpdateModelAsync(user, prefix: "",
    valueProvider: formValueProvider);
if (!bindingSuccessful)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
}

var uploadedData = new UploadedData()
{
    Name = user.Name,
    Age = user.Age,
    Zipcode = user.Zipcode,
    FilePath = targetFilePath
};
return Json(uploadedData);
}

```

疑难解答

以下是上传文件时遇到的一些常见问题及其可能的解决方案。

IIS 发生意外错误“找不到”

以下错误表示文件上传超过服务器配置的 `maxAllowedContentLength`：

```

HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.

```

默认设置为 `30000000` (大约 28.6MB)。可通过编辑 `web.config` 自定义该值：

```
<system.webServer>
  <security>
    <requestFiltering>
      <!-- This will handle requests up to 50MB -->
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

此设置仅适用于 IIS。在 Kestrel 上托管时，默认情况下不会出现此行为。有关详细信息，请参阅 [Request Limits](#) `<requestLimits>` (请求限制)。

IFormFile 的空引用异常

如果控制器正在接受使用 `IFormFile` 上传的文件，但你发现该值始终为 `NULL`，请确认 HTML 窗体指定的 `enctype` 值是否为 `multipart/form-data`。如果未在 `<form>` 元素上设置此属性，则不会发生文件上传，并且任何绑定的 `IFormFile` 参数都将为 `NULL`。

在 ASP.NET Core 中将依赖项注入到控制器

2018/5/14 • 6 min to read • [Edit Online](#)

作者: Steve Smith

ASP.NET Core MVC 控制器应该通过构造函数显式请求其依赖关系。在某些情况下，单独的控制器操作可能需要服务，但在控制器级别上请求可能没有意义。在此情况下，也可在操作方法上选择将服务作为参数注入。

[查看或下载示例代码\(如何下载\)](#)

依赖关系注入

依赖关系注入是遵循 [Dependency Inversion Principle](#)(依赖关系反向原则)的方法，允许应用程序由松散耦合的模块组成。ASP.NET Core 具有对[依赖关系注入](#)的内置支持，能更易于测试和维护应用程序。

构造函数注入

ASP.NET Core 对基于构造函数的依赖关系注入的内置支持扩展到 MVC 控制器。通过将服务类型作为构造函数参数添加到控制器中，ASP.NET Core 将尝试使用其内置的服务容器来解析该类型。通常(但不总是)使用接口来定义服务。例如，如果应用程序具有依赖于当前时间的业务逻辑，则可以注入一个检索时间(而不是对其进行硬编码)的服务，这将允许测试进入使用设置时间的实现。

```
using System;

namespace ControllerDI.Interfaces
{
    public interface IDateTime
    {
        DateTime Now { get; }
    }
}
```

实现这样一个接口，以便在运行时使用系统时钟，此操作很简单：

```
using System;
using ControllerDI.Interfaces;

namespace ControllerDI.Services
{
    public class SystemDateTime : IDateTime
    {
        public DateTime Now
        {
            get { return DateTime.Now; }
        }
    }
}
```

准备就绪后，我们可在控制器中使用服务。在此情况下，我们在 `HomeController` `Index` 方法中添加了一些逻辑，根据每天的时间向用户显示问候语。

```

using ControllerDI.Interfaces;
using Microsoft.AspNetCore.Mvc;

namespace ControllerDI.Controllers
{
    public class HomeController : Controller
    {
        private readonly IDateTime _dateTime;

        public HomeController(IDateTime dateTime)
        {
            _dateTime = dateTime;
        }

        public IActionResult Index()
        {
            var serverTime = _dateTime.Now;
            if (serverTime.Hour < 12)
            {
                ViewData["Message"] = "It's morning here - Good Morning!";
            }
            else if (serverTime.Hour < 17)
            {
                ViewData["Message"] = "It's afternoon here - Good Afternoon!";
            }
            else
            {
                ViewData["Message"] = "It's evening here - Good Evening!";
            }
            return View();
        }
    }
}

```

如果现在运行应用程序，很可能遇到错误：

```

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ControllerDI.Interfaces.IDateTime' while
attempting to activate 'ControllerDI.Controllers.HomeController'.
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type
requiredBy, Boolean isDefaultParameterRequired)

```

当我们尚未在 `Startup` 类的 `ConfigureServices` 方法中配置服务时，会发生此错误。若要指定应使用 `SystemDateTime` 的实例解析 `IDateTime` 的请求，请将下面列表中突出显示的行添加到 `ConfigureServices` 方法中：

```

public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}

```

注意

可以使用几种不同的生存期选项 (`Transient`、`Scoped` 或 `Singleton`) 中的任意一个来实现此特定服务。请参阅[依赖关系注入](#)，了解每个作用域选项将如何影响服务的行为。

一旦服务配置完毕，运行应用程序并导航到主页应按预期显示基于时间的消息：

A Message From The Server

It's afternoon here - Good Afternoon!

提示

请参阅[测试控制器逻辑](http://deviq.com/explicit-dependencies-principle/), 了解如何显式请求控制器中的依赖项 <http://deviq.com/explicit-dependencies-principle/> 可以更容易地测试代码。

ASP.NET Core 的内置依赖关系注入支持请求服务的类只拥有一个构造函数。如果有多个构造函数, 可能会收到如下异常消息:

```
An unhandled exception occurred while processing the request.  
  
InvalidOperationException: Multiple constructors accepting all given argument types have been found in type  
'ControllerDI.Controllers.HomeController'. There should only be one applicable constructor.  
Microsoft.Extensions.DependencyInjection.ActivatorUtilities.FindApplicableConstructor(Type instanceType,  
Type[] argumentTypes, ConstructorInfo& matchingConstructor, Nullable`1[] parameterMap)
```

如错误消息所述, 仅拥有一个构造函数可更正此问题。还可[将默认依赖关系注入支持替换为第三方实现](#), 其中许多可支持多个构造函数。

FromServices 的操作注入

有时, 控制器中不需要多个操作的服务。在此情况下, 将服务作为参数注入操作方法可能是有意义的。通过标记具有属性 `[FromServices]` 的参数来完成此操作, 如下所示:

```
public IActionResult About([FromServices] IDateTime dateTime)  
{  
    ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;  
  
    return View();  
}
```

从控制器访问设置

从控制器中访问应用程序或配置设置是一种常见模式。此访问应使用[配置](#)中所述的选项模式。通常不应使用依赖关系注入直接从控制器请求设置。最好请求 `IOptions<T>` 实例, 其中 `T` 是所需的配置类。

若要使用选项模式, 需要创建一个表示选项的类, 例如:

```
namespace ControllerDI.Model  
{  
    public class SampleWebSettings  
    {  
        public string Title { get; set; }  
        public int Updates { get; set; }  
    }  
}
```

然后, 需要将应用程序配置为使用选项模型, 并将配置类添加到 `ConfigureServices` 中的服务集合:

```

public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("samplewebsettings.json");
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; set; }

// This method gets called by the runtime. Use this method to add services to the container.
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    // Required to use the Options<T> pattern
    services.AddOptions();

    // Add settings from configuration
    services.Configure<SampleWebSettings>(Configuration);

    // Uncomment to add settings from code
    //services.Configure<SampleWebSettings>(settings =>
    //{
    //    settings.Updates = 17;
    //});

    services.AddMvc();

    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}

```

注意

在上面的列表中，我们要将应用程序配置为从 JSON 格式的文件中读取设置。也可以完全使用代码配置设置，如上面注释的代码所示。有关进一步的配置选项，请参阅[配置](#)。

一旦指定了强类型的配置对象（在本例中为 `SampleWebSettings`）并将其添加到服务集合中，就可通过请求 `IOptions<T>` 的实例，从任何 Controller 或 Action 方法请求它（在本例中为 `IOptions<SampleWebSettings>`）。以下代码显示了如何从控制器请求设置：

```

public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}

```

遵循选项模式，可将设置和配置相互分离，并确保控制器遵循 [separation of concerns](#)（问题分离），因为它不需要知道如何或在哪里找到设置信息。由于控制器类中没有 [static cling](#)（静态粘附）或设置类的直接实例化，因此控制器

更易于对[测试控制器逻辑](#)进行单元测试。

ASP.NET Core 中的测试控制器逻辑

2018/5/14 • 19 min to read • [Edit Online](#)

作者: [Steve Smith](#)

ASP.NET MVC 应用的控制器应该是小型的，且重点应对用户界面问题。处理非 UI 问题的大型控制器更难以测试和维护。

[查看或下载 GitHub 中的示例](#)

测试控制器

控制器是任意 ASP.NET Core MVC 应用程序的核心部分。因此，应该对它们按应用计划表现有信心。自动测试可以给你这种信心，还能在生成前检测错误。避免在控制器中安置不必要的职责并确保测试仅专注于控制器职责非常重要。

控制器应采用最简单的逻辑，而不应关注业务逻辑或基础结构问题(例如数据访问)。测试控制器逻辑，而不是框架。测试控制器基于有效或无效输入的行为。测试控制器如何相应其所执行业务操作的结果。

典型控制器职责：

- 验证 `ModelState.IsValid`。
- 如果 `ModelState` 无效，则返回错误响应。
- 从持久性检索业务实体。
- 对业务实体执行操作。
- 将业务实体保存到持久性。
- 返回相应的 `IActionResult`。

单元测试

[单元测试](#)涉及在测试应用程序部分时，使之与它的基础结构和依赖关系相隔离。单元测试控制器逻辑时，仅测试单个操作的内容，不测试其依赖项或框架自身的行为。单元测试控制器操作时，请确保仅关注其行为。控制器单元测试将避开[筛选器、路由或模型绑定](#)等。仅专注测试一项内容，单位测试通常编写简单、运行迅速。正确编写的单位测试集无需过多开销即可经常运行。但单元测试不检测组件间交互的问题，[集成测试](#)才会检测。

如果编写自定义筛选器、路由等，应对其进行单位测试，而不是作为特定控制器操作测试的一部分进行。应该对其进行隔离测试。

提示

[使用 Visual Studio 创建和运行单位测试。](#)

若要演示单位测试，请查看以下控制器。它显示集体讨论会话的列表并允许使用 POST 创建新的集体讨论会话：

```

using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public HomeController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index()
        {
            var sessionList = await _sessionRepository.ListAsync();

            var model = sessionList.Select(session => new StormSessionViewModel()
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
}

```

控制器遵循显式依赖关系原则，需要依赖关系注入提供 `IBrainstormSessionRepository` 实例。因此使用 mock 对象框架（例如 `Moq`）进行测试很简单。`HTTP GET Index` 方法没有循环或分支，且仅调用一个方法。若要测试此 `Index` 方法，我们需要验证返回了 `ViewResult`，附带来自存储库 `List` 方法的 `ViewModel`。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class HomeControllerTests
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);

            // Act
            var result = await controller.Index();

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
                viewResult.ViewData.Model);
            Assert.Equal(2, model.Count());
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
}

```

HomeController `HTTP POST Index` 方法(如上所示)应该验证:

- 操作方法在 `ModelState.IsValid` 是 `false` 时返回有相应数据的错误请求 `ViewResult`
- `ModelState.IsValid` 为 `true` 时, 调用存储库上的 `Add` 方法并返回有正确参数的 `RedirectToActionResult`。

通过使用 `AddModelError` 添加错误, 可以测试无效模型状态, 如下第一个测试所示。

```

[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}

```

第一个测试确认在 `ModelState` 无效时，返回与 `GET` 请求相同的 `ViewResult`。请注意，测试不会尝试传入无效模型。即使传入也无济于事，因为模型绑定不会运行（虽然[集成测试](#)将使用练习模型绑定）。在本例中，不测试模型绑定。这些单位测试仅测试操作方法中代码的行为。

第二个测试验证 `ModelState` 有效时，（通过存储库）添加了新的 `BrainstormSession`，且该方法返回有所需属性的 `RedirectToActionResult`。通常会忽略未调用的模拟调用，但在设置调用末尾调用 `Verifiable` 就可以在测试中对其进行验证。这通过对 `mockRepo.Verify()` 的调用完成，进行这种调用时，如果未调用所需方法，则测试将失败。

注意

通过此示例中使用的 Moq 库，可轻松混合可验证（或称“严格”）mock 和非可验证 mock（也称为“宽松”mock 或存根）。详细了解[使用 Moq 自定义 Mock 行为](#)。

应用中的另一个控制器显示与特定集体讨论会话相关的信息。它包括用于处理无效 ID 值的逻辑：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class SessionController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public SessionController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index(int? id)
        {
            if (!id.HasValue)
            {
                return RedirectToAction(actionName: nameof(Index), controllerName: "Home");
            }

            var session = await _sessionRepository.GetByIdAsync(id.Value);
            if (session == null)
            {
                return Content("Session not found.");
            }

            var viewModel = new StormSessionViewModel()
            {
                DateCreated = session.DateCreated,
                Name = session.Name,
                Id = session.Id
            };

            return View(viewModel);
        }
    }
}

```

控制器操作有三个要测试的事例，每个 `return` 语句对应一个：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class SessionControllerTests
    {
        [Fact]
        public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
        {
            // Arrange
            var controller = new SessionController(sessionRepository: null);

            // Act

```

```

        var result = await controller.Index(id: null);

        // Assert
        var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
        Assert.Equal("Home", redirectToActionResult.ControllerName);
        Assert.Equal("Index", redirectToActionResult.ActionName);
    }

    [Fact]
    public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult((BrainstormSession)null));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var contentResult = Assert.IsType<ContentResult>(result);
        Assert.Equal("Session not found.", contentResult.Content);
    }

    [Fact]
    public async Task IndexReturnsViewResultWithStormSessionViewModel()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult(GetTestSessions().FirstOrDefault(s => s.Id == testSessionId)));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsType<StormSessionViewModel>(viewResult.ViewData.Model);
        Assert.Equal("Test One", model.Name);
        Assert.Equal(2, model.DateCreated.Day);
        Assert.Equal(testSessionId, model.Id);
    }

    private List<BrainstormSession> GetTestSessions()
    {
        var sessions = new List<BrainstormSession>();
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        });
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 1),
            Id = 2,
            Name = "Test Two"
        });
        return sessions;
    }
}
}

```

应用将功能公开为 Web API(与集体讨论会话相关的想法的列表, 以及将新想法添加到会话的方法):

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;

namespace TestingControllersSample.Api
{
    [Route("api/ideas")]
    public class IdeasController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public IdeasController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        [HttpGet("forsession/{sessionId}")]
        public async Task<IActionResult> ForSession(int sessionId)
        {
            var session = await _sessionRepository.GetByIdAsync(sessionId);
            if (session == null)
            {
                return NotFound(sessionId);
            }

            var result = session.Ideas.Select(idea => new IdeaDTO()
            {
                Id = idea.Id,
                Name = idea.Name,
                Description = idea.Description,
                DateCreated = idea.DateCreated
            }).ToList();

            return Ok(result);
        }

        [HttpPost("create")]
        public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var session = await _sessionRepository.GetByIdAsync(model.SessionId);
            if (session == null)
            {
                return NotFound(model.SessionId);
            }

            var idea = new Idea()
            {
                DateCreated = DateTimeOffset.Now,
                Description = model.Description,
                Name = model.Name
            };
            session.AddIdea(idea);

            await _sessionRepository.UpdateAsync(session);

            return Ok(session);
        }
    }
}
```

```
}
```

`ForSession` 方法返回 `IdeaDTO` 类型的列表。避免直接通过 API 调用返回业务域实体，因为它们常包含 API 客户端所需以外的数据，且它们会不必要地将应用的内部域模型与外部公开的 API 配对。域实体和通过网络返回的类型之间的映射可以手动（使用 LINQ `Select`，如此处所示）或使用 [AutoMapper](#) 等库完成。

`Create` 和 `ForSession` API 方法的单位测试：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Api;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class ApiIdeasControllerTests
    {
        [Fact]
        public async Task Create_ReturnsBadRequest_GivenInvalidModel()
        {
            // Arrange & Act
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            var controller = new IdeasController(mockRepo.Object);
            controller.ModelState.AddModelError("error", "some error");

            // Act
            var result = await controller.Create(model: null);

            // Assert
            Assert.IsType<BadRequestObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
        {
            // Arrange
            int testSessionId = 123;
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
                .Returns(Task.FromResult((BrainstormSession)null));
            var controller = new IdeasController(mockRepo.Object);

            // Act
            var result = await controller.Create(new NewIdeaModel());

            // Assert
            Assert.IsType<NotFoundObjectResult>(result);
        }

        [Fact]
        public async Task Create_ReturnsNewlyCreatedIdeaForSession()
        {
            // Arrange
            int testSessionId = 123;
            string testName = "test name";
            string testDescription = "test description";
            var testSession = GetTestSession();
            var mockRepo = new Mock<IBrainstormSessionRepository>();
        }
    }
}
```

```

        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult(testSession));
        var controller = new IdeasController(mockRepo.Object);

        var newIdea = new NewIdeaModel()
        {
            Description = testDescription,
            Name = testName,
            SessionId = testSessionId
        };
        mockRepo.Setup(repo => repo.UpdateAsync(testSession))
            .Returns(Task.CompletedTask)
            .Verifiable();

        // Act
        var result = await controller.Create(newIdea);

        // Assert
        var okResult = Assert.IsType<OkObjectResult>(result);
        var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
        mockRepo.Verify();
        Assert.Equal(2, returnSession.Ideas.Count());
        Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
        Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
    }

    private BrainstormSession GetTestSession()
    {
        var session = new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        };

        var idea = new Idea() { Name = "One" };
        session.AddIdea(idea);
        return session;
    }
}
}

```

如前所述，若要测试方法在 `ModelState` 无效时的行为，请在测试中将模型错误添加到控制器。请勿在单位测试中尝试测试模型有效性或模型绑定 - 仅测试操作方法在遇到特定 `ModelState` 值时的行为。

第二个测试依赖存储库返回 null，所以 mock 存储库配置为返回 null。无需创建测试数据库（在内存中或其他位置）并构建将返回此结果的查询 - 它可以在单个语句中完成，如下所示。

最后一个测试验证调用了存储库的 `Update` 方法。正如我们之前所做，使用 `Verifiable` 调用 mock，然后调用模拟存储库的 `Verify` 方法，以确认执行了可验证方法。确保 `Update` 方法保存数据不是单位测试的职责，这可以通过集成测试完成。

集成测试

完成[集成测试](#)，可确保应用中各独立模块正确协作。通常情况下，可以使用单位测试测试的内容也可以使用集成测试测试，但反之不成立。但是，集成测试比单位测试慢。因此，最好使用单位测试测试其可测内容，并在设计多个协作者时使用集成测试。

虽然它们也可能起作用，但是 mock 对象很少在集成测试中使用。在单位测试中，mock 对象是出于测试目的控制测试单位外协作者行为的有效方法。在集成测试中，使用真实协作者确定整个子系统以正确的方式协同工作。

应用程序状态

执行集成测试时一个重要考虑因素是如何设置应用状态。测试需要互相独立地运行，因此每个测试都应从处于已知

状态的应用开始。如果应用不使用数据库或有任何持久性，这可能不是问题。但是，大多实际应用将其状态保存到某些类型的数据存储，所以一个测试作出的任意修改都可能影响到另一个测试，除非重置数据存储。使用内置 `TestServer`，在集成测试中承载 ASP.NET Core 应用非常简单，但是这并不一定会授予对其要使用的数据的访问权限。如果正在使用实际数据库，则一种方法是让应用连接到测试可以访问的测试数据库，并确保其在每个测试执行前重置到已知状态。

在此示例应用程序中，我是用的是 Entity Framework Core 的 `InMemoryDatabase` 支持，所以我不能从测试项目直接连接到它。相反，我公开应用 `Startup` 类的 `InitializeDatabase` 方法，我在应用在 `Development` 环境中启动时调用该类。只要集成测试将环境设置为 `Development`，就能自动从中获益。我无需担心重置数据库，因为每次应用重启时都会重置 `InMemoryDatabase`。

`Startup` 类：

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.Infrastructure;

namespace TestingControllersSample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(
                optionsBuilder => optionsBuilder.UseInMemoryDatabase("InMemoryDb"));

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory)
        {
            if (env.IsDevelopment())
            {
                var repository = app.ApplicationServices.GetService<IBrainstormSessionRepository>();
                InitializeDatabaseAsync(repository).Wait();
            }

            app.UseStaticFiles();

            app.UseMvcWithDefaultRoute();
        }

        public async Task InitializeDatabaseAsync(IBrainstormSessionRepository repo)
        {
            var sessionList = await repo.ListAsync();
            if (!sessionList.Any())
            {
                await repo.AddAsync(GetTestSession());
            }
        }

        public static BrainstormSession GetTestSession()
        {
```

```

    {
        var session = new BrainstormSession()
        {
            Name = "Test Session 1",
            DateCreated = new DateTime(2016, 8, 1)
        };
        var idea = new Idea()
        {
            DateCreated = new DateTime(2016, 8, 1),
            Description = "Totally awesome idea",
            Name = "Awesome idea"
        };
        session.AddIdea(idea);
        return session;
    }
}
}

```

在以下集成测试中，可以看到频繁使用 `GetTestSession` 方法。

访问视图

每个集成测试类都会配置将运行 ASP.NET Core 应用的 `TestServer`。默认情况下，`TestServer` 会将 Web 应用存储在运行该应用的文件夹（本例中即测试项目文件夹）。因此，当尝试测试返回 `ViewResult` 的控制器操作时，可能看到此错误：

```
The view 'Index' wasn't found. The following locations were searched:  
(list of locations)
```

若要更正此问题，需要配置服务器的内容根，使其可以定位到被测试项目的视图。这可以通过调用 `TestFixture` 类中的 `UseContentRoot` 完成，如下所示：

```

using System;
using System.IO;
using System.Net.Http;
using System.Reflection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.DependencyInjection;

namespace TestingControllersSample.Tests.IntegrationTests
{
    /// <summary>
    /// A test fixture which hosts the target project (project we wish to test) in an in-memory server.
    /// </summary>
    /// <typeparam name="TStartup">Target project's startup type</typeparam>
    public class TestFixture<TStartup> : IDisposable
    {
        private readonly TestServer _server;

        public TestFixture()
            : this(Path.Combine("src"))
        {
        }

        protected TestFixture(string relativeTargetProjectParentDir)
        {
            var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
            var contentRoot = GetProjectPath(relativeTargetProjectParentDir, startupAssembly);

            var builder = new WebHostBuilder()
                .UseContentRoot(contentRoot)
        }
    }
}

```

```

        .ConfigureServices(InitializeServices)
        .UseEnvironment("Development")
        .UseStartup(typeof(TStartup));

    _server = new TestServer(builder);

    Client = _server.CreateClient();
    Client.BaseAddress = new Uri("http://localhost");
}

public HttpClient Client { get; }

public void Dispose()
{
    Client.Dispose();
    _server.Dispose();
}

protected virtual void InitializeServices(IServiceCollection services)
{
    var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;

    // Inject a custom application part manager.
    // Overrides AddMvcCore() because it uses TryAdd().
    var manager = new ApplicationPartManager();
    manager.ApplicationParts.Add(new AssemblyPart(startupAssembly));
    manager.FeatureProviders.Add(new ControllerFeatureProvider());
    manager.FeatureProviders.Add(new ViewComponentFeatureProvider());

    services.AddSingleton(manager);
}

/// <summary>
/// Gets the full path to the target project that we wish to test
/// </summary>
/// <param name="projectRelativePath">
/// The parent directory of the target project.
/// e.g. src, samples, test, or test/Websites
/// </param>
/// <param name="startupAssembly">The target project's assembly.</param>
/// <returns>The full path to the target project.</returns>
private static string GetProjectPath(string projectRelativePath, Assembly startupAssembly)
{
    // Get name of the target project which we want to test
    var projectName = startupAssembly.GetName().Name;

    // Get currently executing test project path
    var applicationBasePath = System.AppContext.BaseDirectory;

    // Find the path to the target project
    var directoryInfo = new DirectoryInfo(applicationBasePath);
    do
    {
        directoryInfo = directoryInfo.Parent;

        var projectDirectoryInfo = new DirectoryInfo(Path.Combine(directoryInfo.FullName,
        projectRelativePath));
        if (projectDirectoryInfo.Exists)
        {
            var fileInfo = new FileInfo(Path.Combine(projectDirectoryInfo.FullName, projectName,
            $"{projectName}.csproj"));
            if (fileInfo.Exists)
            {
                return Path.Combine(projectDirectoryInfo.FullName, projectName);
            }
        }
    }
    while (directoryInfo.Parent != null);
}

```

```
        throw new Exception($"Project root could not be located using the application root  
{applicationBasePath}.");
    }
}
}
```

`TestFixture` 类负责配置和创建 `TestServer`，设置 `HttpClient` 与 `TestServer` 通信。每个集成测试使用 `Client` 属性连接到测试服务器并发出请求。

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class HomeControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        private readonly HttpClient _client;

        public HomeControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ReturnsInitialListOfBrainstormSessions()
        {
            // Arrange - get a session known to exist
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/");

            // Assert
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(testSession.Name, responseString);
        }

        [Fact]
        public async Task PostAddsNewBrainstormSession()
        {
            // Arrange
            string testSessionName = Guid.NewGuid().ToString();
            var data = new Dictionary<string, string>();
            data.Add("SessionName", testSessionName);
            var content = new FormUrlEncodedContent(data);

            // Act
            var response = await _client.PostAsync("/", content);

            // Assert
            Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
            Assert.Equal("/", response.Headers.Location.ToString());
        }
    }
}
```

在上面的第一个测试中，`responseString` 保持“视图”的实际呈现 HTML，可以检查它以确定其包含所需结果。

第二个测试构造具有唯一会话名称的窗体 POST，并将其发布到应用，然后验证返回了预期重定向。

API 方法

如果应用公开 Web API，最好用自动测试确定它们按照预期执行。内置 `TestServer` 使得测试 Web API 更简单。如果 API 方法使用模型绑定，则应始终检查 `ModelState.IsValid`，并用集成测试确定模型验证正确工作。

以下测试集以 `IdeasController` 类中的 `Create` 方法为目标，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class ApiIdeasControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        internal class NewIdeaDto
        {
            public NewIdeaDto(string name, string description, int sessionId)
            {
                Name = name;
                Description = description;
                SessionId = sessionId;
            }

            public string Name { get; set; }
            public string Description { get; set; }
            public int SessionId { get; set; }
        }

        private readonly HttpClient _client;

        public ApiIdeasControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingNameValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("", "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingDescriptionValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }
    }
}
```

```

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooSmall()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 0);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooLarge()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 1000001);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsNotFoundForInvalidSession()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 123);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsCreatedIdeaWithCorrectInputs()
{
    // Arrange
    var testIdeaName = Guid.NewGuid().ToString();
    var newIdea = new NewIdeaDto(testIdeaName, "Description", 1);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    response.EnsureSuccessStatusCode();
    var returnedSession = await response.Content.ReadAsJsonAsync<BrainstormSession>();
    Assert.Equal(2, returnedSession.Ideas.Count);
    Assert.Contains(testIdeaName, returnedSession.Ideas.Select(i => i.Name).ToList());
}

[Fact]
public async Task ForSessionReturnsNotFoundForBadSessionId()
{
    // Arrange & Act
    var response = await _client.GetAsync("/api/ideas/forsession/500");

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task ForSessionReturnsTdeasForValidSessionId()

```

```
public async Task TestForSessionReturnsListOrValidSessionId()
{
    // Arrange
    var testSession = Startup.GetTestSession();

    // Act
    var response = await _client.GetAsync("/api/ideas/forsession/1");

    // Assert
    response.EnsureSuccessStatusCode();
    var ideaList = JsonConvert.DeserializeObject<List<IdeaDTO>>(
        await response.Content.ReadAsStringAsync());
    var firstIdea = ideaList.First();
    Assert.Equal(testSession.Ideas.First().Name, firstIdea.Name);
}
}
```

不同于返回 HTML 视图的操作集成测试，返回结果的 Web API 方法常常可以反序列化为强类型对象，如上面的最后一个测试所示。在此情况下，测试将结果反序列化到 `BrainstormSession` 实例，并确定想法正确添加到其想法集。

可在本文的[示例项目](#)中找到集成测试的其他示例。

适用于 ASP.NET Core MVC 的高级主题

2018/4/10 • 1 min to read • [Edit Online](#)

- [使用应用程序模型](#)
- [筛选器](#)
- [区域](#)
- [应用程序部件](#)
- [自定义模型绑定](#)

使用 ASP.NET Core 中的应用程序模型

2018/5/14 • 11 min to read • [Edit Online](#)

作者: Steve Smith

ASP.NET Core MVC 会定义一个 [应用程序模型](#), 用于表示 MVC 应用的各个组件。通过读取和处理此模型可修改 MVC 元素的行为方式。默认情况下, MVC 遵循特定的约定, 以确定将哪些类视作控制器, 这些类上的哪些方法是操作, 以及参数和路由的行为方式。你可以自定义此行为以满足应用的需要, 方法如下: 创建自己的约定, 并将它们应用于全局或作为属性应用。

模型和提供程序

ASP.NET Core MVC 应用程序模型包括用于描述 MVC 应用程序的抽象接口和具体实现类。此模型是 MVC 根据默认约定发现应用的控制器、操作、操作参数、路由和筛选器的结果。通过使用应用程序模型, 可以修改应用以遵循与默认 MVC 行为不同的约定。参数、名称、路由和筛选器都用作操作和控制器的配置数据。

ASP.NET Core MVC 应用程序模型具有以下结构:

- ApplicationModel
 - 控制器 (ControllerModel)
 - 操作 (ActionModel)
 - 参数 (ParameterModel)

该模型的每个级别都有权访问公用 `Properties` 集合, 层次结构中的较低级别可以访问和覆盖由较高级别设置的属性值。创建操作时, 属性保存到 `ActionDescriptor.Properties` 中。之后, 当处理请求时, 可通过

`ActionContext.ActionDescriptor.Properties` 访问某个约定添加或修改的任何属性。若要基于每项操作对筛选器、模型绑定器等进行配置, 使用属性不失为一个好办法。

注意

一旦完成应用启动, `ActionDescriptor.Properties` 集合就不再是线程安全的(针对写入)。约定是将数据安全添加到此集合的最佳方式。

IApplicationModelProvider

ASP.NET Core MVC 使用提供程序模式(由 [IApplicationModelProvider](#) 接口定义)加载应用程序模型。此部分介绍此提供程序的工作原理的一些内部实现细节。这是一项高级主题 — 利用应用程序模型的大多数应用应使用约定来执行此操作。

`IApplicationModelProvider` 接口的实现相互“包装”, 每个实现都基于其 `Order` 属性以升序调用 `OnProvidersExecuting`。然后, 按相反的顺序调用 `OnProvidersExecuted` 方法。该框架定义了多个提供程序:

首先 (`Order=-1000`):

- `DefaultApplicationModelProvider`

然后 (`Order=-990`):

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

注意

未定义具有相同 `Order` 值的两个提供程序的调用顺序，因此不应依赖此顺序。

注意

`IApplicationModelProvider` 是一种高级概念，框架创建者可对其进行扩展。一般情况下，应用应使用约定，而框架应使用提供程序。主要不同之处在于提供程序始终先于约定运行。

`DefaultApplicationModelProvider` 建立了由 ASP.NET Core MVC 使用的许多默认行为。其职责包括：

- 将全局筛选器添加到上下文
- 将控制器添加到上下文
- 将公共控制器方法作为操作添加
- 将操作方法参数添加到上下文
- 应用路由和其他属性

某些内置行为由 `DefaultApplicationModelProvider` 实现。此提供程序负责构造 `ControllerModel`，后者又引用 `ActionModel`、`PropertyModel` 和 `ParameterModel` 实例。`DefaultApplicationModelProvider` 类是一个内部框架实现细节，未来将对其进行更改。

`AuthorizationApplicationModelProvider` 负责应用与 `AuthorizeFilter` 和 `AllowAnonymousFilter` 属性关联的行为。[详细了解这些属性](#)。

`CorsApplicationModelProvider` 可实现与 `IEnableCorsAttribute`、`IDisableCorsAttribute` 和 `DisableCorsAuthorizationFilter` 关联的行为。[详细了解 CORS](#)。

约定

应用程序模型定义了约定抽象，通过约定抽象来自定义模型行为比重写整个模型或提供程序更简单。建议使用这些抽象来修改应用的行为。通过使用约定，可以编写能动态应用自定义项的代码。使用筛选器可以修改框架的行为，而利用自定义项可以控制整个应用连接在一起的方式。

可用约定如下：

- `IApplicationModelConvention`
- `IControllerModelConvention`
- `IActionModelConvention`
- `IParameterModelConvention`

可通过以下方式应用约定：将它们添加到 MVC 选项，或实现 `Attribute` 并将它们应用于控制器、操作或操作参数（类似于 `Filters`）。与筛选器不同的是，约定仅在应用启动时执行，而不作为每个请求的一部分执行。

示例：修改 ApplicationModel

以下约定用于向应用程序模型添加属性。

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

当在 `Startup` 的 `ConfigureServices` 中添加 MVC 时，应用程序模型约定作为选项应用。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

可从控制器操作内的 `ActionDescriptor` 属性集合中访问属性：

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

示例：修改 `ControllerModel` 说明

与上一个示例一样，也可以修改控制器模型，以包含自定义属性。这些属性将覆盖应用程序模型中指定的具有相同名称的现有属性。以下约定属性可在控制器级别添加说明：

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute, IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}

```

此约定在控制器上作为属性应用。

```

[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}

```

访问“description”属性的方式与前面示例中一样。

示例：修改 ActionModel 说明

可向各项操作应用不同的属性约定，并覆盖已在应用程序或控制器级别应用的行为。

```

using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute, IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}

```

通过将此约定应用于上一示例的控制器中的某项操作，演示了它如何覆盖控制器级别的约定：

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

示例：修改 ParameterModel

可将以下约定应用于操作参数，以修改其 `BindingInfo`。以下约定要求参数为路由参数；忽略其他可能的绑定源（比如查询字符串值）。

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute, IParameterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}
```

该属性可应用于任何操作参数：

```
public class ParameterModelController : Controller
{
    // Will bind: /ParameterModel/.GetById/123
    // WON'T bind: /ParameterModel/.GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)
    {
        return $"Bound to id: {id}";
    }
}
```

示例：修改 ActionModel 名称

以下约定可修改 `ActionModel`，以更新其应用到的操作的名称。新名称以参数形式提供给该属性。此新名称供路由使用，因此它将影响用于访问此操作方法的路由。

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute, IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}
```

此属性应用于 `HomeController` 中的操作方法：

```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

即使方法名称为 `SomeName`，该属性也会覆盖 MVC 使用该方法名称的约定，并将操作名称替换为 `MyCoolAction`。因此，用于访问此操作的路由为 `/Home/MyCoolAction`。

注意

此示例本质上与使用内置 `ActionName` 属性相同。

示例：自定义路由约定

可以使用 `IApplicationModelConvention` 来自定义路由的工作方式。例如，以下约定会将控制器的命名空间合并到其路由中，并将命名空间中的 `.` 替换为路由中的 `/`：

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) // affect one controller in this
sample
                {
                    // Replace the . in the namespace with a / to create the attribute route
                    // Ex: MySite.Admin namespace will correspond to MySite/Admin attribute route
                    // Then attach [controller], [action] and optional {id?} token.
                    // [Controller] and [action] is replaced with the controller and action
                    // name to generate the final template
                    controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
                    {
                        Template = controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
                    };
                }
            }

            // You can continue to put attribute route templates for the controller actions depending on the
way you want them to behave
        }
    }
}

```

该约定作为一个选项添加到 Startup 中。

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}

```

提示

可以使用 `services.Configure<MvcOptions>(c => c.Conventions.Add(YOURCONVENTION));` 来访问 `MvcOptions`，以将约定添加到中间件

此示例将此约定应用于未使用属性路由的路由，其中，控制器名称包含“Namespace”。以下控制器演示了此约定：

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

应用程序模型在 WebApiCompatShim 中的使用

ASP.NET Core MVC 使用一组不同于 ASP.NET Web API 2 的约定。使用自定义约定，可以修改 ASP.NET Core MVC 应用的行为，使其与 Web API 应用保持一致。Microsoft 附带了专用于此的 [WebApiCompatShim](#)。

注意

详细了解[从 ASP.NET Web API 迁移](#)。

若要使用 Web API Compatibility Shim，需将该包添加到项目中，然后通过调用 `Startup` 中的 `AddWebApiConventions`，将约定添加到 MVC：

```
services.AddMvc().AddWebApiConventions();
```

该填充程序提供的约定仅适用于应用中已应用特定属性的部分。以下四个属性用于控制哪些控制器应使用该填充程序的约定来修改自己的约定：

- [UseWebApiActionConventionsAttribute](#)
- [UseWebApiOverloadingAttribute](#)
- [UseWebApiParameterConventionsAttribute](#)
- [UseWebApiRoutesAttribute](#)

操作约定

`UseWebApiActionConventionsAttribute` 用于根据名称将 HTTP 方法映射到操作（例如，`Get` 将映射到 `HttpGet`）。它仅适用于不使用属性路由的操作。

重载

`UseWebApiOverloadingAttribute` 用于应用 `WebApiOverloadingApplicationModelConvention` 约定。此约定可向操作选择过程添加 `OverloadActionConstraint`，以将候选操作限制为其请求满足所有非可选参数的操作。

参数约定

`UseWebApiParameterConventionsAttribute` 用于应用 `WebApiParameterConventionsApplicationModelConvention` 操作约定。此约定指定用作操作参数的简单类型默认来自 URI，而复杂类型来自请求正文。

路由

`UseWebApiRoutesAttribute` 控制是否应用 `WebApiApplicationModelConvention` 控制器约定。启用后，此约定用于向路由添加对 [区域](#) 的支持。

除了一组约定外，该兼容性包还包含一个 `System.Web.Http.ApiController` 基类，用于替换 Web API 提供的等效项。

这允许针对 Web API 编写并且继承自 `ApiController` 的控制器在 ASP.NET Core MVC 上运行时，能够按照设计的方式运行。此控制器基类使用上面列出的所有 `UseWebApi*` 属性进行修饰。`ApiController` 公开了与在 Web API 中找到的属性、方法和结果类型兼容的属性、方法和结果类型。

使用 ApiExplorer 记录应用

应用程序模型在每个级别公开了 `ApiExplorer` 属性，该属性可用于遍历应用的结构。这可用于[使用 Swagger 等工具为 Web API 生成帮助页](#)。`ApiExplorer` 属性公开了 `IsVisible` 属性，后者可设置为指定应公开的应用模型部分。可以使用约定配置此设置：

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

使用此方法（和附加约定，如有需要），可以在应用中的任何级别启用或禁用 API 可见性。

ASP.NET Core 中的筛选器

2018/5/14 • 22 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Tom Dykstra](#) 和 [Steve Smith](#)

ASP.NET Core MVC 中的筛选器允许在请求处理管道中的特定阶段之前或之后运行代码。

重要事项

本主题不适用于 Razor 页面。ASP.NET Core 2.1 预览版及更高版本支持适用于 Razor 页面的 [IPageFilter](#) 和 [IAsyncPageFilter](#)。有关详细信息, 请参阅 [Razor 页面的筛选方法](#)。

内置筛选器处理任务, 例如:

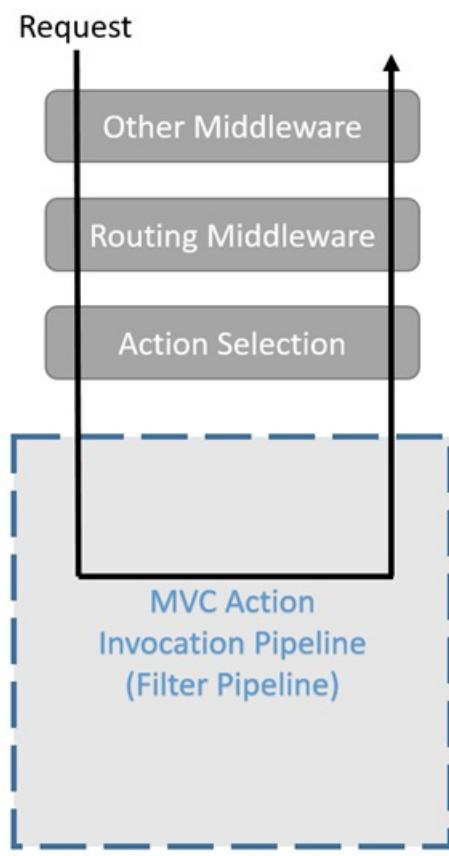
- 授权(防止用户访问未获授权的资源)。
- 确保所有请求都使用 HTTPS。
- 响应缓存(对请求管道进行短路出路, 以便返回缓存的响应)。

可以创建自定义筛选器来处理横切关注点。筛选器可以避免跨操作复制代码。例如, 错误处理异常筛选器可以合并错误处理。

[查看或下载 GitHub 中的示例](#)。

筛选器的工作原理

筛选器在 [MVC 操作调用管道](#)(有时称为 [筛选器管道](#))内运行。筛选器管道在 MVC 选择要执行的操作之后运行。

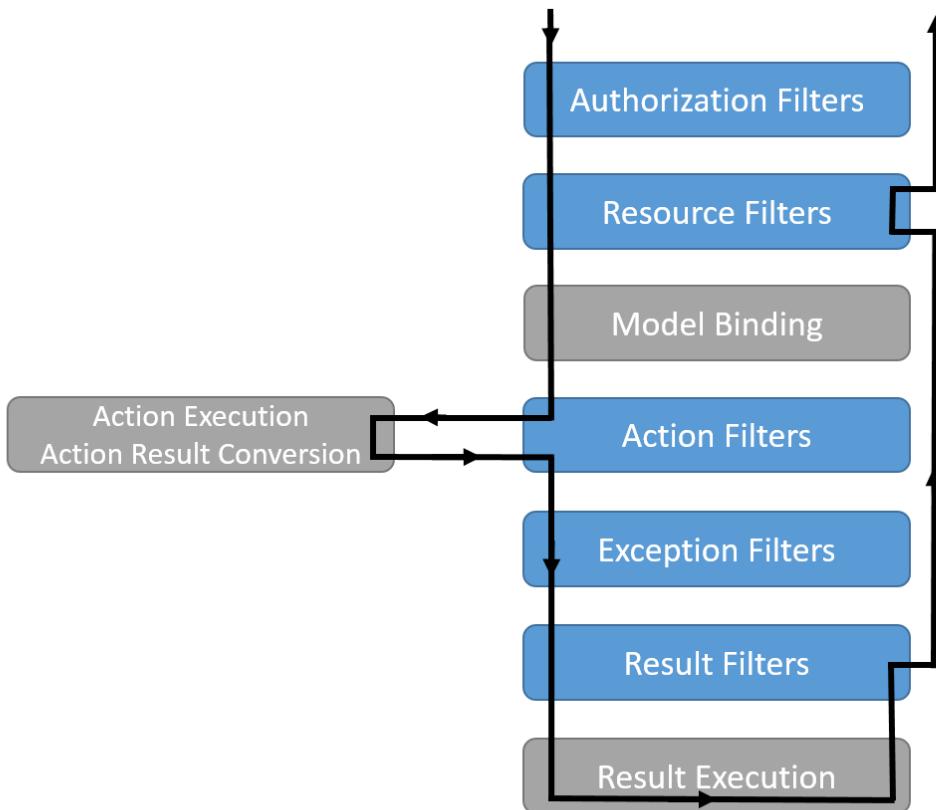


筛选器类型

每种筛选器类型都在筛选器管道中的不同阶段执行。

- 授权筛选器最先运行，用于确定是否已针对当前请求为当前用户授权。如果请求未获授权，它们可以让管道短路。
- 资源筛选器是授权后最先处理请求的筛选器。它们可以在筛选器管道的其余阶段运行之前以及管道的其余阶段完成之后运行代码。出于性能方面的考虑，可以使用它们来实现缓存或以其他方式让筛选器管道短路。它们在模型绑定之前运行，所以可以影响模型绑定。
- 操作筛选器可以在调用单个操作方法之前和之后立即运行代码。它们可用于处理传入某个操作的参数以及从该操作返回的结果。
- 异常筛选器用于在向响应正文写入任何内容之前，对未经处理的异常应用全局策略。
- 结果筛选器可以在执行单个操作结果之前和之后立即运行代码。仅当操作方法成功执行时，它们才会运行。对于必须围绕视图或格式化程序的执行的逻辑，它们很有用。

下图展示了这些筛选器类型在筛选器管道中的交互方式。



实现

筛选器通过不同的接口定义支持同步和异步实现。

可在其管道阶段之前和之后运行代码的同步筛选器定义 `OnStageExecuting` 方法和 `OnStageExecuted` 方法。例如，在调用操作方法之前调用 `OnActionExecuting`，在操作方法返回之后调用 `OnActionExecuted`。

```

using FiltersSample.Helper;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleActionFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            // do something before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // do something after the action executes
        }
    }
}
  
```

异步筛选器定义单一的 `OnStageExecutionAsync` 方法。此方法采用 `FilterTypeExecutionDelegate` 委托来执行筛选器的管道阶段。例如，`ActionExecutionDelegate` 调用该操作方法，用户可以在调用它之前和之后执行代码。

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleAsyncResultFilter : IAsyncResultFilter
    {
        public async Task OnResultExecutingAsync(
            ResultExecutingContext context,
            ResultExecutionDelegate next)
        {
            // do something before the result executes
            var resultContext = await next();
            // do something after the result executes; resultContext.Result will be set
        }
    }
}
```

可以在单个类中为多个筛选器阶段实现接口。例如，[ActionFilterAttribute](#) 类实现 `IActionResultFilter` 和 `IResultFilter`，以及它们的异步等效接口。

注意

筛选器接口的同步和异步版本任意实现一个，而不是同时实现。该框架会先查看筛选器是否实现了异步接口，如果是，则调用该接口。如果不是，则调用同步接口的方法。如果在一个类中同时实现了这两种接口，则仅调用异步方法。使用抽象类时，比如 [ActionFilterAttribute](#)，将为每种筛选器类型仅重写同步方法或仅重写异步方法。

IFilterFactory

`IFilterFactory` 可实现 `IFilter`。因此，`IFilterFactory` 实例可在筛选器管道中的任意位置用作 `IFilter` 实例。当该框架准备调用筛选器时，它会尝试将其转换为 `IFilterFactory`。如果转换成功，则调用 `CreateInstance` 方法来创建将调用的 `IFilter` 实例。这提供了一种很灵活的设计，因为无需在应用启动时显式设置精确的筛选器管道。

用户可以在自己的属性实现上实现 `IFilterFactory` 作为另一种创建筛选器的方法：

```

public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "Header Added" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}

```

内置筛选器属性

该框架包含许多可子类化和自定义的基于属性的内置筛选器。例如，以下结果筛选器会向响应添加标头。

```

using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class AddHeaderAttribute : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute(string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                _name, new string[] { _value });
            base.OnResultExecuting(context);
        }
    }
}

```

属性允许筛选器采用参数，如上面的示例所示。可将此属性添加到控制器或操作方法，并指定 HTTP 标头的名称和值：

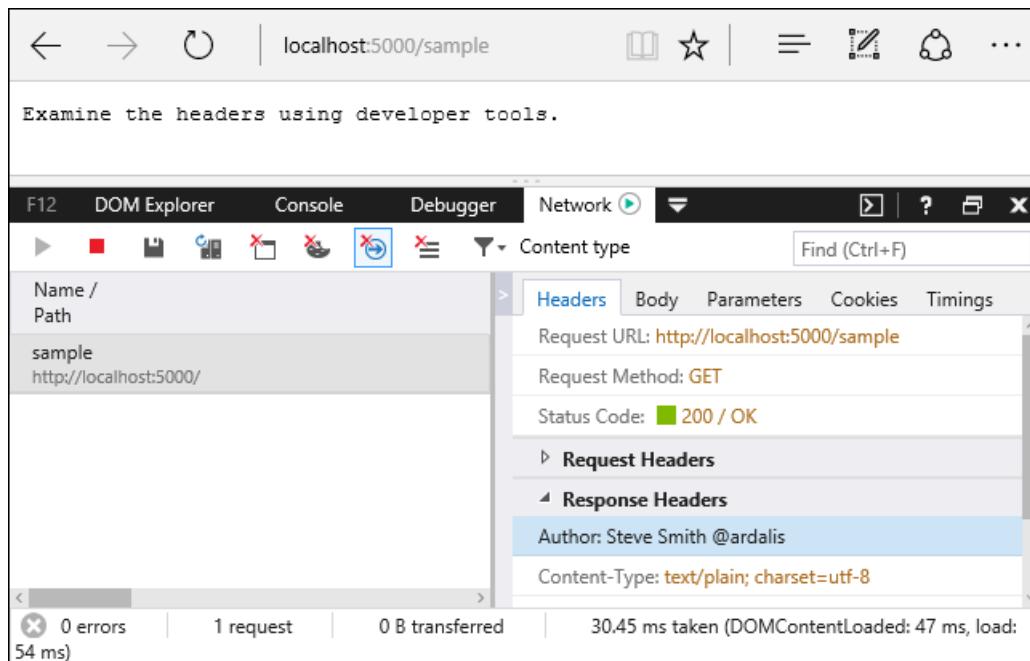
```

[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
}

```

`Index` 操作的结果如下所示：响应标头显示在右下角。



多种筛选器接口具有相应属性，这些属性可用作自定义实现的基类。

筛选器属性：

- `ActionFilterAttribute`
- `ExceptionFilterAttribute`
- `ResultFilterAttribute`
- `FormatFilterAttribute`
- `ServiceFilterAttribute`
- `TypeFilterAttribute`

本文稍后会对 `TypeFilterAttribute` 和 `ServiceFilterAttribute` 进行介绍。

筛选器作用域和执行顺序

可以将筛选器添加到管道中的三个作用域之一。可以使用属性将筛选器添加到特定的操作方法或控制器类。

或者，也可以注册所有控制器和操作的全局筛选器。通过将筛选器添加到 `ConfigureServices` 中的

`MvcOptions.Filters` 集合，可以将其添加为全局筛选器：

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}

```

默认执行顺序

当管道的某个特定阶段有多个筛选器时，作用域可确定筛选器执行的默认顺序。全局筛选器涵盖类筛选器，类筛选器又涵盖方法筛选器。这种模式有时称为“俄罗斯套娃”嵌套，因为增加的每个作用域都包装在前一个作用域中，就像[套娃](#)一样。通常情况下，无需显式确定排序便可获得所需的重写行为。

在这种嵌套模式下，筛选器的 *after* 代码会按照与 *before* 代码相反的顺序运行。其序列如下所示：

- 筛选器的 *before* 代码应用于全局
 - 筛选器的 *before* 代码应用于控制器
 - 筛选器的 *before* 代码应用于操作方法
 - 筛选器的 *after* 代码应用于操作方法
 - 筛选器的 *after* 代码应用于控制器
- 筛选器的 *after* 代码应用于全局

下面的示例阐释了为同步操作筛选器调用筛选器方法的顺序。

序列	筛选器作用域	筛选器方法
1	Global	OnActionExecuting
2	控制器	OnActionExecuting
3	方法	OnActionExecuting
4	方法	OnActionExecuted
5	控制器	OnActionExecuted
6	Global	OnActionExecuted

此序列显示：

- 方法筛选器已嵌套在控制器筛选器中。
- 控制器筛选器已嵌套在全局筛选器中。

换句话说，如果处于异步筛选器的 `OnStageExecutionAsync` 方法内，则当代码位于堆栈上时，所有筛选器都在更严格的作用域中运行。

注意

继承自 `Controller` 基类的每个控制器都包括 `OnActionExecuting` 和 `OnActionExecuted` 方法。这些方法包装针对某项给定操作运行的筛选器：`OnActionExecuting` 在所有筛选器之前调用，`OnActionExecuted` 在所有筛选器之后调用。

重写默认顺序

可以通过实现 `IOrderedFilter` 来重写默认执行序列。此接口公开了一个 `Order` 属性来确定执行顺序，该属性优先于作用域。具有较低 `Order` 值的筛选器会在具有较高 `Order` 值的筛选器之前执行其 *before* 代码。具有较低 `Order` 值的筛选器会在具有较高 `Order` 值的筛选器之后执行其 *after* 代码。可使用构造函数参数来设置 `Order` 属性：

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

如果具有上述示例中所示的 3 个相同的操作筛选器，但将控制器和全局筛选器的 `Order` 属性分别设置为 1 和 2，则会反转执行顺序。

序列	筛选器作用域	ORDER 属性	筛选器方法
1	方法	0	<code>OnActionExecuting</code>
2	控制器	1	<code>OnActionExecuting</code>
3	Global	2	<code>OnActionExecuting</code>
4	Global	2	<code>OnActionExecuted</code>
5	控制器	1	<code>OnActionExecuted</code>
6	方法	0	<code>OnActionExecuted</code>

在确定筛选器的运行顺序时，`Order` 属性优先于作用域。先按顺序对筛选器排序，然后使用作用域消除并列问题。所有内置筛选器实现 `IOrderedFilter` 并将默认 `Order` 值设为 0。对于内置筛选器，作用域会确定顺序，除非将 `Order` 设为非零值。

取消和设置短路

通过设置提供给筛选器方法的 `context` 参数上的 `Result` 属性，可以在筛选器管道的任意位置设置短路。例如，以下资源筛选器将阻止执行管道的其余阶段。

```

using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ShortCircuitingResourceFilterAttribute : Attribute,
        IResourceFilter
    {
        public void OnResourceExecuting(ResourceExecutingContext context)
        {
            context.Result = new ContentResult()
            {
                Content = "Resource unavailable - header should not be set"
            };
        }

        public void OnResourceExecuted(ResourceExecutedContext context)
        {
        }
    }
}

```

在下面的代码中，`ShortCircuitingResourceFilter` 和 `AddHeader` 筛选器都以 `SomeResource` 操作方法为目标。

`ShortCircuitingResourceFilter`：

- 先运行，因为它是资源筛选器且 `AddHeader` 是操作筛选器。
- 对管道的其余部分进行短路处理。

这样 `AddHeader` 筛选器就不会为 `SomeResource` 操作运行。如果这两个筛选器都应用于操作方法级别，只要 `ShortCircuitingResourceFilter` 先运行，此行为就不会变。先运行 `ShortCircuitingResourceFilter`（考虑到它的筛选器类型），或显式使用 `Order` 属性。

```

[AddHeader("Author", "Steve Smith @ardalis")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header should be set.");
    }
}

```

依赖关系注入

可按类型或实例添加筛选器。如果添加实例，该实例将用于每个请求。如果添加类型，则将激活该类型，这意味着将为每个请求创建一个实例，并且[依赖关系注入 \(DI\)](#) 将填充所有构造函数依赖项。按类型添加筛选器等效于 `filters.Add(new TypeFilterAttribute(typeof(MyFilter)))`。

如果将筛选器作为属性实现并直接添加到控制器类或操作方法中，则该筛选器不能由[依赖关系注入 \(DI\)](#) 提供构造函数依赖项。这是因为属性在应用时必须提供自己的构造函数参数。这是属性工作原理上的限制。

如果筛选器具有一些需要从 DI 访问的依赖项，有几种受支持的方法可用。可以使用以下接口之一，将筛选器应用于类或操作方法：

- `ServiceFilterAttribute`

- `TypeFilterAttribute`
- 在属性上实现的 `IFilterFactory`

注意

记录器就是一种可能需要从 DI 获取的依赖项。但是，应避免单纯为进行日志记录而创建和使用筛选器，因为[内置的框架日志记录功能](#)可能已经提供用户所需。如果要将日志记录功能添加到筛选器，它应重点关注业务领域问题或特定于筛选器的行为，而非 MVC 操作或其他框架事件。

ServiceFilterAttribute

`ServiceFilter` 可从 DI 检索筛选器实例。将筛选器添加到该容器的 `ConfigureServices` 中，并在 `ServiceFilter` 属性中引用它

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters")); // an instance
        options.Filters.Add(typeof(SampleActionFilter)); // by type
        options.Filters.Add(new SampleGlobalActionFilter()); // an instance
    });

    services.AddScoped<AddHeaderFilterWithDi>();
}
```

```
[ServiceFilter(typeof(AddHeaderFilterWithDi))]
public IActionResult Index()
{
    return View();
}
```

使用 `ServiceFilter` 时不注册筛选器类型会引发异常：

```
System.InvalidOperationException: No service for type
'FiltersSample.Filters.AddHeaderFilterWithDI' has been registered.
```

`ServiceFilterAttribute` 可实现 `IFilterFactory`。`IFilterFactory` 公开用于创建 `IFilter` 实例的 `CreateInstance` 方法。`CreateInstance` 方法从服务容器 (DI) 中加载指定的类型。

TypeFilterAttribute

`TypeFilterAttribute` 与 `ServiceFilterAttribute` 类似，但不会直接从 DI 容器解析其类型。它使用 `Microsoft.Extensions.DependencyInjection.ObjectFactory` 对类型进行实例化。

由于存在这种差异，所以存在以下情况：

- 使用 `TypeFilterAttribute` 引用的类型不需要先注册在容器中。它们具备由容器实现的依赖项。
- `TypeFilterAttribute` 可以选择为类型接受构造函数参数。

下面的示例演示如何使用 `TypeFilterAttribute` 将参数传递到类型：

```
[TypeFilter(typeof(AddHeaderAttribute),
    Arguments = new object[] { "Author", "Steve Smith (@ardalis)" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

如果你的筛选器符合以下描述：

- 不需要任何参数。
- 具备需要由 DI 填充的构造函数依赖项。

在类和方法上可以不使用 `[TypeFilter(typeof(FilterType))]` 改用自己命名的属性。下面的筛选器展示了如何实现此操作：

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))
    {

    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl	ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("Business action starting...");
            // perform some business logic work

        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // perform some business logic work
            _logger.LogInformation("Business action completed.");
        }
    }
}
```

可以使用 `[SampleActionFilter]` 语法将此筛选器应用于类或方法，而不必使用 `[TypeFilter]` 或 `[ServiceFilter]`。

授权筛选器

*授权筛选器：

- 控制对操作方法的访问。
- 是筛选器管道中要执行的第一个筛选器。
- 具有在它之前的执行的方法，但没有之后执行的方法。

用户只有在编写自己的授权框架时，才应编写自定义授权筛选器。建议配置授权策略或编写自定义授权策略，而不是编写自定义筛选器。内置筛选器实现只负责调用授权系统。

切勿在授权筛选器内引发异常，因为没有任何能处理该异常的组件（异常筛选器不会进行处理）。在出现异常时请小心应对。

详细了解[授权](#)。

资源筛选器

- 实现 `IResourceFilter` 或 `IAsyncResourceFilter` 接口,
- 它们的执行会覆盖筛选器管道的绝大部分。
- 只有[授权筛选器](#)在资源筛选器之前运行。

如果需要使某个请求正在执行的大部分工作短路, 资源筛选器会很有用。例如, 如果响应在缓存中, 则缓存筛选器可以绕开管道的其余阶段。

前面所示的[短路资源筛选器](#)便是一种资源筛选器。另一个示例是 `DisableFormValueModelBindingAttribute`:

- 可以防止模型绑定访问表单数据。
- 如果要上传大型文件, 同时想防止表单被读入内存, 那么此筛选器会很有用。

操作筛选器

操作筛选器:

- 实现 `IActionFilter` 或 `IAsyncActionFilter` 接口。
- 它们的执行围绕着操作方法的执行。

下面是一个操作筛选器示例:

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // do something before the action executes
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // do something after the action executes
    }
}
```

`ActionExecutingContext` 提供以下属性:

- `ActionArguments`: 用于处理对操作的输入。
- `Controller`: 用于处理控制器实例。
- `Result`: 设置此属性会使操作方法和后续操作筛选器的执行短路。引发异常也会阻止操作方法和后续筛选器的执行, 但会被视为失败, 而不是一个成功的结果。

`ActionExecutedContext` 提供 `Controller` 和 `Result` 以及下列属性:

- `Canceled`: 如果操作执行已被另一个筛选器设置短路, 则为 `true`。
- `Exception`: 如果操作或后续操作筛选器引发了异常, 则为非 `NULL` 值。将此属性设置为 `NULL` 可有效地“处理”异常, 并且将执行 `Result`, 就像它是从操作方法正常返回的一样。

对于 `IAsyncActionFilter`, 一个向 `ActionExecutionDelegate` 的调用可以达到以下目的:

- 执行所有后续操作筛选器和操作方法。
- 返回 `ActionExecutedContext`。

若要设置短路, 可将 `ActionExecutingContext.Result` 分配到某个结果实例, 并且不调用

`ActionExecutionDelegate`。

该框架提供一个可子类化的抽象 `ActionFilterAttribute`。

操作筛选器可用于验证模型状态，并在状态为无效时返回任何错误：

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class ValidateModelAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext context)
        {
            if (!context.ModelState.IsValid)
            {
                context.Result = new BadRequestObjectResult(context.ModelState);
            }
        }
    }
}
```

`OnActionExecuted` 方法在操作方法之后运行，可通过 `ActionExecutedContext.Result` 属性查看和处理操作结果。如果操作执行已被另一个筛选器设置短路，则 `ActionExecutedContext.Canceled` 设置为 true。如果操作或后续操作筛选器引发了异常，则 `ActionExecutedContext.Exception` 设置为非 NULL 值。将 `ActionExecutedContext.Exception` 设置为 null：

- 有效地“处理”异常。
- 执行 `ActionExecutedContext.Result`，从操作方法中将它正常返回。

异常筛选器

异常筛选器可实现 `IExceptionFilter` 或 `IAsyncExceptionFilter` 接口。它们可用于为应用实现常见的错误处理策略。

下面的异常筛选器示例使用自定义开发人员错误视图，显示在开发应用时发生的异常的相关详细信息：

```

public class CustomExceptionFilterAttribute : ExceptionFilterAttribute
{
    private readonly IHostingEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilterAttribute(
        IHostingEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public override void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            // do nothing
            return;
        }
        var result = new ViewResult {ViewName = "CustomError"};
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider, context.ModelState);
        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}

```

异常筛选器：

- 没有之前和之后的事件。
- 实现 `OnException` 或 `OnExceptionAsync`。
- 处理控制器创建、[模型绑定](#)、操作筛选器或操作方法中发生的未经处理的异常。
- 请不要捕获资源筛选器、结果筛选器或 MVC 结果执行中发生的异常。

若要处理异常，请将 `ExceptionContext.ExceptionHandled` 属性设置为 `true`，或编写响应。这将停止传播异常。
异常筛选器无法将异常转变为“成功”。只有操作筛选器才能执行该转变。

注意

在 ASP.NET Core 1.1 中，如果将 `ExceptionHandled` 设置为 `true` 并编写响应，则不会发送响应。在这种情况下，ASP.NET Core 1.0 不发送响应，ASP.NET Core 1.1.2 则恢复为 1.0 的行为。有关详细信息，请参阅 GitHub 存储库中的[问题编号 5594](#)。

异常筛选器：

- 非常适合捕获发生在 MVC 操作中的异常。
- 并不像错误处理中间件那么灵活。

建议使用中间件处理异常。仅在需要根据所选 MVC 操作以不同方式执行错误处理时，才使用异常筛选器。例如，应用可能具有用于 API 终结点和视图/HTML 的操作方法。API 终结点可能返回 JSON 形式的错误信息，而基于视图的操作可能返回 HTML 形式的错误页。

`ExceptionFilterAttribute` 可以子类化。

结果筛选器

- 实现 `IResultFilter` 或 `IAsyncResultFilter` 接口。
- 它们的执行围绕着操作结果的执行。

下面是一个添加 HTTP 标头的结果筛选器示例。

```
public class AddHeaderFilterWithDi : IResultFilter
{
    private ILogger _logger;
    public AddHeaderFilterWithDi	ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderFilterWithDi>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation($"Header added: {headerName}");
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has already begun.
    }
}
```

要执行的结果类型取决于所执行的操作。返回视图的 MVC 操作会将所有 Razor 处理作为要执行的 `ViewResult` 的一部分。API 方法可能会将某些序列化操作作为结果执行的一部分。详细了解[操作结果](#)

当操作或操作筛选器生成操作结果时，仅针对成功的结果执行结果筛选器。当异常筛选器处理异常时，不执行结果筛选器。

`OnResultExecuting` 方法可以将 `ResultExecutingContext.Cancel` 设置为 true，使操作结果和后续结果筛选器的执行短路。设置短路时，通常应写入响应对象，以免生成空响应。如果引发异常，则会导致：

- 阻止操作结果和后续筛选器的执行。
- 结果被视为失败而不是成功。

当 `OnResultExecuted` 方法运行时，响应可能已发送到客户端，而且不能再更改（除非引发了异常）。如果操作结果执行已被另一个筛选器设置短路，则 `ResultExecutedContext.Canceled` 设置为 true。

如果操作结果或后续结果筛选器引发了异常，则 `ResultExecutedContext.Exception` 设置为非 NULL 值。将 `Exception` 设置为 NULL 可有效地“处理”异常，并防止 MVC 在管道的后续阶段重新引发该异常。在处理结果筛选器中的异常时，可能无法向响应写入任何数据。如果操作结果在其执行过程中引发异常，并且标头已刷新到客户端，则没有任何可靠的机制可用于发送失败代码。

对于 `IAsyncResultFilter`，通过调用 `ResultExecutionDelegate` 上的 `await next` 可执行所有后续结果筛选器和操作结果。若要设置短路，可将 `ResultExecutingContext.Cancel` 设置为 true，并且不调用 `ResultExecutionDelegate`。

该框架提供一个可子类化的抽象 `ResultFilterAttribute`。前面所示的 `AddHeaderAttribute` 类是一种结果筛选器属性。

在筛选器管道中使用中间件

资源筛选器的工作方式与[中间件](#)类似，即涵盖管道中的所有后续执行。但筛选器又不同于中间件，它们是 MVC 的一部分，这意味着它们有权访问 MVC 上下文和构造。

在 ASP.NET Core 1.1 中，可以在筛选器管道中使用中间件。如果有一个中间件组件，该组件需要访问 MVC 路由数据，或者只能针对特定控制器或操作运行，则可能需要这样做。

若要将中间件用作筛选器，可创建一个具有 `Configure` 方法的类型，该方法可指定要注入到筛选器管道的中

间件。下面的示例使用本地化中间件为请求建立当前区域性：

```
public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {

            DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
        { new RouteDataRequestCultureProvider() { Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}
```

然后，可以使用 `MiddlewareFilterAttribute` 为所选控制器或操作或者在全局范围内运行中间件：

```
[Route("{culture}/{controller}/{action}")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name}",
        +"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

中间件筛选器与资源筛选器在筛选器管道的相同阶段运行，即，在模型绑定之前以及管道的其余阶段之后。

后续操作

若要尝试使用筛选器，请[下载、测试并修改该示例](#)。

ASP.NET Core 中的区域

2018/5/14 • 5 min to read • [Edit Online](#)

作者: Dhananjay Kumar 和 Rick Anderson

区域是 ASP.NET MVC 功能, 用于将相关功能以单独的名称空间(用于路由)和文件夹结构(用于视图)的形式组织到一个组中。使用区域, 将通过为 `controller` 和 `action` 添加另一个路由参数 `area`, 创建用于路由目的的层次结构。

区域提供了将大型 ASP.NET Core MVC Web 应用分区为较小功能分组的方法。区域实际上是应用程序内的一个 MVC 结构。在 MVC 项目中, 模型、控制器和视图等逻辑组件保存在不同的文件夹中, MVC 使用命名约定来创建这些组件之间的关系。对于大型应用, 将应用分区为独立的高级功能区域可能更有利。例如, 具有多个业务单位(如结账、计费、搜索等)的电子商务应用。每个单位都有自己的逻辑组件视图、控制器和模型。在此方案中, 可使用区域对同一项目中的业务组件进行物理分区。

区域可以说是 ASP.NET Core MVC 项目中的较小功能单位, 它具有自己的一组控制器、视图和模型。

以下情况, 请考虑在 MVC 项目中使用区域:

- 应用程序由应进行逻辑区分的多个高级功能组件组成
- 想对 MVC 项目进行分区, 使每个功能区域可以独立工作

区域特性:

- 一个 ASP.NET Core MVC 应用可以有任意数量的区域
- 每个区域都有自己的控制器、模型和视图
- 可用于将大型 MVC 项目组织为可以独立工作的多个高级组件
- 支持具有相同名称的多个控制器 - 只要它们具有不同的区域

我们来看看演示如何创建和使用区域的示例。假设你有一个商店应用, 它有两组不同的控制器和视图:产品和服务。使用 MVC 区域的典型文件夹结构如下所示:

- 项目名称
 - 区域
 - 产品
 - Controllers
 - HomeController.cs
 - ManageController.cs
 - 视图
 - 主页
 - Index.cshtml
 - 管理
 - Index.cshtml
 - 服务

- Controllers
 - HomeController.cs
- 视图
 - 主页
 - Index.cshtml

当 MVC 尝试呈现 Area 中的视图时，它将默认尝试检查以下位置：

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
/Views/Shared/<Action-Name>.cshtml
```

以上是默认位置，可以通过 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions` 上的 `AreaViewLocationFormats` 进行更改。

例如，在下面的代码中，文件夹名称不是“Area”，而是改为了“Categories”。

```
services.Configure<RazorViewEngineOptions>(options =>
{
    options.AreaViewLocationFormats.Clear();
    options.AreaViewLocationFormats.Add("/Categories/{2}/Views/{1}/{0}.cshtml");
    options.AreaViewLocationFormats.Add("/Categories/{2}/Views/Shared/{0}.cshtml");
    options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
});
```

值得注意的一点是，这里唯独看重的是 Views 文件夹的结构，而 Controllers 和 Models 等其他文件夹则无关紧要。例如，完全可以不设 Controllers 和 Models 文件夹。这是可行的，因为 Controllers 和 Models 的内容只是请求相应视图时才会编译到 .dll 中的代码，而 Views 的内容则不是。

定义文件夹层次结构后，需告知 MVC 每个控制器与一个区域相关联。可使用 `[Area]` 属性修饰控制器名称，来完成此操作。

```
...
namespace MyStore.Areas.Products.Controllers
{
    [Area("Products")]
    public class HomeController : Controller
    {
        // GET: /Products/Home/Index
        public IActionResult Index()
        {
            return View();
        }

        // GET: /Products/Home/Create
        public IActionResult Create()
        {
            return View();
        }
    }
}
```

设置适用于新创建区域的路由定义。[路由到控制器操作文章](#)详细介绍了如何创建路由定义，包括使用常规路由和属性路由。在此示例中，我们将使用常规路由。为此，请打开 Startup.cs 文件，并通过添加以下名为 `areaRoute` 的路由定义修改文件。

```
...
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "areaRoute",
            template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
});
```

浏览到 `http://<yourApp>/products`，将调用 `Products` 区域中 `HomeController` 的 `Index` 操作方法。

链接生成

- 生成从基于区域的控制器内的动作到同一控制器内另一个动作的链接。

假设当前的请求路径类似于 `/Products/Home/Create`

HtmlHelper 语法: `@Html.ActionLink("Go to Product's Home Page", "Index")`

TagHelper 语法: `<a asp-action="Index">Go to Product's Home Page`

请注意，我们不需要提供“area”和“controller”值，因为它们在当前请求的上下文中已经可用。此类值称为 `ambient` 值。

- 生成从基于区域的控制器内的动作到不同控制器上另一个动作的链接

假设当前的请求路径类似于 `/Products/Home/Create`

HtmlHelper 语法: `@Html.ActionLink("Go to Manage Products Home Page", "Index", "Manage")`

TagHelper 语法: `<a asp-controller="Manage" asp-action="Index">Go to Manage Products Home Page`

请注意，此处使用“area”的环境值，但上面显式指定了“controller”值。

- 生成从基于区域的控制器内的动作到不同区域中不同控制器上另一个动作的链接。

假设当前的请求路径类似于 `/Products/Home/Create`

HtmlHelper 语法:

`@Html.ActionLink("Go to Services Home Page", "Index", "Home", new { area = "Services" })`

TagHelper 语法:

`<a asp-area="Services" asp-controller="Home" asp-action="Index">Go to Services Home Page`

请注意，此处不使用环境值。

- 生成从基于区域的控制器内的动作到不在区域中的不同控制器上另一个动作的链接。

HtmlHelper 语法:

`@Html.ActionLink("Go to Manage Products Home Page", "Index", "Home", new { area = "" })`

TagHelper 语法:

`<a asp-area="" asp-controller="Manage" asp-action="Index">Go to Manage Products Home Page`

由于我们要生成到基于非区域的控制器操作的链接，因此在此清空“area”的环境值。

发布区域

当 .csproj 文件中包含 `<Project Sdk="Microsoft.NET.Sdk.Web">` 时，所有 `*.cshtml` 和 `wwwroot/**` 文件将发布到输出。

ASP.NET Core 中的应用程序部件

2018/5/14 • 5 min to read • [Edit Online](#)

[查看或下载示例代码\(如何下载\)](#)

应用程序部件是应用程序资源的一种抽象，可通过它发现控制器、视图组件或标记帮助程序等 MVC 功能。

AssemblyPart 就是一种应用程序部件，用于封装程序集引用以及公开类型和编译引用。功能提供程序使用应用程序部件填充 ASP.NET Core MVC 应用的功能。应用程序部件的主要用例是允许将应用配置为从程序集中发现（或避免加载）MVC 功能。

应用程序部件简介

MVC 应用从[应用程序部件](#)中加载其功能。具体而言，[AssemblyPart](#) 类表示受程序集支持的应用程序部件。可以使用这些类发现和加载 MVC 功能，比如控制器、视图组件、标记帮助程序和 Razor 编译源。[ApplicationPartManager](#) 负责跟踪可用于 MVC 应用的应用程序部件和功能提供程序。配置 MVC 时，可以与 [Startup](#) 中的

[ApplicationPartManager](#) 交互：

```
// create an assembly part from a class's assembly
var assembly = typeof(Startup).GetTypeInfo().Assembly;
services.AddMvc()
    .AddApplicationPart(assembly);

// OR
var assembly = typeof(Startup).GetTypeInfo().Assembly;
var part = new AssemblyPart(assembly);
services.AddMvc()
    .ConfigureApplicationPartManager(apm => apm.ApplicationParts.Add(part));
```

默认情况下，MVC 将搜索依赖项树并查找控制器（甚至在其他程序集中）。若要加载任意程序集（例如，从在编译时未引用的插件），可以使用[应用程序部件](#)。

可以使用[应用程序部件](#)避免查找特定程序集或位置中的控制器。可以通过修改 [ApplicationPartManager](#) 的 [ApplicationParts](#) 集合，控制应用可用的部件（或程序集）。[ApplicationParts](#) 集合中条目的顺序并不重要。重要的是在使用 [ApplicationPartManager](#) 配置容器中的服务之前，对该类进行完全配置。例如，应在调用 [AddControllersAsServices](#) 之前完全配置 [ApplicationPartManager](#)。如果未进行完全配置，就意味着，在该方法调用之后添加的应用程序部件中的控制器不受影响（不注册为服务），这可能导致不正确的应用行为。

如果某个程序集包含你不想使用的控制器，则将该程序集从 [ApplicationPartManager](#) 中删除：

```
services.AddMvc()
    .ConfigureApplicationPartManager(apm =>
{
    var dependentLibrary = apm.ApplicationParts
        .FirstOrDefault(part => part.Name == "DependentLibrary");

    if (dependentLibrary != null)
    {
        apm.ApplicationParts.Remove(dependentLibrary);
    }
})
```

除了项目的程序集及其从属程序集，[ApplicationPartManager](#) 还默认包含 [Microsoft.AspNetCore.Mvc.TagHelpers](#) 和 [Microsoft.AspNetCore.Mvc.Razor](#) 的部件。

应用程序功能提供程序

应用程序功能提供程序用于检查应用程序部件，并为这些部件提供功能。以下 MVC 功能有内置功能提供程序：

- 控制器
- 元数据引用
- 标记帮助程序
- 视图组件

功能提供程序从 `IApplicationFeatureProvider<T>` 继承，其中 `T` 是功能的类型。你可以为上面列出的任意 MVC 功能类型实现自己的功能提供程序。`ApplicationPartManager.FeatureProviders` 集合中功能提供程序的顺序可能很重要，因为靠后的提供程序可以对前面的提供程序所执行的操作作出反应。

示例：泛型控制器功能

默认情况下，ASP.NET Core MVC 会忽略泛型控制器（例如，`SomeController<T>`）。此示例使用的控制器功能提供程序在默认提供程序后面运行并为指定的类型列表（在 `EntityTypes.Types` 中定义）添加泛型控制器实例：

```
public class GenericControllerFeatureProvider : IApplicationFeatureProvider<ControllerFeature>
{
    public void PopulateFeature(IEnumerable<ApplicationPart> parts, ControllerFeature feature)
    {
        // This is designed to run after the default ControllerTypeProvider,
        // so the list of 'real' controllers has already been populated.
        foreach (var entityType in EntityTypes.Types)
        {
            var typeName = entityType.Name + "Controller";
            if (!feature.Controllers.Any(t => t.Name == typeName))
            {
                // There's no 'real' controller for this entity, so add the generic version.
                var controllerType = typeof(GenericController<>)
                    .MakeGenericType(entityType.AsType()).GetTypeInfo();
                feature.Controllers.Add(controllerType);
            }
        }
    }
}
```

实体类型：

```
public static class EntityTypes
{
    public static IReadOnlyList<TypeInfo> Types => new List<TypeInfo>()
    {
        typeof(Sprocket).GetTypeInfo(),
        typeof(Widget).GetTypeInfo(),
    };

    public class Sprocket { }
    public class Widget { }
}
```

将该功能提供程序添加到 `Startup` 中：

```
services.AddMvc()
    .ConfigureApplicationPartManager(apm =>
        apm.FeatureProviders.Add(new GenericControllerFeatureProvider()));
```

默认情况下，用于路由的泛型控制器名称的格式为 `GenericController`1[Widget]`，而不是 `Widget`。以下属性用于修改该名称，以便与控制器使用的泛型类型对应：

```

using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System;

namespace AppPartsSample
{
    // Used to set the controller name for routing purposes. Without this convention the
    // names would be like 'GenericController`1[Widget]' instead of 'Widget'.
    //
    // Conventions can be applied as attributes or added to MvcOptions.Conventions.
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = true)]
    public class GenericControllerNameConvention : Attribute, IControllerModelConvention
    {
        public void Apply(ControllerModel controller)
        {
            if (controller.ControllerType.GetGenericTypeDefinition() !=
                typeof(GenericController<>))
            {
                // Not a GenericController, ignore.
                return;
            }

            var entityType = controller.ControllerType.GenericTypeArguments[0];
            controller.ControllerName = entityType.Name;
        }
    }
}

```

`GenericController` 类:

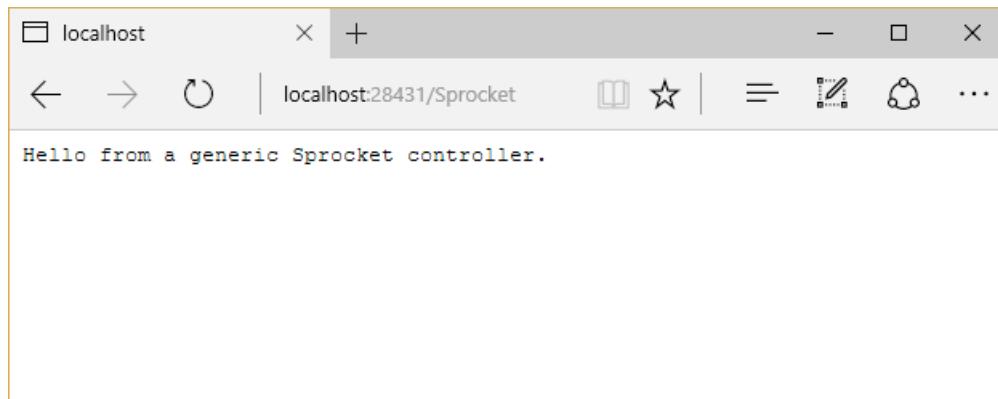
```

using Microsoft.AspNetCore.Mvc;

namespace AppPartsSample
{
    [GenericControllerNameConvention] // Sets the controller name based on typeof(T).Name
    public class GenericController<T> : Controller
    {
        public IActionResult Index()
        {
            return Content($"Hello from a generic {typeof(T).Name} controller.");
        }
    }
}

```

当请求匹配的路由时, 结果如下:



示例: 显示可用功能

可循环访问可用于应用的已填充功能, 方法为通过[依赖关系注入](#)请求 `ApplicationPartManager`, 并用它来填充相应功能的实例:

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }

        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var metaDataReferenceFeature = new MetadataReferenceFeature();
            _partManager.PopulateFeature(metaDataReferenceFeature);
            viewModel.MetadataReferences = metaDataReferenceFeature.MetadataReferences
                .ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents = viewComponentFeature.ViewComponents.ToList();

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

示例输出：

Home Page - AppPartS: X +

localhost:28431/Features

Back Forward Refresh Home Stop More

Features

Controllers:

- FeaturesController
- HomeController
- HelloController
- GenericController`1
- GenericController`1

Metadata References:

- C:\dev\ssmith\github.com\aspnet-docs\aspnetcore\mvc\extensibility\app-parts\sample\src\AppPartSample\bin\Debug\netcoreapp1.0\AppPartSample.dll
- C:\Users\steve_000\.nuget\packages\Microsoft.AspNetCore.Antiforgery\1.0.1\lib\netstandard1.3\Microsoft.AspNetCore.Antiforgery.dll
- C:\Users\steve_000\.nuget\packages\Microsoft.AspNetCore.Authorization\1.0.0

Tag Helpers:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

View Components:

- AnchorTagHelper
- CacheTagHelper
- DistributedCacheTagHelper
- EnvironmentTagHelper
- FormTagHelper

[Home](#)

ASP.NET Core 中的自定义模型绑定

2018/5/14 • 7 min to read • [Edit Online](#)

作者: [Steve Smith](#)

通过模型绑定，控制器操作可直接使用模型类型(作为方法参数传入)而不是 HTTP 请求。由模型绑定器处理传入的请求数据和应用程序模型之间的映射。开发人员可以通过实现自定义模型绑定器来扩展内置的模型绑定功能(尽管通常不需要编写自己的提供程序)。

[查看或下载 GitHub 中的示例](#)

默认模型绑定器限制

默认模型绑定器支持大多数常见的 .NET Core 数据类型，能够满足大部分开发人员的需求。他们希望将基于文本的输入从请求直接绑定到模型类型。绑定输入之前，可能需要对其进行转换。例如，当拥有某个可以用来查找模型数据的键时。基于该键，用户可以使用自定义模型绑定器来获取数据。

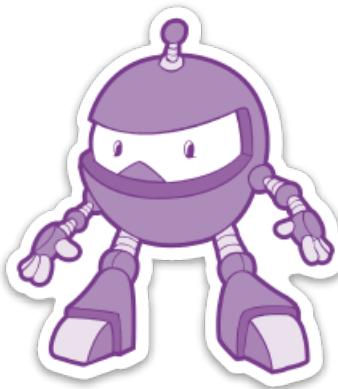
模型绑定查看

模型绑定为其操作对象的类型使用特定定义。简单类型转换自输入中的单个字符串。复杂类型转换自多个输入值。框架基于是否存在 `TypeConverter` 来确定差异。如果简单 `string` -> `SomeType` 映射不需要外部资源，建议创建类型转换器。

创建自己的自定义模型绑定器之前，有必要查看现有模型绑定器的实现方式。考虑使用 [ByteArrayModelBinder](#)，它可将 base64 编码的字符串转换为字节数组。字节数组通常存储为文件或数据库 BLOB 字段。

使用 `ByteArrayModelBinder`

Base64 编码的字符串可用来表示二进制数据。例如，下图可编码为一个字符串。



如下显示了编码字符串的一小部分：

The screenshot shows a Notepad window titled "Base64 Encoded Dotnet Bot.txt - Notepad". The content of the window is a single, extremely long base64 encoded string. The string starts with "iVBORw0KGgoAAAANSUhEUgAAAQAAAEEFCAYAAAdq1vKRAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQU" and continues for several thousand characters. It contains various characters including letters, numbers, and special symbols typical of a base64 encoded file.

按照[示例的自述文件](#)中的说明将 base64 编码的字符串转换为文件。

ASP.NET Core MVC 可以采用 base64 编码的字符串，并使用 `ByteArrayModelBinder` 将其转换为字节数组。实现 `IModelBinderProvider` 的 `ByteArrayModelBinderProvider` 将 `byte[]` 参数映射到 `ByteArrayModelBinder`：

```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        return new ByteArrayModelBinder();
    }

    return null;
}
```

创建自己的自定义模型绑定器时，可实现自己的 `IModelBinderProvider` 类型，或使用 `ModelBinderAttribute`。

以下示例显示如何使用 `ByteArrayModelBinder` 将 base64 编码的字符串转换为 `byte[]`，并将结果保存到文件中：

```
// POST: api/image
[HttpPost]
public void Post(byte[] file, string filename)
{
    string filePath = Path.Combine(_env.ContentRootPath, "wwwroot/images/upload", filename);
    if (System.IO.File.Exists(filePath)) return;
    System.IO.File.WriteAllBytes(filePath, file);
}
```

可以使用 [Postman](#) 等工具将 base64 编码的字符串发布到此 api 方法：

只要绑定器可以将请求数据绑定到相应命名的属性或参数，模型绑定就会成功。以下示例演示如何将 `ByteArrayModelBinder` 与 视图模型结合使用：

```
[HttpPost("Profile")]
public void SaveProfile(ProfileViewModel model)
{
    string filePath = Path.Combine(_env.ContentRootPath, "wwwroot/images/upload", model.FileName);
    if (System.IO.File.Exists(model.FileName)) return;
    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

自定义模型绑定器示例

在本部分中，我们将实现具有以下功能的自定义模型绑定器：

- 将传入的请求数据转换为强类型键参数。
- 使用 Entity Framework Core 来提取关联的实体。
- 将关联的实体作为自变量传递给操作方法。

以下示例在 `Author` 模型上使用 `ModelBinder` 属性：

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

在前面的代码中，`ModelBinder` 属性指定应当用于绑定 `Author` 操作参数的 `IModelBinder` 的类型。

通过 Entity Framework Core 和 `authorId` 提取数据源中的实体，可使用 `AuthorEntityBinder` 来绑定 `Author` 参数：

```
public class AuthorEntityBinder : IModelBinder
{
    private readonly AppDbContext _db;
    public AuthorEntityBinder(AppDbContext db)
    {
        _db = db;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        // Specify a default argument name if none is set by ModelBinderAttribute
        var modelName = bindingContext.BinderModelName;
        if (string.IsNullOrEmpty(modelName))
        {
            modelName = "authorId";
        }

        // Try to fetch the value of the argument by name
        var valueProviderResult =
            bindingContext.ValueProvider.GetValue(modelName);

        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
            valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        int id = 0;
        if (!int.TryParse(value, out id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                bindingContext.ModelName,
                "Author Id must be an integer.");
            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _db.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

以下代码显示如何在操作方法中使用 `AuthorEntityBinder` :

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    return Ok(author);
}
```

可使用 `ModelBinder` 属性将 `AuthorEntityBinder` 应用于不使用默认约定的参数：

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")]Author author)
{
    if (author == null)
    {
        return NotFound();
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    return Ok(author);
}
```

在此示例中，由于参数的名称不是默认的 `authorId`，因此使用 `ModelBinder` 属性在参数上指定该名称。请注意，比起在操作方法中查找实体，控制器和操作方法都得到了简化。使用 Entity Framework Core 获取创建者的逻辑会移动到模型绑定器。如果有几种方法绑定到创建者模型，就能得到很大程度的简化，并且有助于遵循 DRY 原则。

可以将 `ModelBinder` 属性应用到各个模型属性(例如视图模型上)或操作方法参数，以便为该类型或操作指定某一模型绑定器或模型名称。

实现 `ModelBinderProvider`

可以实现 `IModelBinderProvider`，而不是应用属性。这就是内置框架绑定器的实现方式。指定绑定器所操作的类型时，指定它生成的参数的类型，而不是绑定器接受的输入。以下绑定器提供程序适用于 `AuthorEntityBinder`。将其添加到 MVC 提供程序的集合中时，无需在 `Author` 或 `Author` 类型参数上使用 `ModelBinder` 属性。

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

注意: 上述代码返回 `BinderTypeModelBinder`。`BinderTypeModelBinder` 充当模型绑定器中心, 并提供依赖关系注入 (DI)。`AuthorEntityBinder` 需要 DI 来访问 EF Core。如果模型绑定器需要 DI 中的服务, 请使用 `BinderTypeModelBinder`。

若要使用自定义模型绑定器提供程序, 请将其添加到 `ConfigureServices` 中:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

评估模型绑定器时, 按顺序检查提供程序的集合。使用返回绑定器的第一个提供程序。

下图显示调试程序中的默认模型绑定器。

Name	Value
options.ModelBind	Count = 14
[0]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.BinderTypeModelBinderProvider}
[1]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ServicesModelBinderProvider}
[2]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.BodyModelBinderProvider}
[3]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.HeaderModelBinderProvider}
[4]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.SimpleTypeModelBinderProvider}
[5]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CancellationTokenModelBinderProvider}
[6]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ByteArrayModelBinderProvider}
[7]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormFileModelBinderProvider}
[8]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormCollectionModelBinderProvider}
[9]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.KeyValuePairModelBinderProvider}
[10]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DictionaryModelBinderProvider}
[11]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ArrayModelBinderProvider}
[12]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CollectionModelBinderProvider}
[13]	{Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinderProvider}

向集合的末尾添加提供程序, 可能会导致在调用自定义绑定器之前调用内置模型绑定器。在此示例中, 向集合的开头添加自定义提供程序, 确保它用于 `Author` 操作参数。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase());

    services.AddMvc(options =>
    {
        // add custom binder to beginning of collection
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

建议和最佳做法

自定义模型绑定:

- 不应尝试设置状态代码或返回结果(例如 404 Not Found)。如果模型绑定失败, 那么该操作方法本身的操作筛选器或逻辑会处理失败。
- 对于消除操作方法中的重复代码和跨领域问题最为有用。
- 通常不应用其将字符串转换为自定义类型, 而应选择用 `TypeConverter` 来完成此操作。

使用 ASP.NET Core 构建 Web API

2018/5/14 • 3 min to read • [Edit Online](#)

作者: Scott Addie

[查看或下载示例代码\(如何下载\)](#)

本文档说明如何在 ASP.NET Core 中构建 Web API 以及每项功能的最佳适用场景。

从 ControllerBase 派生类

从控制器(旨在用作 Web API)中的 `ControllerBase` 类继承。例如:

```
[!code-csharp]
```

```
[!code-csharp]
```

通过 `ControllerBase` 类可使用大量属性和方法。前面的示例中包含某些此类方法, 如 `BadRequest` 和 `CreatedAtAction`。将在操作方法中调用这些方法以分别返回 HTTP 400 和 201 状态代码。将使用 `ModelState` 属性(还可由 `ControllerBase` 提供)执行请求模型验证。

使用 ApiControllerAttribute 批注类

ASP.NET Core 2.1 引入了 `[ApiController]` 特性, 用于批注 Web API 控制器类。例如:

```
[!code-csharp]
```

此特性通常与 `ControllerBase` 配合使用以获得其他有用的方法和属性。通过 `ControllerBase` 可使用 `NotFound` 和 `File` 等方法。

另一种方法是创建使用 `[ApiController]` 特性进行批注的自定义基本控制器类:

```
[!code-csharp]
```

以下各部分说明该特性添加的便利功能。

自动 HTTP 400 响应

验证错误会自动触发 HTTP 400 响应。操作中不需要以下代码:

```
[!code-csharp]
```

在 `Startup.ConfigureServices` 中使用以下代码将禁用此默认行为:

```
[!code-csharp]
```

绑定源参数推理

绑定源特性定义可找到操作参数值的位置。存在以下绑定源特性:

特性	绑定源
<code>[FromBody]</code>	请求正文
<code>[FromForm]</code>	请求正文中的表单数据
<code>[FromHeader]</code>	请求标头

特性	绑定源
<code>[FromQuery]</code>	请求查询字符串参数
<code>[FromRoute]</code>	当前请求中的路由数据
<code>[FromServices]</code>	作为操作参数插入的请求服务

注意

如果值可能包含 `%2f` (即 `/`)，请不要使用 `[FromRoute]`，因为 `%2f` 不会非转义为 `/`。如果值可能包含 `%2f`，则使用 `[FromQuery]`。

没有 `[ApiController]` 特性时，将显式定义绑定源特性。在下面的示例中，`[FromQuery]` 特性指示 `discontinuedOnly` 参数值在请求 URL 的查询字符串中提供：

```
![code-csharp]
```

推理规则应用于操作参数的默认数据源。这些规则将配置绑定资源，否则你可以手动应用操作参数。绑定源特性的行为如下：

- **[FromBody]**，针对复杂类型参数进行推断。此规则不适用于具有特殊含义的任何复杂的内置类型，如 `IFormCollection` 和 `CancellationToken`。绑定源推理代码将忽略这些特殊类型。当操作中的多个参数为显式指定（通过 `[FromBody]`）或在请求正文（body）中作为绑定进行推断时，将会引发异常。例如，下面的操作签名会导致异常：

```
![code-csharp]
```

- **[FromForm]**，针对 `IFormFile` 和 `IFormFileCollection` 类型的操作参数进行推断。该特性不针对任何简单类型或用户定义类型进行推断。
- **[FromRoute]**，针对与路由模板中的参数相匹配的任何操作参数名称进行推断。当多个路由与一个操作参数匹配时，任何路由值都视为 `[FromRoute]`。
- **[FromQuery]**，针对任何其他操作参数进行推断。

在 `Startup.ConfigureServices` 中使用以下代码将禁用默认推理规则：

```
![code-csharp]
```

Multipart/form-data 请求推理

使用 `[FromForm]` 特性批注操作参数时，将推断 `multipart/form-data` 请求内容类型。

在 `Startup.ConfigureServices` 中使用以下代码将禁用默认行为：

```
![code-csharp]
```

特性路由要求

特性路由是必要条件。例如：

```
![code-csharp]
```

不能通过在 `UseMvc` 中定义的 [传统路由](#) 或通过 `Startup.Configure` 中的 `UseMvcWithDefaultRoute` 访问操作。

其他资源

- [控制器操作返回类型](#)
- [自定义格式化程序](#)

- 格式化响应数据
- 使用 Swagger 的帮助页
- 路由到控制器操作

适用于 ASP.NET Core Web API 的高级主题

2018/4/28 • 1 min to read • [Edit Online](#)

- [自定义格式化程序](#)
- [格式化响应数据](#)

在 ASP.NET Core 中测试和调试

2018/4/28 • 1 min to read • [Edit Online](#)

- [单元测试](#)
- [集成测试](#)
- [Razor 页面单位与集成测试](#)
- [测试控制器](#)
- [调试 ASP.NET Core 2.x 源](#)
- [远程调试](#)
- [快照调试](#)
- [YouTube: 诊断 ASP.NET Core 应用程序中的问题](#)

在 ASP.NET Core 的集成测试

2018/5/14 • 8 min to read • [Edit Online](#)

作者: Steve Smith

集成测试可确保应用程序的组件组合在一起时正常工作。ASP.NET Core 使用单元测试框架和可用于处理请求(无网络费用)的内置测试 web 主机支持集成测试。

[查看或下载示例代码\(如何下载\)](#)

集成测试的简介

集成测试来验证应用程序的不同部分正常会在一起。与不同[单元测试](#), 集成测试经常涉及应用程序基础结构问题, 如数据库、文件系统、网络资源, 或 web 请求和响应。单元测试使用 fakes 或使用模拟对象来替代这些问题, 但集成测试的目的是确认系统按预期的这些系统。

集成测试, 因为在执行较大代码段, 并且它们依赖于基础结构元素倾向于数量级慢于单元测试。因此, 它是一个好办法限制多少集成测试编写, 尤其是可以使用单元测试中测试相同的行为。

注意

如果某些行为可以使用单元测试或集成测试进行测试, 更愿意相应单元测试, 因为它将为几乎始终是更快。你可能有几十或数百个使用许多不同的输入的单元测试, 但只需少量的集成测试介绍最重要的方案。

测试你自己的方法中的逻辑通常是域的单元测试。测试你的应用程序在其框架, 例如, 使用 ASP.NET Core, 或与数据库工作原理是集成测试的其中派上用场。它并不需要太多的集成测试, 以确认你能够将行写入到数据库和读回。你不需要进行测试每个可能的排列的数据访问代码-只需测试足以以为你提供你的应用程序正常的置信度。

集成测试 ASP.NET 核心

若要获取设置为运行的集成测试, 你将需要创建的测试项目、添加到 ASP.NET 核心 web 项目中, 引用和安装的测试运行程序。此过程所述[单元测试](#)文档, 以及运行测试和命名你的测试和测试类的建议的更多详细说明。

注意

分隔单元测试和集成测试使用不同的项目。这有助于确保你不会意外地引入到你的单元测试的基础结构问题, 并可以轻松地选择要运行的测试集。

测试主机

ASP.NET 核心包括可以添加到集成测试项目, 并用于托管 ASP.NET Core 应用程序, 而无需实际 web 宿主请求的服务测试的测试主机。提供的示例包括集成测试项目已配置为使用[xUnit](#)和测试主机。它使用[Microsoft.AspNetCore.TestHost](#) NuGet 包。

一次 `Microsoft.AspNetCore.TestHost` 包括在项目中后, 你将能够创建和配置 `TestServer` 在测试中。下面的代码演示如何验证对站点的根目录的请求返回"Hello World !" 并且应成功运行针对默认值由 Visual Studio 创建的 ASP.NET 核心空 Web 模板。

```

public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();

        var responseString = await response.Content.ReadAsStringAsync();

        // Assert
        Assert.Equal("Hello World!",
            responseString);
    }
}

```

此测试使用排列 Act 断言模式。在创建的实例的构造函数中完成准备步骤 `TestServer`。一个已配置 `WebHostBuilder` 将用于创建 `TestHost`；在此示例中，`Configure` 方法从待测试 (SUT) 系统 `Startup` 类传递给 `WebHostBuilder`。此方法将用于配置的请求管道 `TestServer` 到 SUT 服务器将配置方式相同。

在测试的 Act 部分中，向发出请求 `TestServer` “/” 路径和响应的实例返回读入字符串。此字符串与“Hello World！”的预期字符串进行比较。如果它们匹配，则测试通过；否则，它将失败。

现在你可以添加了几个其他集成测试，以确认检查功能主要通过 web 应用程序能否正常运行：

```

public class PrimeWebCheckPrimeShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebCheckPrimeShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    private async Task<string> GetCheckPrimeResponseString(
        string querystring = "")
    {
        var request = "/checkprime";
        if(!string.IsNullOrEmpty(querystring))
        {
            request += "?" + querystring;
        }
        var response = await _client.GetAsync(request);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }

    [Fact]
    public async Task ReturnInstructionsGivenEmptyQueryString()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString();

        // Assert
        Assert.Equal("Pass in a number to check in the form /checkprime?5",
            responseString);
    }

    [Fact]
    public async Task ReturnPrimeGiven5()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("5");

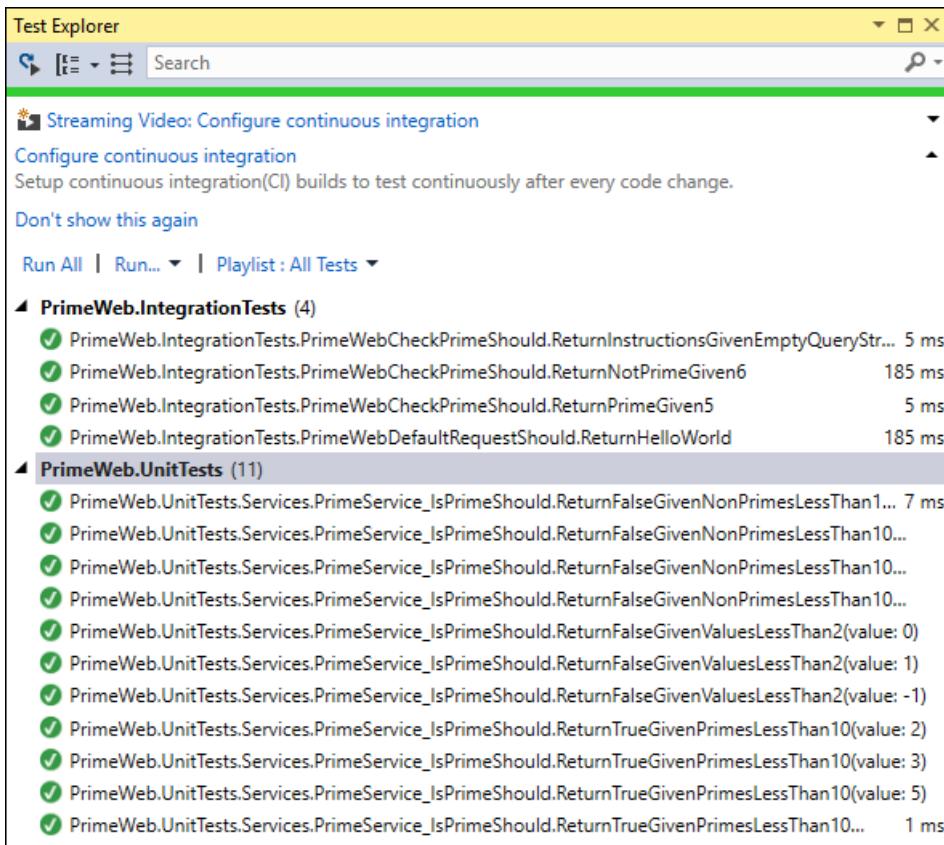
        // Assert
        Assert.Equal("5 is prime!",
            responseString);
    }

    [Fact]
    public async Task ReturnNotPrimeGiven6()
    {
        // Act
        var responseString = await GetCheckPrimeResponseString("6");

        // Assert
        Assert.Equal("6 is NOT prime!",
            responseString);
    }
}

```

请注意，你实际上不尝试使用这些测试进行测试的质数检查程序的正确性但而应假定 web 应用程序执行的操作，你的预期。你已为你提供中的置信度的单元测试覆盖率 `PrimeService`，如你可以在此处看到：



你可以了解有关中的单元测试[单元测试](#)文章。

集成测试 Mvc/Razor

测试项目，其中包含 Razor 视图需要 `<PreserveCompilationContext>` 设置为 true 中 `.csproj` 文件：

```
<PreserveCompilationContext>true</PreserveCompilationContext>
```

缺少此元素的项目将生成类似于以下错误：

```
Microsoft.AspNetCore.Mvc.Razor.Compilation.CompilationFailedException: 'One or more compilation failures occurred:  
oobebhccx.1bd(4,62): error CS0012: The type 'Attribute' is defined in an assembly that is not referenced. You must add a reference to assembly 'netstandard, Version=2.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51'.
```

重构以使用中间件

重构是应用程序的代码更改不会更改其行为改进其设计的过程。它应理想情况下进行一套传递测试，因为这些帮助确保系统的行为保持不变之前，和之后所做的更改时。在其中检查逻辑质数实现中 web 应用程序的方式查看 `Configure` 方法，你会看到：

```

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

    app.Run(async (context) =>
    {
        if (context.Request.Path.Value.Contains("checkprime"))
        {
            int numberToCheck;
            try
            {
                numberToCheck = int.Parse(context.Request.QueryString.Value.Replace("?", ""));
                var primeService = new PrimeService();
                if (primeService.IsPrime(numberToCheck))
                {
                    await context.Response.WriteAsync($"{numberToCheck} is prime!");
                }
                else
                {
                    await context.Response.WriteAsync($"{numberToCheck} is NOT prime!");
                }
            }
            catch
            {
                await context.Response.WriteAsync("Pass in a number to check in the form /checkprime?5");
            }
        }
        else
        {
            await context.Response.WriteAsync("Hello World!");
        }
    });
}

```

此代码有效，但它并不如何你想要在 ASP.NET Core 应用程序，即使简单，因为这是实现此类功能。假设什么 `Configure` 方法将如下所示如果您需要将这么多代码添加到其中，每次您添加另一个 URL 终结点！

要考虑的一个选项添加[MVC](#)与应用程序创建控制器来处理主要检查。但是，假定当前不需要任何其他 MVC 的功能，即一个位浪费。

你可以但是，充分利用 ASP.NET Core[中间件](#)，这将帮助我们封装检查其自己的类中的逻辑校准并更好地实现[关注点分离](#)中 `Configure` 方法。

你想要允许该中间件用于指定作为参数，因此中间件类需要的路径 `RequestDelegate` 和 `PrimeCheckerOptions` 其构造函数中的实例。如果请求的路径不匹配此中间件是配置需要，你只需调用链中的下一步的中间件并执行任何进一步操作。中的实现代码的其余部分 `Configure` 现已在 `Invoke` 方法。

注意

因为取决于该中间件 `PrimeService` 服务，还所请求的构造函数使用此服务实例。框架将提供此服务通过[依赖关系注入](#)，假设它已配置，例如，在 `ConfigureServices`。

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using PrimeWeb.Services;
using System;
using System.Threading.Tasks;

namespace PrimeWeb.Middleware

```

```

    {
        public class PrimeCheckerMiddleware
        {
            private readonly RequestDelegate _next;
            private readonly PrimeCheckerOptions _options;
            private readonly PrimeService _primeService;

            public PrimeCheckerMiddleware(RequestDelegate next,
                PrimeCheckerOptions options,
                PrimeService primeService)
            {
                if (next == null)
                {
                    throw new ArgumentNullException(nameof(next));
                }
                if (options == null)
                {
                    throw new ArgumentNullException(nameof(options));
                }
                if (primeService == null)
                {
                    throw new ArgumentNullException(nameof(primeService));
                }

                _next = next;
                _options = options;
                _primeService = primeService;
            }

            public async Task Invoke(HttpContext context)
            {
                var request = context.Request;
                if (!request.Path.HasValue ||
                    request.Path != _options.Path)
                {
                    await _next.Invoke(context);
                }
                else
                {
                    int numberToCheck;
                    if (int.TryParse(request.QueryString.Value.Replace("?", ""), out numberToCheck))
                    {
                        if (_primeService.IsPrime(numberToCheck))
                        {
                            await context.Response.WriteAsync($"{{numberToCheck}} is prime!");
                        }
                        else
                        {
                            await context.Response.WriteAsync($"{{numberToCheck}} is NOT prime!");
                        }
                    }
                    else
                    {
                        await context.Response.WriteAsync($"Pass in a number to check in the form {{_options.Path}}?5");
                    }
                }
            }
        }
    }
}

```

由于其路径与匹配时，此中间件将充当请求委托链中的终结点，因此时不需要调用 `_next.Invoke` 时此中间件将处理该请求。

使用位置和一些有用的扩展方法中创建以方便将其配置，重构此中间件 `Configure` 方法如下所示：

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

    app.UsePrimeChecker();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
}
```

此重构，以下确保，web 应用程序仍然正常工作和前面一样，因为所有通过集成测试。

注意

它是一个好办法之后完成重构，你的测试都通过将所做的更改提交到源代码管理。如果您一起练习测试驱动开发，[考虑向你红-绿-重构循环添加提交](#)。

资源

- [单元测试](#)
- [中间件](#)
- [测试控制器](#)

在 ASP.NET Core razor 页单元和集成测试

2018/4/10 • 15 min to read • [Edit Online](#)

作者: [Luke Latham](#)

ASP.NET 核心支持单元和集成测试的 Razor 页应用。测试数据访问层 (DAL)、页面模型和集成页上组件可帮助确保:

- 部分 Razor 页面应用程序在应用程序构造期间工作独立以及组合在一起作为一个单元。
- 类和方法具有有限的作用域的责任。
- 上应用的行为方式存在其他文档。
- 回归, 不由更新对代码的错误, 将自动的生成和部署期间发现。

本主题假定你已基本了解 Razor 页应用、单元测试, 以及集成测试。如果你熟悉 Razor 页应用或测试概念, 请参阅以下主题:

- [Razor 页面介绍](#)
- [Razor 页面入门](#)
- [单元测试 C# 中使用 dotnet 测试和 xUnit 的.NET 核心](#)
- [集成测试](#)

[查看或下载示例代码\(如何下载\)](#)

示例项目组成两个应用程序:

应用	项目文件夹	描述
消息应用程序	<i>src/RazorPagesTestingSample</i>	允许用户添加、删除其中一个, 删除所有, 和分析消息。
测试应用程序	<i>tests/RazorPagesTestingSample.Tests</i>	用于测试消息应用程序。 <ul style="list-style-type: none">单元测试: 数据访问层 (DAL), 索引页模型集成测试: 索引页

可以使用内置的测试功能的一个 IDE, 如运行测试[Visual Studio](#)。如果使用[Visual Studio Code](#)或命令行中, 执行以下命令在命令提示符中*tests/RazorPagesTestingSample.Tests*文件夹:

```
dotnet test
```

消息应用组织

消息应用程序是一个简单的 Razor 页消息系统, 具有以下特征:

- 应用程序的索引页 (*Pages/Index.cshtml* 和 *Pages/Index.cshtml.cs*) 提供一个用户界面和网页模型方法, 用于控制添加、删除和分析消息 (每个消息的平均词)。
- 一条消息都由描述 `Message` 类 (*Data/Message.cs*) 具有两个属性: `Id` (密钥) 和 `Text` (消息)。`Text` 属性是必需的限制为 200 个字符。
- 消息存储使用[实体框架的内存中数据库](#)†。
- 此应用程序包含在其数据库上下文类中, 数据访问层 (DAL) `AppDbContext` (*Data/AppDbContext.cs*)。DAL 方

法将被标记 `virtual`，这样，模拟在测试中使用的方法。

- 如果数据库为空应用程序启动时，消息存储区初始化与三条消息。这些 **设定种子消息**也用在测试。

EF 主题，[测试以及 InMemory](#)，说明如何使用内存中数据库来使用 MSTest 测试。本主题使用 [xUnit](#) 测试框架。测试概念和测试跨不同测试框架的实现是类似，但不是完全相同。

尽管应用程序不使用[存储库模式](#)并不是有效的示例**工作单元 (UoW) 模式**，Razor 页中支持的开发模式。有关详细信息，请参阅[设计基础结构持久性层](#)，[在 ASP.NET MVC 应用程序中实现的存储库和单元的工作模式](#)，和[测试控制器逻辑](#)(此示例实现的存储库模式)。

测试应用程序的组织

测试应用程序是在一个控制台应用 `tests/RazorPagesTestingSample.Tests` 文件夹：

测试应用程序文件夹	描述
<code>IntegrationTests</code>	<ul style="list-style-type: none"><code>IndexPageTest.cs</code> 包含索引页的集成测试。<code>TestFixture.cs</code> 创建测试主机消息应用程序。
<code>UnitTests</code>	<ul style="list-style-type: none"><code>DataAccessLayerTest.cs</code> DAL 包含单元测试。<code>IndexPageTest.cs</code> 包含针对索引页模型的单元测试。
实用程序	<p><code>Utilities.cs</code> 包含：</p> <ul style="list-style-type: none"><code>TestingDbContextOptions</code> 用于创建新的数据库上下文对于每个 DAL 单元测试的选项，以便数据库重置为其基线条件的每个测试方法。<code>GetRequestContentAsync</code> 方法用于准备 <code>HttpClient</code> 和内容的集成测试在发送到消息应用程序的请求。

测试框架为 [xUnit](#)。模拟框架的对象是 [Moq](#)。集成测试使用执行 [ASP.NET 核心测试主机](#)。

单元测试的数据访问层 (DAL)

消息应用程序中包含的四个方法与出现 DAL `AppDbContext` 类

(`src/RazorPagesTestingSample/Data/AppDbContext.cs`)。每个方法中测试应用程序包含一个或两个单元测试。

DAL 方法	函数
<code>GetMessagesAsync</code>	获取 <code>List<Message></code> 从数据库按排序 <code>Text</code> 属性。
<code>AddMessageAsync</code>	将添加 <code>Message</code> 到数据库。
<code>DeleteAllMessagesAsync</code>	删除所有 <code>Message</code> 从数据库的条目。
<code>DeleteMessageAsync</code>	将删除单个 <code>Message</code> 从数据库 <code>Id</code> 。

DAL 的单元测试需要 `DbContextOptions` 时创建新 `AppDbContext` 为每个测试。创建的一种方法 `DbContextOptionsBuilder` 每个测试是使用 `DbContextOptionsBuilder`：

```

var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}

```

使用此方法的问题是，每个测试就会收到数据库中以前的测试中保留它的任何状态。在尝试写入不会相互影响的原子单元测试时，这可能会产生问题。若要强制 `AppDbContext` 若要对每个测试中使用新的数据库上下文，提供 `DbContextOptions` 基于新的服务提供程序的实例。测试应用程序演示如何执行此操作使用其 `Utilities` 类方法 `TestingDbContextOptions` (`tests/RazorPagesTestingSample.Tests/Utilities/Utilities.cs`)：

```

public static DbContextOptions<AppDbContext> TestingDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}

```

使用 `DbContextOptions` DAL 单元中测试允许每个测试运行以原子方式与全新的数据库实例：

```

using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
{
    // Use the db here in the unit test.
}

```

在每个测试方法 `DataAccessLayerTest` 类 (`UnitTests/DataAccessLayerTest.cs`) 遵循类似的排列 Act 断言模式：

1. 排列：测试配置了数据库和/或定义预期的结果。
2. Act: 执行测试。
3. 断言：断言可确定测试结果是否成功。

例如，`DeleteMessageAsync` 方法负责删除单个消息由其 `Id` (`src/RazorPagesTestingSample/Data/AppDbContext.cs`)：

```

public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}

```

没有为此方法的两个测试。一个测试检查数据库中存在消息时，该方法将删除一条消息。如果不会更改数据库的

其他方法测试消息 `Id` 删除不存在。 `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` 方法如下所示：

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new ApplicationDbContext(Utilities.TestingDbContextOptions()))
    {
        // Arrange
        var seedMessages = ApplicationContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(x => x.Id),
            actualMessages.OrderBy(x => x.Id),
            new Utilities.MessageComparer());
    }
}
```

首先，该方法执行准备步骤中，执行步骤准备发生。获取并保存在种子设定消息 `seedMessages`。种子设定的消息会保存到数据库中。将消息与 `Id` 的 `1` 设置为删除。当 `DeleteMessageAsync` 执行方法时，预期的消息应具有所有除外与消息 `Id` 的 `1`。`expectedMessages` 变量表示此预期的结果。

```
// Arrange
var seedMessages = ApplicationContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

方法是运行：`DeleteMessageAsync` 执行方法中传递 `recId` 的 `1`：

```
// Act
await db.DeleteMessageAsync(recId);
```

最后，此方法获取 `Messages` 上下文中，并将其到 `expectedMessages` 两个相等的断言：

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

若要比较的两个 `List<Message>` 相同：

- 消息有序 `Id`。
- 消息对比较上 `Text` 属性。

类似的测试方法，`DeleteMessageAsync_NoMessageIsDeleted_WhenMessageNotFound` 检查正在尝试删除一条消息，不存在的结果。在这种情况下，在数据库中预期的消息应等于后的实际消息 `DeleteMessageAsync` 执行方法。应没有更

改数据库的内容：

```
[Fact]
public async Task DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestingDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

单元测试页模型方法

另一个组的单元测试负责测试页模型方法。在消息应用中，索引页模型中找到 `IndexModel` 类 `src/RazorPagesTestingSample/Pages/Index.cshtml.cs`。

页模型方法	函数
<code>OnGetAsync</code>	从 UI 使用 DAL 获取消息 <code>GetMessagesAsync</code> 方法。
<code>OnPostAddMessageAsync</code>	如果 <code>ModelState</code> 无效，请调用 <code>AddMessageAsync</code> 向数据库添加一条消息。
<code>OnPostDeleteAllMessagesAsync</code>	调用 <code>DeleteAllMessagesAsync</code> 删除所有数据库中的消息。
<code>OnPostDeleteMessageAsync</code>	执行 <code>DeleteMessageAsync</code> 若要删除的消息 <code>Id</code> 指定。
<code>OnPostAnalyzeMessagesAsync</code>	如果一个或多个消息是在数据库中，将计算每个消息的单词的平均的数目。

使用七个测试中的测试页模型方法 `IndexPageTest` 类

(`tests/RazorPagesTestingSample.Tests/UnitTests/IndexPageTest.cs`)。测试使用熟悉的排列断言 `Act` 模式。这些测试的重点：

- 确定方法是否遵循正确的行为时 `ModelState` 无效。
- 确认方法生成正确 `IActionResult`。
- 正在检查属性的值分配都正确。

测试此组通常模拟 DAL 来生成 `Act` 步骤执行页模型方法的预期的数据的方法。例如，`GetMessagesAsync` 方法 `AppDbContext` 模拟以生成输出。当页模型方法执行此方法时，模型将返回的结果。数据不会从数据库中。这将创建在页模型测试中使用 DAL 的可预测、可靠的测试条件。

`OnGetAsync_PopulatesThePageModel_WithAListOfMessages` 测试显示如何 `GetMessagesAsync` 方法模拟的页面模型：

```
var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

当 `OnGetAsync` Act 步骤中执行方法时，它调用页模型 `GetMessagesAsync` 方法。

单元测试 Act 步骤 (`tests/RazorPagesTestingSample.Tests/UnitTests/IndexPageTest.cs`):

```
// Act
await pageModel.OnGetAsync();
```

`IndexPage` 页模型 `OnGetAsync` 方法 (`src/RazorPagesTestingSample/Pages/Index.cshtml.cs`):

```
public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

`GetMessagesAsync` 中 DAL 方法不返回此方法调用的结果。该方法的模拟的版本返回的结果。

在 `Assert` 步骤，实际的消息 (`actualMessages`) 从分配 `Messages` 页模型的属性。在将消息分配时也执行类型检查。通过进行比较的预期和实际的消息其 `Text` 属性。测试断言，这两个 `List<Message>` 实例包含对相同消息。

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

此组中的其他测试创建页包括的模型对象 `DefaultHttpContext`、`ModelStateDictionary`、`ActionContext` 建立 `PageContext`、 `ViewDataDictionary`，和一个 `PageContext`。这些可用于执行测试。例如，消息应用程序建立 `ModelState` 错误 `AddModelError` 检查是否是有效 `PageResult` 时返回 `OnPostAddMessageAsync` 执行：

```

[Fact]
public async Task OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var ModelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new PageActionDescriptor(),
    ModelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var ViewData = new ViewDataDictionary(modelMetadataProvider, ModelState);
    var TempData = new TempDataDictionary(httpContext, Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = ViewData,
        TempData = TempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is required.");

    // Act
    var result = await pageModel.OnPostAddMessageAsync();

    // Assert
    Assert.IsType<PageResult>(result);
}

```

测试应用程序的集成

集成测试在测试应用程序的组件配合工作的焦点。集成测试使用执行[ASP.NET 核心测试主机](#)。完整的请求-响应生命周期处理进行测试。这些测试断言页生成正确的状态代码和 `Location` 标头，如果设置。

集成测试的示例的示例检查请求的消息应用程序索引页的结果

(*tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs*):

```

[Fact]
public async Task Request_ReturnsSuccess()
{
    // Act
    var response = await _client.GetAsync("/");

    // Assert
    response.EnsureSuccessStatusCode();
}

```

没有排列步。`GetAsync` 方法调用 `HttpClient` 将 GET 请求发送到终结点。测试断言，结果是 200 OK 状态代码。

对消息应用任何 POST 请求必须满足 antiforgery 检查自动是由应用程序的开发[数据保护 antiforgery 系统](#)。若要安排某个测试的 POST 请求，测试应用程序必须：

1. 请对页的请求。
2. 分析 antiforgery cookie 和响应的请求验证令牌。
3. 到位，请使用 antiforgery 的 cookie 与请求验证的 POST 请求令牌。

Post_AddMessageHandler_ReturnsRedirectToRoot 测试方法:

- 准备一条消息和 `HttpClient`。
- 向应用程序发出的 POST 请求。
- 检查响应重定向回索引页面。

Post_AddMessageHandler_ReturnsRedirectToRoot 方法

(tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs):

```
[Fact]
public async Task Post_AddMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var data = new Dictionary<string, string>()
    {
        { "Message.Text", "Test message to add." }
    };
    var content = await Utilities.GetRequestContentAsync(_client, "/", data);

    // Act
    var response = await _client.PostAsync("?handler=AddMessage", content);

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

`GetRequestContentAsync` 实用工具方法管理准备客户端使用 antiforgery cookie 和请求验证令牌。请注意如何方法接收 `IDictionary` 允许调用测试方法通过数据用于编码以及请求验证令牌的请求中

(tests/RazorPagesTestingSample.Tests/Utilities/Utilities.cs):

```

public static async Task<FormUrlEncodedContent> GetRequestContentAsync(
    HttpClient _client, string path, IDictionary<string, string> data)
{
    // Make a request for the resource.
    var getResponse = await _client.GetAsync(path);

    // Set the response's antiforgery cookie on the HttpClient.
    _client.DefaultRequestHeaders.Add("Cookie",
        getResponse.Headers.GetValues("Set-Cookie"));

    // Obtain the request verification token from the response.
    // Any <form> element in the response contains a token, and
    // they're all the same within a single response.
    //
    // This method uses Regex to parse the element and its value
    // from the response markup. A better approach in a production
    // app would be to use an HTML parser (for example,
    // HtmlAgilityPack: http://html-agility-pack.net/).
    var responseMarkup = await getResponse.Content.ReadAsStringAsync();
    var regExp_RequestVerificationToken = new Regex(
        "<input name=__RequestVerificationToken type=hidden value=(.*?)\\>",
        RegexOptions.Compiled);
    var matches = regExp_RequestVerificationToken.Matches(responseMarkup);
    // Group[1] represents the captured characters, represented
    // by (.*?) in the Regex pattern string.
    var token = matches?.FirstOrDefault().Groups[1].Value;

    // Add the token to the form data for the request.
    data.Add("__RequestVerificationToken", token);

    return new FormUrlEncodedContent(data);
}

```

集成测试还可以将错误数据传递给应用程序以测试应用程序的响应行为。消息应用程序限制到 200 个字符的消息长度 (*src/RazorPagesTestingSample/Data/Message.cs*):

```

public class Message
{
    public int Id { get; set; }

    [Required]
    [DataType(DataType.Text)]
    [StringLength(200, ErrorMessage = "There's a 200 character limit on messages. Please shorten your message.")]
    public string Text { get; set; }
}

```

`Post_AddMessageHandler_ReturnsSuccess_WhenMessageTextTooLong` 测试 `Message` 显式传递中带有 201 "X" 字符的文本。这会导致 `ModelState` 错误。发布不重定向回索引页面。它将返回 200 OK 与 `null``Location` 标头 (*tests/RazorPagesTestingSample.Tests/IntegrationTests/IndexPageTest.cs*):

```
[Fact]
public async Task Post_AddMessageHandler_ReturnsSuccess_WhenMessageTextTooLong()
{
    // Arrange
    var data = new Dictionary<string, string>()
    {
        { "Message.Text", new string('X', 201) }
    };
    var content = await Utilities.GetRequestContentAsync(_client, "/", data);

    // Act
    var response = await _client.PostAsync("?handler=AddMessage", content);

    // Assert
    // A ModelState failure returns to Page (200-OK) and doesn't redirect.
    response.EnsureSuccessStatusCode();
    Assert.Null(response.Headers.Location?.OriginalString);
}
```

请参阅

- [单元测试 C# 中使用 dotnet 测试和 xUnit 的.NET 核心](#)
- [集成测试](#)
- [测试控制器](#)
- [单元测试代码\(Visual Studio\)](#)
- [xUnit.net](#)
- [入门 xUnit.net \(.NET Core/ASP.NET 核\)](#)
- [Moq](#)
- [Moq 快速入门](#)

ASP.NET Core 中的测试控制器逻辑

2018/5/14 • 19 min to read • [Edit Online](#)

作者 : [Steve Smith](#)

ASP.NET MVC 应用的控制器应该是小型的，且重点应对用户界面问题。处理非 UI 问题的大型控制器更难以测试和维护。

[查看或下载 GitHub 中的示例](#)

测试控制器

控制器是任意 ASP.NET Core MVC 应用程序的核心部分。因此，应该对它们按应用计划表现有信心。自动测试可以给你这种信心，还能在生成前检测错误。避免在控制器中安置不必要的职责并确保测试仅专注于控制器职责非常重要。

控制器应采用最简单的逻辑，而不应关注业务逻辑或基础结构问题（例如数据访问）。测试控制器逻辑，而不是框架。测试控制器基于有效或无效输入的行为。测试控制器如何相应其所执行业务操作的结果。

典型控制器职责：

- 验证 `ModelState.IsValid`。
- 如果 `ModelState` 无效，则返回错误响应。
- 从持久性检索业务实体。
- 对业务实体执行操作。
- 将业务实体保存到持久性。
- 返回相应的 `IActionResult`。

单元测试

[单元测试](#) 涉及在测试应用程序部分时，使之与它的基础结构和依赖关系相隔离。单元测试控制器逻辑时，仅测试单个操作的内容，不测试其依赖项或框架自身的行为。单元测试控制器操作时，请确保仅关注其行为。控制器单元测试将避开筛选器、路由或模型绑定等。仅专注测试一项内容，单位测试通常编写简单、运行迅速。正确编写的单位测试集无需过多开销即可经常运行。但单元测试不检测组件间交互的问题，[集成测试](#) 才会检测。

如果编写自定义筛选器、路由等，应对其进行单位测试，而不是作为特定控制器操作测试的一部分进行。应该对其进行隔离测试。

提示

[使用 Visual Studio 创建和运行单位测试。](#)

若要演示单位测试，请查看以下控制器。它显示集体讨论会话的列表并允许使用 POST 创建新的集体讨论会话：

```

using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class HomeController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public HomeController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index()
        {
            var sessionList = await _sessionRepository.ListAsync();

            var model = sessionList.Select(session => new StormSessionViewModel()
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
}

```

控制器遵循 [显式依赖关系原则](#)，需要依赖关系注入提供 `IBrainstormSessionRepository` 实例。因此使用 mock 对象框架（例如 [Moq](#)）进行测试很简单。`HTTP GET Index` 方法没有循环或分支，且仅调用一个方法。若要测试此 `Index` 方法，我们需要验证返回了 `ViewResult`，附带来自存储库 `List` 方法的 `ViewModel`。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class HomeControllerTests
    {
        [Fact]
        public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
        {
            // Arrange
            var mockRepo = new Mock<IBrainstormSessionRepository>();
            mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
            var controller = new HomeController(mockRepo.Object);

            // Act
            var result = await controller.Index();

            // Assert
            var viewResult = Assert.IsType<ViewResult>(result);
            var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
                viewResult.ViewData.Model);
            Assert.Equal(2, model.Count());
        }

        private List<BrainstormSession> GetTestSessions()
        {
            var sessions = new List<BrainstormSession>();
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 2),
                Id = 1,
                Name = "Test One"
            });
            sessions.Add(new BrainstormSession()
            {
                DateCreated = new DateTime(2016, 7, 1),
                Id = 2,
                Name = "Test Two"
            });
            return sessions;
        }
    }
}

```

HomeController `HTTP POST Index` 方法(如上所示)应该验证:

- 操作方法在 `ModelState.IsValid` 是 `false` 时返回有相应数据的错误请求 `ViewResult`
- `ModelState.IsValid` 为 `true` 时, 调用存储库上的 `Add` 方法并返回有正确参数的 `RedirectToActionResult`
 - 。

通过使用 `AddModelError` 添加错误, 可以测试无效模型状态, 如下第一个测试所示。

```

[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync()).Returns(Task.FromResult(GetTestSessions()));
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}

```

第一个测试确认在 `ModelState` 无效时，返回与 `GET` 请求相同的 `ViewResult`。请注意，测试不会尝试传入无效模型。即使传入也无济于事，因为模型绑定不会运行（虽然[集成测试](#)将使用练习模型绑定）。在本例中，不测试模型绑定。这些单位测试仅测试操作方法中代码的行为。

第二个测试验证 `ModelState` 有效时，（通过存储库）添加了新的 `BrainstormSession`，且该方法返回有所需属性的 `RedirectToActionResult`。通常会忽略未调用的模拟调用，但在设置调用末尾调用 `Verifiable` 就可以在测试中对其进行验证。这通过对 `mockRepo.Verify` 的调用完成，进行这种调用时，如果未调用所需方法，则测试将失败。

注意

通过此示例中使用的 Moq 库，可轻松混合可验证（或称“严格”）mock 和非可验证 mock（也称为“宽松”mock 或存根）。详细了解[使用 Moq 自定义 Mock 行为](#)。

应用中的另一个控制器显示与特定集体讨论会话相关的信息。它包括用于处理无效 ID 值的逻辑：

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.ViewModels;

namespace TestingControllersSample.Controllers
{
    public class SessionController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public SessionController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        public async Task<IActionResult> Index(int? id)
        {
            if (!id.HasValue)
            {
                return RedirectToAction(actionName: nameof(Index), controllerName: "Home");
            }

            var session = await _sessionRepository.GetByIdAsync(id.Value);
            if (session == null)
            {
                return Content("Session not found.");
            }

            var viewModel = new StormSessionViewModel()
            {
                DateCreated = session.DateCreated,
                Name = session.Name,
                Id = session.Id
            };

            return View(viewModel);
        }
    }
}

```

控制器操作有三个要测试的事例，每个 `return` 语句对应一个：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Moq;
using TestingControllersSample.Controllers;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.ViewModels;
using Xunit;

namespace TestingControllersSample.Tests.UnitTests
{
    public class SessionControllerTests
    {
        [Fact]
        public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
        {
            // Arrange
            var controller = new SessionController(sessionRepository: null);

            // Act

```

```

        var result = await controller.Index(id: null);

        // Assert
        var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
        Assert.Equal("Home", redirectToActionResult.ControllerName);
        Assert.Equal("Index", redirectToActionResult.ActionName);
    }

    [Fact]
    public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult((BrainstormSession)null));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var contentResult = Assert.IsType<ContentResult>(result);
        Assert.Equal("Session not found.", contentResult.Content);
    }

    [Fact]
    public async Task IndexReturnsViewResultWithStormSessionViewModel()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult(GetTestSessions().FirstOrDefault(s => s.Id == testSessionId)));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsType<StormSessionViewModel>(viewResult.ViewData.Model);
        Assert.Equal("Test One", model.Name);
        Assert.Equal(2, model.DateCreated.Day);
        Assert.Equal(testSessionId, model.Id);
    }

    private List<BrainstormSession> GetTestSessions()
    {
        var sessions = new List<BrainstormSession>();
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        });
        sessions.Add(new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 1),
            Id = 2,
            Name = "Test Two"
        });
        return sessions;
    }
}

```

应用将功能公开为 Web API(与集体讨论会话相关的想法的列表, 以及将新想法添加到会话的方法):

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;

namespace TestingControllersSample.Api
{
    [Route("api/ideas")]
    public class IdeasController : Controller
    {
        private readonly IBrainstormSessionRepository _sessionRepository;

        public IdeasController(IBrainstormSessionRepository sessionRepository)
        {
            _sessionRepository = sessionRepository;
        }

        [HttpGet("forSession/{sessionId}")]
        public async Task<IActionResult> ForSession(int sessionId)
        {
            var session = await _sessionRepository.GetByIdAsync(sessionId);
            if (session == null)
            {
                return NotFound(sessionId);
            }

            var result = session.Ideas.Select(idea => new IdeaDTO()
            {
                Id = idea.Id,
                Name = idea.Name,
                Description = idea.Description,
                DateCreated = idea.DateCreated
            }).ToList();

            return Ok(result);
        }

        [HttpPost("create")]
        public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var session = await _sessionRepository.GetByIdAsync(model.SessionId);
            if (session == null)
            {
                return NotFound(model.SessionId);
            }

            var idea = new Idea()
            {
                DateCreated = DateTimeOffset.Now,
                Description = model.Description,
                Name = model.Name
            };
            session.AddIdea(idea);

            await _sessionRepository.UpdateAsync(session);

            return Ok(session);
        }
    }
}
```

```
    }  
}
```

`ForSession` 方法返回 `IdeaDTO` 类型的列表。避免直接通过 API 调用返回业务域实体，因为它们常包含 API 客户端所需以外的数据，且它们会不必要地将应用的内部域模型与外部公开的 API 配对。域实体和通过网络返回的类型之间的映射可以手动（使用 LINQ `Select`，如此处所示）或使用 [AutoMapper](#) 等库完成。

`Create` 和 `ForSession` API 方法的单位测试：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Mvc;  
using Moq;  
using TestingControllersSample.Api;  
using TestingControllersSample.ClientModels;  
using TestingControllersSample.Core.Interfaces;  
using TestingControllersSample.Core.Model;  
using Xunit;  
  
namespace TestingControllersSample.Tests.UnitTests  
{  
    public class ApiIdeasControllerTests  
    {  
        [Fact]  
        public async Task Create_ReturnsBadRequest_GivenInvalidModel()  
        {  
            // Arrange & Act  
            var mockRepo = new Mock<IBrainstormSessionRepository>();  
            var controller = new IdeasController(mockRepo.Object);  
            controller.ModelState.AddModelError("error", "some error");  
  
            // Act  
            var result = await controller.Create(model: null);  
  
            // Assert  
            Assert.IsType<BadRequestObjectResult>(result);  
        }  
  
        [Fact]  
        public async Task Create_ReturnsNotFound_ForInvalidSession()  
        {  
            // Arrange  
            int testSessionId = 123;  
            var mockRepo = new Mock<IBrainstormSessionRepository>();  
            mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))  
                .Returns(Task.FromResult((BrainstormSession)null));  
            var controller = new IdeasController(mockRepo.Object);  
  
            // Act  
            var result = await controller.Create(new NewIdeaModel());  
  
            // Assert  
            Assert.IsType<NotFoundObjectResult>(result);  
        }  
  
        [Fact]  
        public async Task Create_ReturnsNewlyCreatedIdeaForSession()  
        {  
            // Arrange  
            int testSessionId = 123;  
            string testName = "test name";  
            string testDescription = "test description";  
            var testSession = GetTestSession();  
            var mockRepo = new Mock<IBrainstormSessionRepository>();
```

```

        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .Returns(Task.FromResult(testSession));
        var controller = new IdeasController(mockRepo.Object);

        var newIdea = new NewIdeaModel()
        {
            Description = testDescription,
            Name = testName,
            SessionId = testSessionId
        };
        mockRepo.Setup(repo => repo.UpdateAsync(testSession))
            .Returns(Task.CompletedTask)
            .Verifiable();

        // Act
        var result = await controller.Create(newIdea);

        // Assert
        var okResult = Assert.IsType<OkObjectResult>(result);
        var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
        mockRepo.Verify();
        Assert.Equal(2, returnSession.Ideas.Count());
        Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
        Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
    }

    private BrainstormSession GetTestSession()
    {
        var session = new BrainstormSession()
        {
            DateCreated = new DateTime(2016, 7, 2),
            Id = 1,
            Name = "Test One"
        };

        var idea = new Idea() { Name = "One" };
        session.AddIdea(idea);
        return session;
    }
}

```

如前所述，若要测试方法在 `ModelState` 无效时的行为，请在测试中将模型错误添加到控制器。请勿在单位测试中尝试测试模型有效性或模型绑定 - 仅测试操作方法在遇到特定 `ModelState` 值时的行为。

第二个测试依赖存储库返回 null，所以 mock 存储库配置为返回 null。无需创建测试数据库（在内存中或其他位置）并构建将返回此结果的查询 - 它可以在单个语句中完成，如下所示。

最后一个测试验证调用了存储库的 `Update` 方法。正如我们之前所做，使用 `Verifiable` 调用 mock，然后调用模拟存储库的 `Verify` 方法，以确认执行了可验证方法。确保 `Update` 方法保存数据不是单位测试的职责，这可以通过集成测试完成。

集成测试

完成集成测试，可确保应用中各独立模块正确协作。通常情况下，可以使用单位测试测试的内容也可以使用集成测试测试，但反之不成立。但是，集成测试比单位测试慢。因此，最好使用单位测试测试其可测内容，并在设计多个协作者时使用集成测试。

虽然它们也可能起作用，但是 mock 对象很少在集成测试中使用。在单位测试中，mock 对象是出于测试目的控制测试单位外协作者行为的有效方法。在集成测试中，使用真实协作者确定整个子系统以正确的方式协同工作。

应用程序状态

执行集成测试时一个重要考虑因素是如何设置应用状态。测试需要互相独立地运行，因此每个测试都应从处于

已知状态的应用开始。如果应用不使用数据库或有任何持久性，这可能不是问题。但是，大多实际应用将其状态保存到某些类型的数据存储，所以一个测试作出的任意修改都可能影响到另一个测试，除非重置数据存储。使用内置 `TestServer`，在集成测试中承载 ASP.NET Core 应用非常简单，但是这并不一定会授予对其要使用的数据的访问权限。如果正在使用实际数据库，则一种方法是让应用连接到测试可以访问的测试数据库，并确保其在每个测试执行前重置到已知状态。

在此示例应用程序中，我是用的是 Entity Framework Core 的 `InMemoryDatabase` 支持，所以我不能从测试项目直接连接到它。相反，我公开应用 `Startup` 类的 `InitializeDatabase` 方法，我在应用在 `Development` 环境中启动时调用该类。只要集成测试将环境设置为 `Development`，就能自动从中获益。我无需担心重置数据库，因为每次应用重启时都会重置 `InMemoryDatabase`。

`Startup` 类：

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using TestingControllersSample.Core.Interfaces;
using TestingControllersSample.Core.Model;
using TestingControllersSample.Infrastructure;

namespace TestingControllersSample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(
                optionsBuilder => optionsBuilder.UseInMemoryDatabase("InMemoryDb"));

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

        public void Configure(IApplicationBuilder app,
            IHostingEnvironment env,
            ILoggerFactory loggerFactory)
        {
            if (env.IsDevelopment())
            {
                var repository = app.ApplicationServices.GetService<IBrainstormSessionRepository>();
                InitializeDatabaseAsync(repository).Wait();
            }

            app.UseStaticFiles();

            app.UseMvcWithDefaultRoute();
        }

        public async Task InitializeDatabaseAsync(IBrainstormSessionRepository repo)
        {
            var sessionList = await repo.ListAsync();
            if (!sessionList.Any())
            {
                await repo.AddAsync(GetTestSession());
            }
        }

        public static BrainstormSession GetTestSession()
    }
```

```

    {
        var session = new BrainstormSession()
        {
            Name = "Test Session 1",
            DateCreated = new DateTime(2016, 8, 1)
        };
        var idea = new Idea()
        {
            DateCreated = new DateTime(2016, 8, 1),
            Description = "Totally awesome idea",
            Name = "Awesome idea"
        };
        session.AddIdea(idea);
        return session;
    }
}
}

```

在以下集成测试中，可以看到频繁使用 `GetTestSession` 方法。

访问视图

每个集成测试类都会配置将运行 ASP.NET Core 应用的 `TestServer`。默认情况下，`TestServer` 会将 Web 应用存储在运行该应用的文件夹（本例中即测试项目文件夹）。因此，当尝试测试返回 `ViewResult` 的控制器操作时，可能看到此错误：

```
The view 'Index' wasn't found. The following locations were searched:  
(list of locations)
```

若要更正此问题，需要配置服务器的内容根，使其可以定位到被测试项目的视图。这可以通过调用 `TestFixture` 类中的 `UseContentRoot` 完成，如下所示：

```

using System;
using System.IO;
using System.Net.Http;
using System.Reflection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.DependencyInjection;

namespace TestingControllersSample.Tests.IntegrationTests
{
    /// <summary>
    /// A test fixture which hosts the target project (project we wish to test) in an in-memory server.
    /// </summary>
    /// <typeparam name="TStartup">Target project's startup type</typeparam>
    public class TestFixture<TStartup> : IDisposable
    {
        private readonly TestServer _server;

        public TestFixture()
            : this(Path.Combine("src"))
        {

        }

        protected TestFixture(string relativeTargetProjectParentDir)
        {
            var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
            var contentRoot = GetProjectPath(relativeTargetProjectParentDir, startupAssembly);

            var builder = newWebHostBuilder()
                .UseContentRoot(contentRoot)
        }
    }
}

```

```

        .ConfigureServices(InitializeServices)
        .UseEnvironment("Development")
        .UseStartup(typeof(TStartup));

    _server = new TestServer(builder);

    Client = _server.CreateClient();
    Client.BaseAddress = new Uri("http://localhost");
}

public HttpClient Client { get; }

public void Dispose()
{
    Client.Dispose();
    _server.Dispose();
}

protected virtual void InitializeServices(IServiceCollection services)
{
    var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;

    // Inject a custom application part manager.
    // Overrides AddMvcCore() because it uses TryAdd().
    var manager = new ApplicationPartManager();
    manager.ApplicationParts.Add(new AssemblyPart(startupAssembly));
    manager.FeatureProviders.Add(new ControllerFeatureProvider());
    manager.FeatureProviders.Add(new ViewComponentFeatureProvider());

    services.AddSingleton(manager);
}

/// <summary>
/// Gets the full path to the target project that we wish to test
/// </summary>
/// <param name="projectRelativePath">
/// The parent directory of the target project.
/// e.g. src, samples, test, or test/Websites
/// </param>
/// <param name="startupAssembly">The target project's assembly.</param>
/// <returns>The full path to the target project.</returns>
private static string GetProjectPath(string projectRelativePath, Assembly startupAssembly)
{
    // Get name of the target project which we want to test
    var projectName = startupAssembly.GetName().Name;

    // Get currently executing test project path
    var applicationBasePath = System.AppContext.BaseDirectory;

    // Find the path to the target project
    var directoryInfo = new DirectoryInfo(applicationBasePath);
    do
    {
        directoryInfo = directoryInfo.Parent;

        var projectDirectoryInfo = new DirectoryInfo(Path.Combine(directoryInfo.FullName,
projectRelativePath));
        if (projectDirectoryInfo.Exists)
        {
            var projectFileInfo = new FileInfo(Path.Combine(projectDirectoryInfo.FullName,
projectName, $"{projectName}.csproj"));
            if (projectFileInfo.Exists)
            {
                return Path.Combine(projectDirectoryInfo.FullName, projectName);
            }
        }
    }
    while (directoryInfo.Parent != null);
}

```

```
        throw new Exception($"Project root could not be located using the application root  
{applicationBasePath}.");
    }
}
```

`TestFixture` 类负责配置和创建 `TestServer`，设置 `HttpClient` 与 `TestServer` 通信。每个集成测试使用 `Client` 属性连接到测试服务器并发出请求。

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class HomeControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        private readonly HttpClient _client;

        public HomeControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ReturnsInitialListOfBrainstormSessions()
        {
            // Arrange - get a session known to exist
            var testSession = Startup.GetTestSession();

            // Act
            var response = await _client.GetAsync("/");

            // Assert
            response.EnsureSuccessStatusCode();
            var responseString = await response.Content.ReadAsStringAsync();
            Assert.Contains(testSession.Name, responseString);
        }

        [Fact]
        public async Task PostAddsNewBrainstormSession()
        {
            // Arrange
            string testSessionName = Guid.NewGuid().ToString();
            var data = new Dictionary<string, string>();
            data.Add("SessionName", testSessionName);
            var content = new FormUrlEncodedContent(data);

            // Act
            var response = await _client.PostAsync("/", content);

            // Assert
            Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
            Assert.Equal("/", response.Headers.Location.ToString());
        }
    }
}
```

在上面的第一个测试中，`responseString` 保持“视图”的实际呈现 HTML，可以检查它以确定其包含所需结果。

第三个测试构造具有唯一会话名称的窗体 POST，并将其发布到应用，然后验证返回了预期重定向。

API 方法

如果应用公开 Web API，最好用自动测试确定它们按照预期执行。内置 `TestServer` 使得测试 Web API 更简单。如果 API 方法使用模型绑定，则应始终检查 `ModelState.IsValid`，并用集成测试确定模型验证正确工作。

以下测试集以 `IdeasController` 类中的 `Create` 方法为目标，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json;
using TestingControllersSample.ClientModels;
using TestingControllersSample.Core.Model;
using Xunit;

namespace TestingControllersSample.Tests.IntegrationTests
{
    public class ApiIdeasControllerTests : IClassFixture<TestFixture<TestingControllersSample.Startup>>
    {
        internal class NewIdeaDto
        {
            public NewIdeaDto(string name, string description, int sessionId)
            {
                Name = name;
                Description = description;
                SessionId = sessionId;
            }

            public string Name { get; set; }
            public string Description { get; set; }
            public int SessionId { get; set; }
        }

        private readonly HttpClient _client;

        public ApiIdeasControllerTests(TestFixture<TestingControllersSample.Startup> fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingNameValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("", "Description", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }

        [Fact]
        public async Task CreatePostReturnsBadRequestForMissingDescriptionValue()
        {
            // Arrange
            var newIdea = new NewIdeaDto("Name", "", 1);

            // Act
            var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

            // Assert
            Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
        }
    }
}
```

```

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooSmall()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 0);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsBadRequestForSessionIdValueTooLarge()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 1000001);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsNotFoundForInvalidSession()
{
    // Arrange
    var newIdea = new NewIdeaDto("Name", "Description", 123);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task CreatePostReturnsCreatedIdeaWithCorrectInputs()
{
    // Arrange
    var testIdeaName = Guid.NewGuid().ToString();
    var newIdea = new NewIdeaDto(testIdeaName, "Description", 1);

    // Act
    var response = await _client.PostAsJsonAsync("/api/ideas/create", newIdea);

    // Assert
    response.EnsureSuccessStatusCode();
    var returnedSession = await response.Content.ReadAsJsonAsync<BrainstormSession>();
    Assert.Equal(2, returnedSession.Ideas.Count);
    Assert.Contains(testIdeaName, returnedSession.Ideas.Select(i => i.Name).ToList());
}

[Fact]
public async Task ForSessionReturnsNotFoundForBadSessionId()
{
    // Arrange & Act
    var response = await _client.GetAsync("/api/ideas/forsession/500");

    // Assert
    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}

[Fact]
public async Task ForSessionReturnsTdeasForValidSessionId()

```

```
public async Task TestSessionMethodInsteadOfVirtualSession()
{
    // Arrange
    var testSession = Startup.GetTestSession();

    // Act
    var response = await _client.GetAsync("/api/ideas/forsession/1");

    // Assert
    response.EnsureSuccessStatusCode();
    var ideaList = JsonConvert.DeserializeObject<List<IdeaDTO>>(
        await response.Content.ReadAsStringAsync());
    var firstIdea = ideaList.First();
    Assert.Equal(testSession.Ideas.First().Name, firstIdea.Name);
}
}
```

不同于返回 HTML 视图的操作集成测试，返回结果的 Web API 方法常常可以反序列化为强类型对象，如上面的最后一个测试所示。在此情况下，测试将结果反序列化到 `BrainstormSession` 实例，并确定想法正确添加到其想法集。

可在本文的[示例项目](#)中找到集成测试的其他示例。

解决 ASP.NET 核心项目

2018/5/18 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

以下链接提供了故障排除指导:

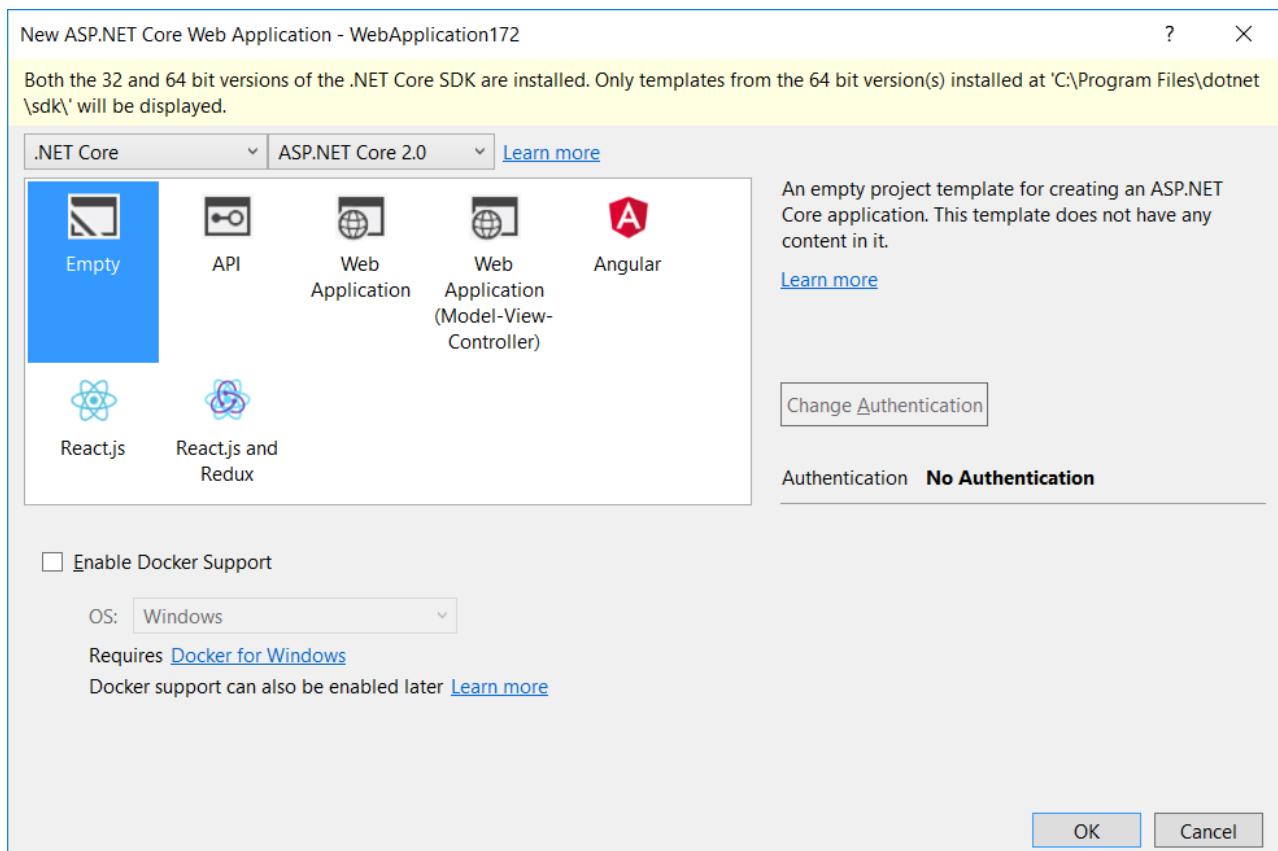
- 对 Azure 应用服务上的 ASP.NET Core 进行故障排除
- 对 IIS 上的 ASP.NET Core 进行故障排除
- Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考
- YouTube: [诊断 ASP.NET Core 应用程序中的问题](#)

.NET 核心 SDK 警告

安装的 32 位和 64 位版本的.NET 核心 SDK

在新项目对话框为 ASP.NET Core, 你可能会看到以下警告:

Both 32 and 64 bit versions of the .NET Core SDK are installed. Only templates from the 64 bit version(s) installed at C:\Program Files\dotnet\sdk\ will be displayed.



时, 此警告会出现 (x86) 32 位和 64 位 (x64) 版本的.NET 核心 SDK 安装。可以安装两个版本的常见原因包括:

- 你最初下载.NET 核心 SDK 安装程序使用 32 位计算机, 但然后复制它跨并安装在 64 位计算机上。
- 由另一个应用程序安装了 32 位.NET 核心 SDK。
- 下载并安装了错误版本。

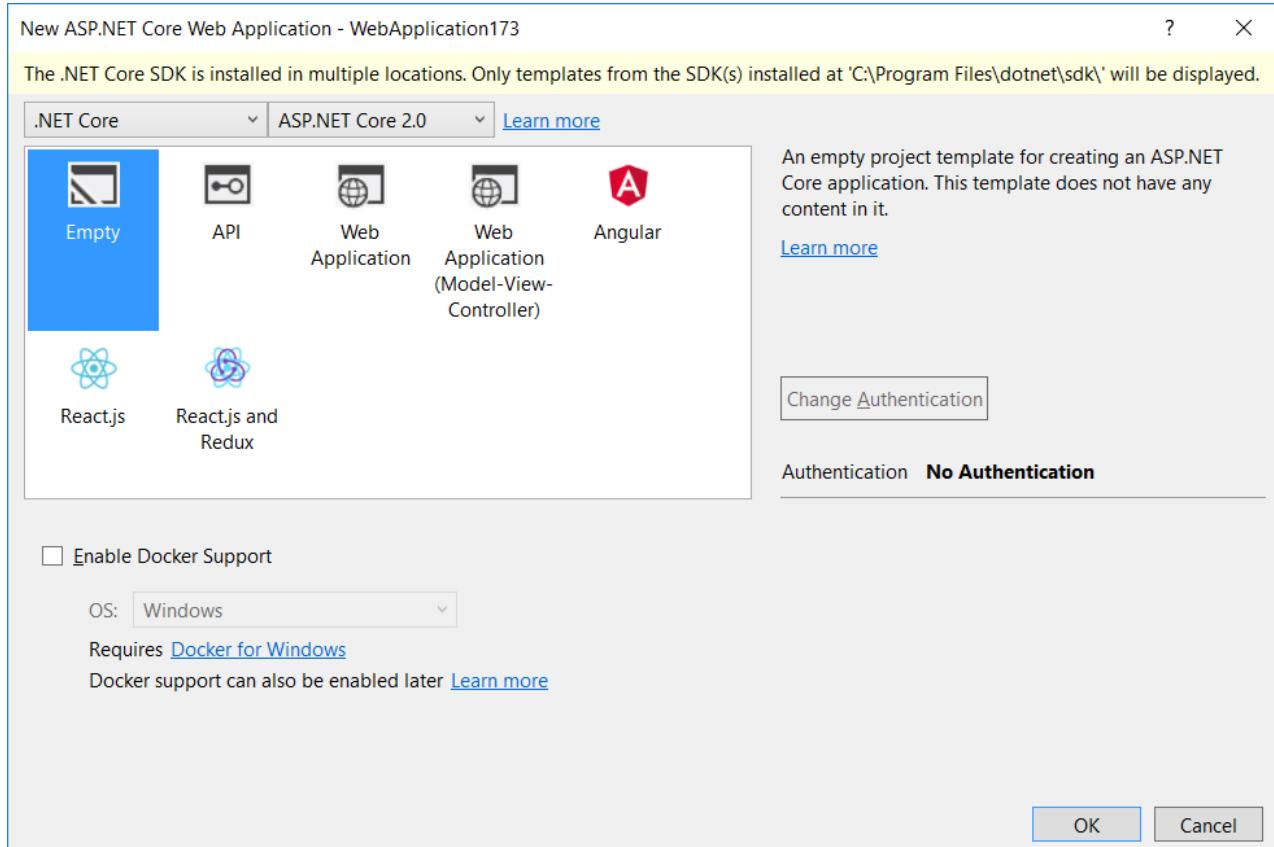
卸载 32 位.NET 核心 SDK, 以防止出现此警告。从卸载控制面板 > 程序和功能 > 卸载或更改程序。如果你了解

为何会出现的警告和它的含义，你可以忽略该警告。

.NET 核心 SDK 的安装在多个位置

在新项目对话框为 ASP.NET Core 可能会看到以下警告：

.NET 核心 SDK 安装在多个位置中。从安装在 SDK(s) 的模板 C:\Program Files\dotnet\sdk' 将显示。



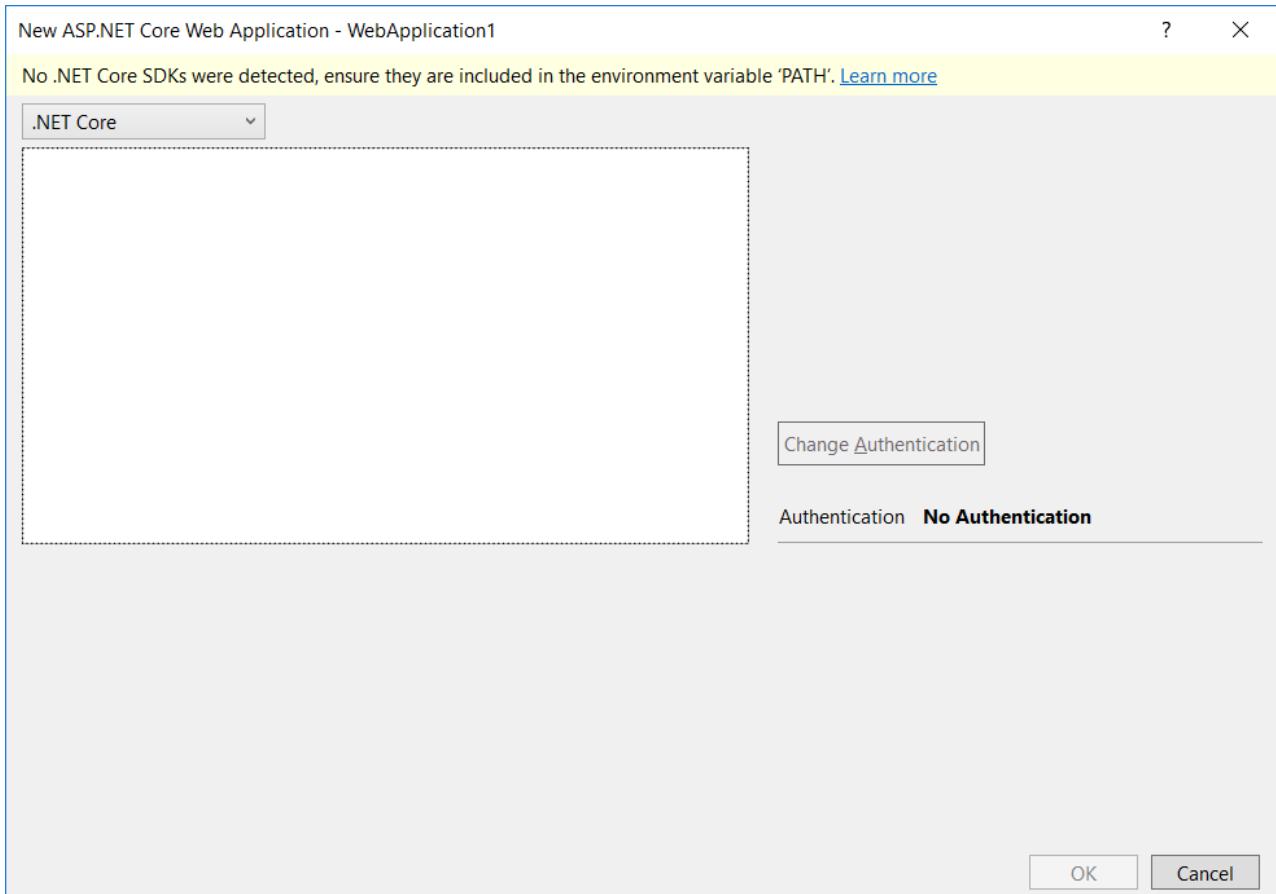
外部的一个目录中有至少一个安装的.NET 核心 SDK 时，将显示此消息 * C:\Program Files\dotnet\sdk*。通常，这发生在使用复制/粘贴，而不 MSI 安装程序的计算机上部署了.NET 核心 SDK 时。

卸载 32 位.NET 核心 SDK，以防止出现此警告。从卸载控制面板 > 程序和功能 > 卸载或更改程序。如果你了解为何会出现的警告和它的含义，你可以忽略该警告。

检测到没有.NET 核心 Sdk

在新项目对话框为 ASP.NET Core 可能会看到以下警告：

检测到没有.NET 核心 Sdk，请确保将它们包括在环境变量 PATH



时，此警告会出现的环境变量 `PATH` 没有指向计算机上任何.NET 核心 Sdk。若要解决此问题：

- 安装或验证.NET 核心 SDK 的安装。
- 验证 `PATH` 环境变量指向 SDK 的安装位置。安装程序通常设置 `PATH`。

利用 `IHtmlHelper.Partial` 可能会导致应用程序死锁

在 ASP.NET 核心 2.1 及更高版本，则调用 `Html.Partial` 导致死锁的可能性由于分析器警告。警告消息为：

`利用 IHtmlHelper.Partial 可能会导致应用程序死锁。请考虑使用 <partial> 标记帮助器或 IHtmlHelper.PartialAsync`
◦

调用 `@Html.Partial` 应替换为 `@await Html.PartialAsync` 或部分标记帮助器 `<partial name="_Partial" />`。

在 ASP.NET Core 中使用数据

2018/4/10 • 1 min to read • [Edit Online](#)

- [通过 Visual Studio 开始使用 Razor 页面和 Entity Framework Core](#)
 - [Razor 页面及 EF 入门](#)
 - [创建、读取、更新和删除操作](#)
 - [排序、筛选器、页和组](#)
 - [迁移](#)
 - [创建复杂数据模型](#)
 - [读取相关数据](#)
 - [更新相关数据](#)
 - [处理并发冲突](#)
- [ASP.NET Core MVC 和 Entity Framework Core 入门\(使用 Visual Studio\)](#)
 - [入门](#)
 - [创建、读取、更新和删除操作](#)
 - [排序、筛选器、页和组](#)
 - [迁移](#)
 - [创建复杂数据模型](#)
 - [读取相关数据](#)
 - [更新相关数据](#)
 - [处理并发冲突](#)
 - [继承](#)
 - [高级主题](#)
- [ASP.NET Core 和 EF Core - 新数据库](#)(Entity Framework Core 文档站点)
- [ASP.NET Core 和 EF Core - 现有数据库](#)(Entity Framework Core 文档站点)
- [开始使用 ASP.NET Core 和 Entity Framework 6](#)
- [Azure 存储](#)
 - [使用 Visual Studio 连接服务添加 Azure 存储](#)
 - [开始使用 Azure Blob 存储和 Visual Studio 连接服务](#)
 - [开始使用队列存储和 Visual Studio 连接服务](#)
 - [开始使用 Azure 表存储和 Visual Studio 连接服务](#)

ASP.NET Core 中的 Razor 页面和 Entity Framework Core - 第 1 个教程(共 8 个)

2018/5/14 • 21 min to read • [Edit Online](#)

作者: [Tom Dykstra](#) 和 [Rick Anderson](#)

Contoso University 示例 Web 应用演示了如何使用 Entity Framework (EF) Core 2.0 和 Visual Studio 2017 创建 ASP.NET Core 2.0 MVC Web 应用程序。

该示例应用是一个虚构的 Contoso University 的网站。其中包括学生录取、课程创建和讲师分配等功能。本页是介绍如何构建 Contoso University 示例应用系列教程中的第一部分。

[下载或查看已完成的应用。下载说明。](#)

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

熟悉 [Razor 页面](#)。新程序员在开始学习本系列之前，应先完成 [Razor 页面入门](#)。

疑难解答

如果遇到无法解决的问题，可以通过与[已完成的阶段](#)对比代码来查找解决方案。常见错误以及对应的解决方案，请参阅[最新教程中的故障排除](#)。如果在该处找不到所需内容，可以在[StackOverflow.com](#) 上发表有关 [ASP.NET Core](#) 或 [EF Core](#) 的问题。

提示

本系列教程以之前教程中已完成的工作为基础。每次成功完成教程后，请考虑保存项目副本。如果遇到问题，便可以从上一教程重新开始，而无需从头开始。或者可以下载[已完成阶段](#)并使用已完成阶段重新开始。

Contoso University Web 应用

这些教程中所构建的应用是一个基本的大学网站。

用户可以查看和更新学生、课程和讲师信息。以下是在本教程中创建的几个屏幕。

The screenshot shows a web browser window titled "Index - Contoso University". The address bar displays "localhost:1234/Students". The page header says "Contoso University". The main content area has a title "Index" and a "Create New" link. Below that is a search bar with placeholder "Find by name:" and a "Search" button, followed by a link "Back to full List". A table lists three student records:

Last Name	First Name	Enrollment Date	
Smith	Joe	2017-10-31	Edit Details Delete
Alexander	Carson	2010-09-01	Edit Details Delete
Alonso	Meredith	2012-09-01	Edit Details Delete

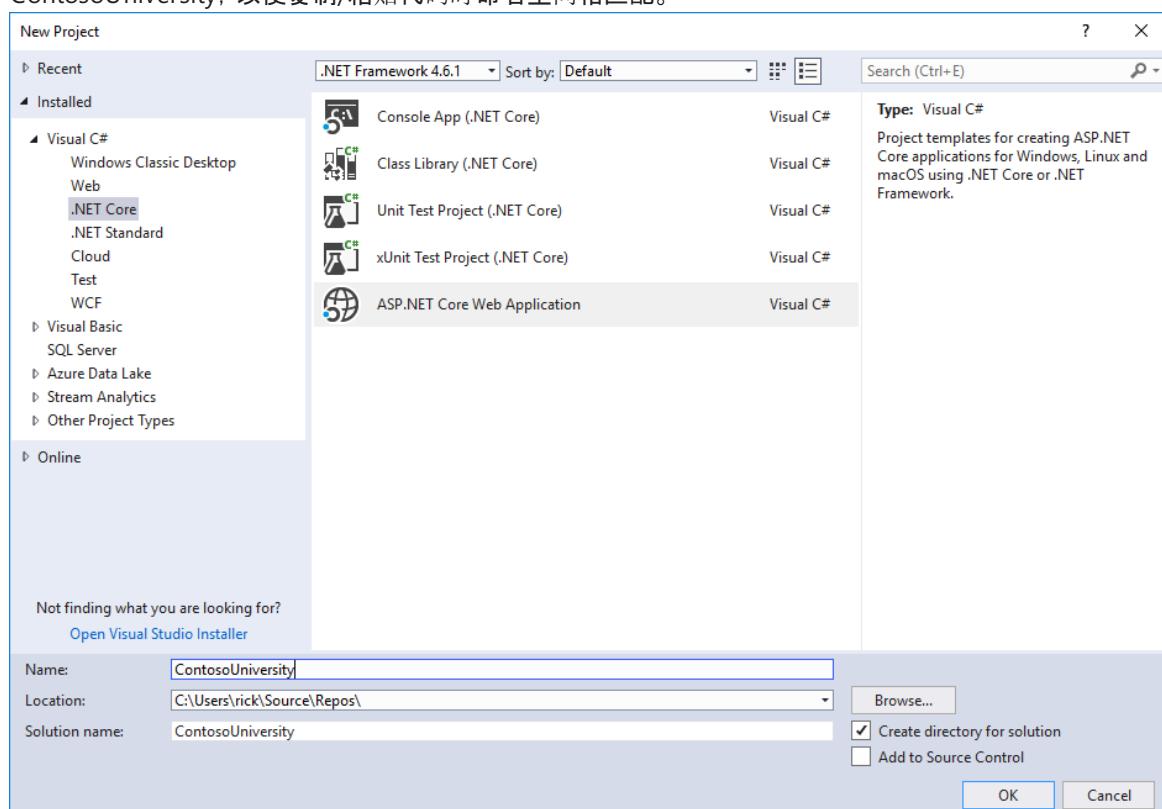
At the bottom are "Previous" and "Next" navigation buttons.

The screenshot shows a web browser window titled "Edit - Contoso University". The address bar displays "localhost:1234/Students/Edit/1". The page header says "Contoso University". The main content area has a title "Edit" and a subtitle "Student". It contains three form fields: "Last Name" (value: Alexander), "First Name" (value: Carson), and "Enrollment Date" (value: 09/01/2010). Below the form is a "Save" button and a "Back to List" link.

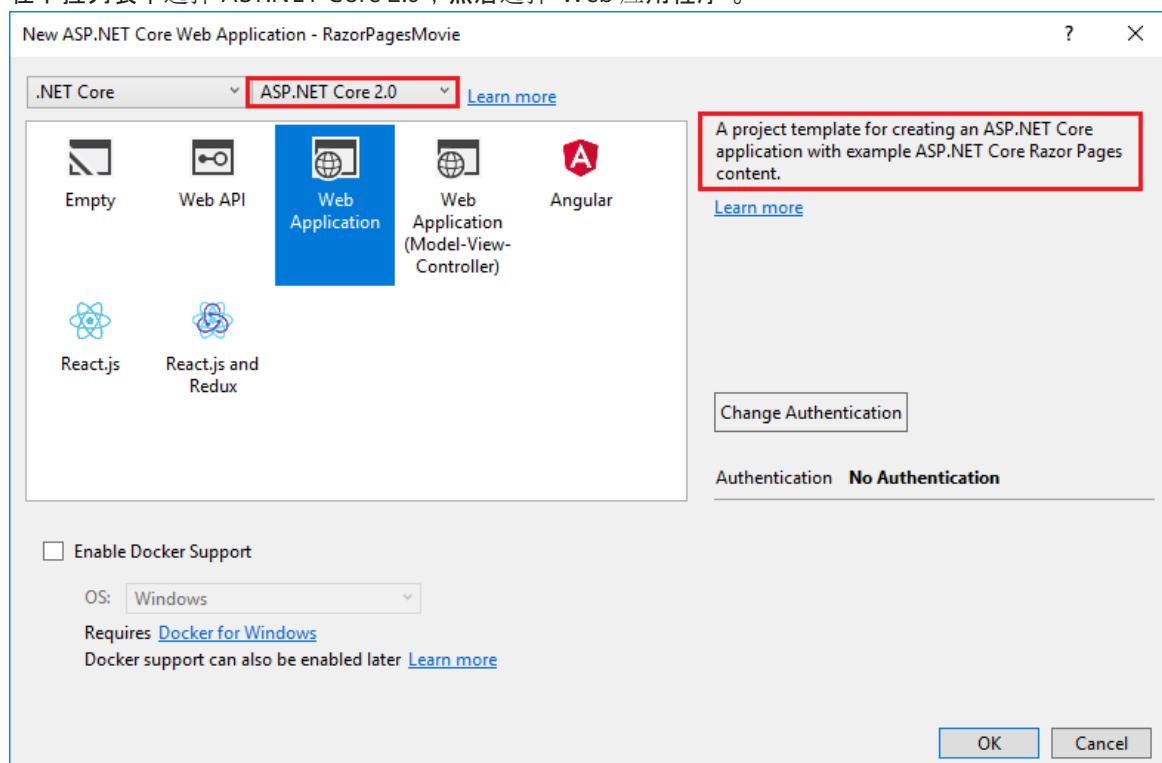
此网站的 UI 样式与内置模板生成的 UI 样式类似。教程的重点是 EF Core 和 Razor 页面，而非 UI。

创建 Razor 页面 Web 应用

- 从 Visual Studio“文件”菜单中选择“新建” > “项目”。
- 创建新的 ASP.NET Core Web 应用程序。将该项目命名为 ContosoUniversity。务必确保该项目命名为 ContosoUniversity，以便复制/粘贴代码时命名空间相匹配。



- 在下拉列表中选择“ASP.NET Core 2.0”，然后选择“Web 应用程序”。



按 F5 在调试模式下运行应用，或按 Ctrl-F5 在运行(不附加调试器)

设置网站样式

设置网站菜单、布局和主页时需作少量更改。

打开 Pages/_Layout.cshtml 并进行以下更改：

- 将“ContosoUniversity”的每个匹配项都改为“Contoso University”。共有三个匹配项。
- 添加菜单项 **Students**, **Courses**, **Instructors**, 和 **Department**, 并删除 **Contact**菜单项。

突出显示所作更改。(所有标记均不显示。)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
            href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
            asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
            asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Students/Index">Students</a></li>
                    <li><a asp-page="/Courses/Index">Courses</a></li>
                    <li><a asp-page="/Instructors/Index">Instructors</a></li>
                    <li><a asp-page="/Departments/Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2017 - Contoso University</p>
        </footer>
    </div>
```

在 Pages/Index.cshtml 中，将文件内容替换为以下代码，以将有关 ASP.NET 和 MVC 的文本替换为有关本应用的文本：

```
@page
@model IndexModel
 @{
     ViewData["Title"] = "Home page";
 }

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default"
            href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
            See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
            href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-rp/intro/samples/cu-
final">
            See project source code &raquo;</a></p>
    </div>
</div>
```

按 Ctrl+F5 运行项目。将显示主页以及后续教程中创建的标签：

The screenshot shows a web browser window titled "Home page - Contoso U" with the URL "localhost:1234". The page content is as follows:

Contoso University

Welcome to Contoso University

Contoso University is a sample application that demonstrates how to use Entity Framework Core in an ASP.NET Core Razor Pages web app.

Build it from scratch

You can build the application by following the steps in a series of tutorials.

[See the tutorial »](#)

Download it

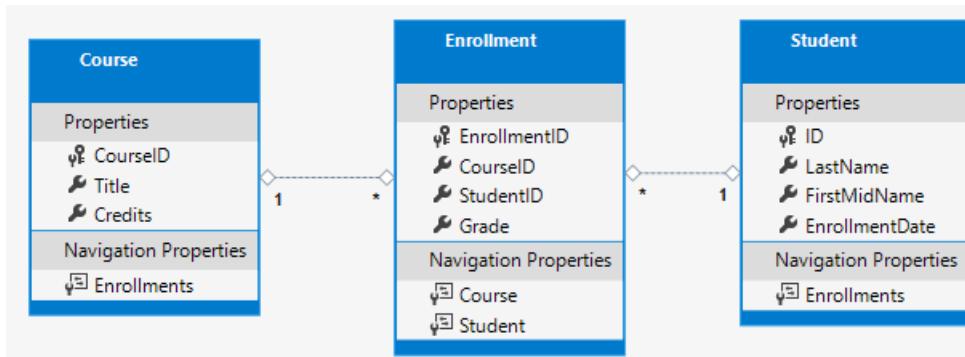
You can download the completed project from GitHub.

[See project source code »](#)

© 2017 - Contoso University

创建数据模型

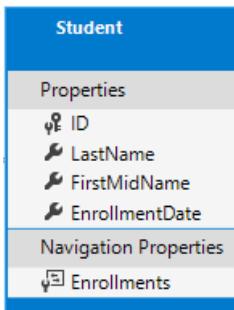
创建 Contoso University 应用的实体类。从以下三个实体开始：



Student 和 **Enrollment** 实体之间存在一对多关系。**Course** 和 **Enrollment** 实体之间存在一对多关系。一名学生可以报名参加任意数量的课程。一门课程中可以包含任意数量的学生。

以下部分将为这几个实体中的每一个实体创建一个类。

Student 实体



创建 Models 文件夹。在 Models 文件夹中，使用以下代码创建一个名为 Student.cs 的类文件：

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

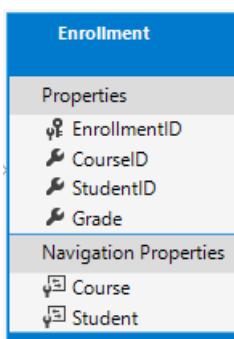
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

`ID` 属性成为此类对应的数据库 (DB) 表的主键列。默认情况下，EF Core 将名为 `ID` 或 `classnameID` 的属性视为主键。`classnameID` 中的 `classname` 是类的名称，例如上述示例中的 `Student`。

`Enrollments` 属性是导航属性。导航属性链接到与此实体相关的其他实体。在这种情况下，`Student entity` 的 `Enrollments` 属性包含与该 `Student` 相关的所有 `Enrollment` 实体。例如，如果数据库中的 `Student` 行有两个相关的 `Enrollment` 行，则 `Enrollments` 导航属性包含这两个 `Enrollment` 实体。相关的 `Enrollment` 行是 `StudentID` 列中包含该学生的主键值的行。例如，假设 `ID=1` 的学生在 `Enrollment` 表中有两行。`Enrollment` 表中有两行的 `StudentID = 1`。`StudentID` 是 `Enrollment` 表中的外键，用于指定 `Student` 表中的学生。

如果导航属性包含多个实体，则导航属性必须是列表类型，例如 `ICollection<T>`。可以指定 `ICollection<T>` 或诸如 `List<T>` 或 `HashSet<T>` 的类型。使用 `ICollection<T>` 时，EF Core 会默认创建 `HashSet<T>` 集合。包含多个实体的导航属性来自于多对多和一对多关系。

Enrollment 实体



在 Models 文件夹中，使用以下代码创建 Enrollment.cs：

```

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}

```

`EnrollmentID` 属性为主键。`Student` 实体使用的是 `ID` 模式，而本实体使用的是 `classnameID` 模式。通常情况下，开发者会选择一种模式并在整个数据模型中都使用该模式。下一个教程将介绍如何使用不带类名的 ID，以便更轻松地在数据模型中实现集成。

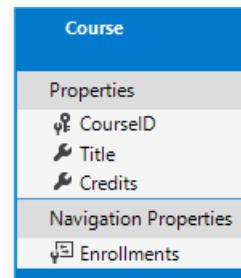
`Grade` 属性为 `enum`。`Grade` 声明类型后的 `?` 表示 `Grade` 属性可以为 `null`。评级为 `null` 和评级为零是有区别的 --`null` 意味着评级未知或者尚未分配。

`StudentID` 属性是外键，其对应的导航属性为 `Student`。`Enrollment` 实体与一个 `Student` 实体相关联，因此该属性只包含一个 `Student` 实体。`Student` 实体与 `Student.Enrollments` 导航属性不同，后者包含多个 `Enrollment` 实体。

`CourseID` 属性是外键，其对应的导航属性为 `Course`。`Enrollment` 实体与一个 `Course` 实体相关联。

如果属性命名为 `<navigation property name><primary key property name>`，EF Core 会将其视为外键。例如 `Student` 导航属性的 `StudentID`，因为 `Student` 实体的主键为 `ID`。还可以将外键属性命名为 `<primary key property name>`。例如 `CourseID`，因为 `Course` 实体的主键为 `CourseID`。

Course 实体



在 Models 文件夹中，使用以下代码创建 Course.cs：

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

`Enrollments` 属性是导航属性。`Course` 实体可与任意数量的 `Enrollment` 实体相关。

应用可以通过 `DatabaseGenerated` 特性指定主键，而无需靠数据库生成。

创建 SchoolContext 数据库上下文

数据库上下文类是为给定数据模型协调 EF Core 功能的主类。数据上下文派生自 `Microsoft.EntityFrameworkCore.DbContext`。数据上下文指定数据模型中包含哪些实体。在此项目中，类命名为 `SchoolContext`。

在项目文件夹中，创建一个名为 Data 的文件夹。

在 Data 文件夹中，使用以下代码创建 `SchoolContext.cs`：

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {}

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}

```

此代码会为每个实体集创建一个 `DbSet` 属性。在 EF Core 术语中：

- 实体集通常对应一个数据库表。
- 实体对应表中的行。

`DbSet<Enrollment>` 和 `DbSet<Course>` 可以省略。EF Core 隐式包含了它们，因为 `Student` 实体引用 `Enrollment` 实体，而 `Enrollment` 实体引用 `Course` 实体。在本教程中，将 `DbSet<Enrollment>` 和 `DbSet<Course>` 保留在 `SchoolContext` 中。

创建数据库时，EF Core 会创建名称与 `DbSet` 属性名相同的表。集合的属性名通常采用复数形式（使用 `Students`，而不使用 `Student`）。开发者对表名称是否应为复数形式意见不一。在这些教程中，在 `DbContext` 中指定单数形式的表名称会覆盖默认行为。若要指定单数形式的表名称，请添加以下突出显示的代码：

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

通过依赖关系注入注册上下文

ASP.NET Core 包含[依赖关系注入](#)。服务(例如 EF Core 数据库上下文)在应用程序启动期间通过依赖关系注入进行注册。需要这些服务(如 Razor 页面)的组件通过构造函数提供相应服务。本教程的后续部分介绍了用于获取数据库上下文实例的构造函数代码。

要将 `SchoolContext` 注册为服务, 请打开 `Startup.cs`, 并将突出显示的行添加到 `ConfigureServices` 方法。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

通过调用 `DbContextOptionsBuilder` 中的一个方法将数据库连接字符串在配置文件中的名称传递给上下文对象。进行本地开发时, [ASP.NET Core 配置系统](#)会从 `appsettings.json` 文件读取连接字符串。

为 `ContosoUniversity.Data` 和 `Microsoft.EntityFrameworkCore` 命名空间添加 `using` 语句。生成项目。

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
```

打开 `appsettings.json` 文件, 并按以下代码所示添加连接字符串:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;ConnectRetryCount=0;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

上述连接字符串使用 `ConnectRetryCount=0` 来防止 SQLClient 挂起。

SQL Server Express LocalDB

连接字符串指定 SQL Server LocalDB 数据库。LocalDB 是轻型版本 SQL Server Express 数据库引擎，专门针对应用开发，而非生产使用。LocalDB 作为按需启动并在用户模式下运行的轻量级数据库没有复杂的配置。默认情况下，LocalDB 会在 `C:/Users/<user>` 目录中创建 .mdf 数据库文件。

添加代码，以使用测试数据初始化该数据库

EF Core 会创建一个空的数据库。本部分中编写了 Seed 方法来使用测试数据填充该数据库。

在 Data 文件夹中，新建一个名为 `DbInitializer.cs` 的类文件，并添加以下代码：

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Parse("2005-09-01")},
                new Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Yan", LastName="Li", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new Student{FirstMidName="Peggy", LastName="Justice", EnrollmentDate=DateTime.Parse("2001-09-01")},
                new Student{FirstMidName="Laura", LastName="Norman", EnrollmentDate=DateTime.Parse("2003-09-01")},
                new Student{FirstMidName="Nino", LastName="Olivetto", EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student c in students)

```

```

foreach (Student s in students)
{
    context.Students.Add(s);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course{CourseID=1050,Title="Chemistry",Credits=3},
    new Course{CourseID=4022,Title="Microeconomics",Credits=3},
    new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
    new Course{CourseID=1045,Title="Calculus",Credits=4},
    new Course{CourseID=3141,Title="Trigonometry",Credits=4},
    new Course{CourseID=2021,Title="Composition",Credits=3},
    new Course{CourseID=2042,Title="Literature",Credits=4}
};
foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
    new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
    new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
    new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
    new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
    new Enrollment{StudentID=3,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=1050},
    new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
    new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
    new Enrollment{StudentID=6,CourseID=1045},
    new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
};
foreach (Enrollment e in enrollments)
{
    context.Enrollments.Add(e);
}
context.SaveChanges();
}
}
}

```

该代码会检查数据库中是否存在任何学生。如果数据库中没有任何学生，则会填充测试数据。它会将测试数据加载到数组中，而不是 `List<T>` 集合中，以便优化性能。

`EnsureCreated` 方法自动为数据库上下文创建数据库。如果数据库已存在，则返回 `EnsureCreated`，并且不修改数据库。

在 `Program.cs` 中，修改 `Main` 方法以执行以下操作：

- 从依赖关系注入容器获取数据库上下文实例。
- 调用 `seed` 方法，并将上下文传递给它。
- `Seed` 方法完成时释放上下文。

下面的代码显示更新后的 `Program.cs` 文件。

```

// Unused usings removed
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = BuildWebHost(args);

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while seeding the database.");
                }
            }

            host.Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

第一次运行该应用时，会使用测试数据创建并填充数据库。更新数据模型时：

- **删除数据库。**
- **更新 seed 方法。**
- **运行该应用，并创建新的种子数据库。**

后续教程中，在更改数据模型时会更新该数据库，而不会删除并重新创建该数据库。

添加基架工具

本部分中将使用包管理器控制台 (PMC) 来添加 Visual Studio Web 代码生成包。必须添加此包才能运行基架引擎。

从“工具”菜单中，选择“NuGet 包管理器”>“包管理器控制台”。

在包管理器控制台 (PMC) 中输入以下命令：

```

Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design
Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Utils

```

前一个命令将 NuGet 包添加到 *.csproj 文件中：

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Utils" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>
</Project>

```

构架模型

- 打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。
- 运行以下命令：

```

dotnet restore
dotnet aspnet-codegenerator razorpage -m Student -dc SchoolContext -udl -outDir Pages\Students --
referenceScriptLibraries

```

如果收到错误：

```
No executable found matching command "dotnet-aspnet-codegenerator"
```

打开项目目录(包含 Program.cs、Startup.cs 和 .csproj 文件的目录)中的命令窗口。

生成项目。此版本生成如下错误：

```
1>Pages\Students\Index.cshtml.cs(26,38,26,45): error CS1061: 'SchoolContext' does not contain a
definition for 'Student'
```

将 `_context.Student` 全局更改为 `_context.Students` (即向 `Student` 添加一个“s”)。找到并更新 7 个匹配项。我们计划在下一版本中修复此 bug。

下表详细说明了 ASP.NET Core 代码生成器的参数：

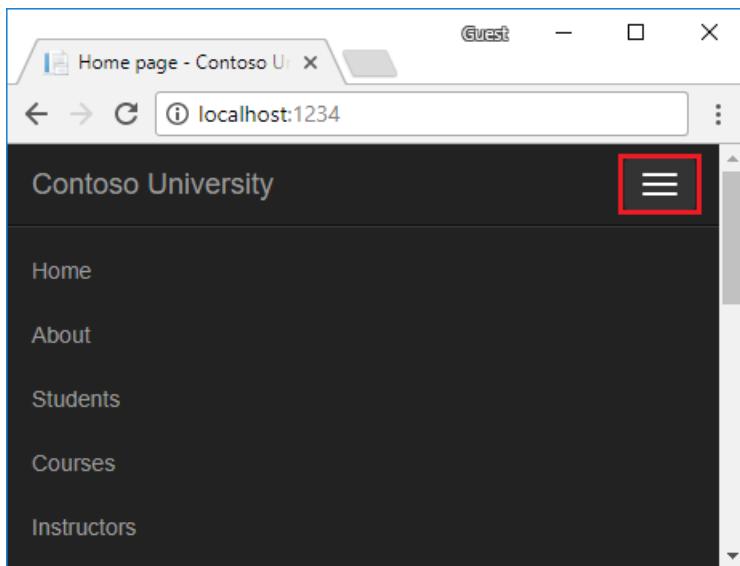
参数	描述
-m	模型的名称。
-dc	数据上下文。
-udl	使用默认布局。
-outDir	用于创建视图的相对输出文件夹路径。
--referenceScriptLibraries	向“编辑”和“创建”页面添加 <code>_ValidationScriptsPartial</code>

使用 `h` 开关获取 `aspnet-codegenerator razorpage` 命令方面的帮助：

```
dotnet aspnet-codegenerator razorpage -h
```

测试应用

运行应用并选择“学生”链接。“学生”链接将显示在页面顶部，具体取决于浏览器宽度。如果看不到“学生”链接，请单击右上角的导航图标。



测试“创建”，“编辑”和“详细信息”链接。

查看数据库

启动应用时，`DbInitializer.Initialize` 会调用 `EnsureCreated`。`EnsureCreated` 检测数据库是否存在，并在必要时创建一个数据库。如果数据库中没有学生，`Initialize` 方法将添加学生。

从 Visual Studio 中的“视图”菜单打开 SQL Server 对象资源管理器 (SSOX)。在 SSOX 中，单击“(localdb)\MSSQLLocalDB”>“数据库”>“ContosoUniversity1”。

展开“表”节点。

右键单击 Student 表，然后单击“查看数据”，以查看创建的列和插入到表中的行。

.mdf 和 .ldf 数据库文件位于 C:\Users\ 文件夹中。

启动应用时会调用 `EnsureCreated`，以进行以下工作流：

- 删除数据库。
- 更改数据库架构(例如添加一个 `EmailAddress` 字段)。
- 运行应用。

`EnsureCreated` 创建一个带有 `EmailAddress` 列的数据库。

约定

因为使用了约定或 EF Core 所做的假设，为使 EF Core 创建完整数据库而编写的代码量是最小的。

- 使用 `DbSet` 属性的名称作为表名。如果实体未被 `DbSet` 属性引用，实体类名称用作表名称。
- 使用实体属性名作为列名。
- 以 ID 或 classNameID 命名的实体属性被视为主键属性。
- 如果属性命名为，将视为外键属性(例如 `Student` 导航属性的 `StudentID`)，因为 `Student` 实体的主键为 `ID`。可将外键属性命名为(例如 `EnrollmentID`)，因为 `Enrollment` 实体的主键为 `EnrollmentID`。

可以重写常规行为。例如，可以显式指定表名，如本教程中前面部分所示。可以显式设置列名。可以显式设置主键和外键。

异步代码

异步编程是 ASP.NET Core 和 EF Core 的默认模式。

Web 服务器的可用线程是有限的，而在高负载情况下的可能所有线程都被占用。当发生这种情况的时候，服务器就无法处理新请求，直到线程被释放。使用同步代码时，可能会出现多个线程被占用但不能执行任何操作的情况，因为它们正在等待 I/O 完成。使用异步代码时，当进程正在等待 I/O 完成，服务器可以将其线程释放用于处理其他请求。因此，使用异步代码可以更有效地利用服务器资源，并且可以让服务器在没有延迟的情况下处理更多流量。

异步代码会在运行时引入少量开销。流量较低时，对性能的影响可以忽略不计，但流量较高时，潜在的性能改善非常显著。

在以下代码中，`async` 关键字、`Task<T>` 返回值、`await` 关键字和 `ToListAsync` 方法让代码异步执行。

```
public async Task OnGetAsync()
{
    Student = await _context.Students.ToListAsync();
}
```

- `async` 关键字让编译器执行以下操作：
 - 为方法主体的各部分生成回调。
 - 自动创建返回的 `Task` 对象。有关详细信息，请参阅[任务返回类型](#)。
- 隐式返回类型 `Task` 表示正在进行的工作。
- `await` 关键字让编译器将该方法拆分为两个部分。第一部分是以异步方式结束已启动的操作。第二部分是当操作完成时注入调用回调方法的地方。
- `ToListAsync` 是 `ToList` 扩展方法的异步版本。

编写使用 EF Core 的异步代码时需要注意的一些事项：

- 只会异步执行导致查询或命令被发送到数据库的语句。这包括 `ToListAsync`、`SingleOrDefaultAsync`、`FirstOrDefaultAsync` 和 `SaveChangesAsync`。不包括只会更改 `IQueryable` 的语句，例如
`var students = context.Students.Where(s => s.LastName == "Davolio")`。
- EF Core 上下文并非线程安全：请勿尝试并行执行多个操作。
- 若要利用异步代码的性能优势，请验证在调用向数据库发送查询的 EF Core 方法时，库程序包（如用于分页）是否使用异步。

有关在 .NET 中进行异步编程的详细信息，请参阅[异步概述](#)。

下一个教程将介绍基本的 CRUD（创建、读取、更新、删除）操作。

[下一篇](#)

ASP.NET Core MVC 和 EF Core - 教程系列

2018/5/17 • 1 min to read • [Edit Online](#)

本教程介绍具有控制器和视图的 ASP.NET Core MVC 和 Entity Framework Core。Razor 页面是 ASP.NET Core 2.0 中的一个新选择，它是基于页面的编程模型，可以实现更简单、更高效地生成 Web UI。建议使用 MVC 版本的 [Razor 页面教程](#)。Razor 页面教程：

- 易于关注。
- 提供更多 EF Core 最佳做法。
- 使用更高效的查询。
- 通过最新 API 更新到更高版本。
- 涵盖更多功能。
- 是开发新应用程序的首选方法。

1. [入门](#)
2. [创建、读取、更新和删除操作](#)
3. [排序、筛选、分页和分组](#)
4. [迁移](#)
5. [创建复杂数据模型](#)
6. [读取相关数据](#)
7. [更新相关数据](#)
8. [处理并发冲突](#)
9. [继承](#)
10. [高级主题](#)

ASP.NET Core 和 Entity Framework 6 入门

2018/5/17 • 5 min to read • [Edit Online](#)

作者: Paweł Grudzień、Damien Pontifex 和 Tom Dykstra

本文演示如何在 ASP.NET Core 应用程序中使用 Entity Framework 6。

概述

若要使用 Entity Framework 6，则项目必须面向 .NET Framework 进行编译，因为 Entity Framework 6 不支持 .NET Core。如果需要跨平台功能，需升级到 [Entity Framework Core](#)。

在 ASP.NET Core 应用程序中使用 Entity Framework 6 的推荐方法是：将 EF6 上下文和模型类放入面向完整框架的类库项目中。添加对 ASP.NET Core 项目中的类库的引用。请参阅示例针对 EF6 和 ASP.NET Core 项目的 Visual Studio 解决方案。

不能将 EF6 上下文放入 ASP.NET Core 项目，因为 .NET Core 项目不支持 EF6 命令（如 Enable-Migrations）所需的各项功能。

无论 EF6 上下文属于哪种项目类型，只有 EF6 命令行工具才能使用 EF6 上下文。例如，[Scaffold-DbContext](#) 仅在 Entity Framework Core 中可用。如果需要对数据库执行反向工程以使其成为 EF6 模型，请参阅[从 Code First 到现有数据库](#)。

在 ASP.NET Core 项目中引用完整框架和 EF6

ASP.NET Core 项目需要引用 .NET Framework 和 EF6。例如，ASP.NET Core 项目的 .csproj 文件将与以下示例类似（仅显示该文件的相关部分）。

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>MVCCore</AssemblyName>
  <OutputType>Exe</OutputType>
  <PackageId>MVCCore</PackageId>
</PropertyGroup>
```

创建新项目时，请使用 ASP.NET Core Web 应用程序 (.NET Framework) 模板。

处理连接字符串

需通过默认构造函数在 EF6 类库项目中使用 EF6 命令行工具，以便它们能够实例化上下文。但是，如果想指定要在 ASP.NET Core 项目中使用的连接字符串，则上下文构造函数必须具有可允许你在连接字符串中进行传递的参数。示例如下。

```
public class SchoolContext : DbContext
{
    public SchoolContext(string connString) : base(connString)
    {
    }
```

由于 EF6 上下文不具有无参数构造函数，因此 EF6 项目必须提供 [IDbContextFactory](#) 的实现。EF6 命令行工具将查找和使用该实现，以便它们能够实例化上下文。示例如下。

```
public class SchoolContextFactory : IDbContextFactory<SchoolContext>
{
    public SchoolContext Create()
    {
        return new EF6.SchoolContext("Server=
(localdb)\\mssqllocaldb;Database=EF6MVCCore;Trusted_Connection=True;MultipleActiveResultSets=true");
    }
}
```

在此示例代码中，`IDbContextFactory` 实现将在硬编码的连接字符串中传递。这是命令行工具要使用的连接字符串。你需要实施策略以确保类库与调用应用程序使用相同的连接字符串。例如，可从这两个项目的环境变量中获取值。

在 ASP.NET Core 项目中设置依赖项注入

在 Core 项目的 `Startup.cs` 文件中，为 `ConfigureServices` 中的依赖项注入 (DI) 设置 EF6 上下文。应将 EF 上下文对象的范围设置为按请求生存期。

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddScoped<SchoolContext>(_ => new
    SchoolContext(Configuration.GetConnectionString("DefaultConnection")));
}
```

然后即可使用 DI 在控制器中获取上下文的实例。此代码与针对 EF Core 上下文编写的代码相似：

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context)
    {
        _context = context;
    }
}
```

示例应用程序

若要获取有效的示例应用程序，请参阅本文随附的[示例 Visual Studio 解决方案](#)。

可在 Visual Studio 中按照以下步骤从头创建此示例：

- 创建解决方案。
- 添加新项目 > Web > **ASP.NET Core Web 应用程序 (.NET Framework)**
- 添加新项目 > Windows 经典桌面 > 类库 (.NET Framework)
- 在两个项目的“包管理器控制台”(PMC) 中运行 `Install-Package Entityframework` 命令。
- 在类库项目中，创建数据模型类和上下文类，并创建 `IDbContextFactory` 的实现。
- 在类库项目的 PMC 中，运行 `Enable-Migrations` 和 `Add-Migration Initial` 命令。如果已将 ASP.NET Core 项目设置为启动项目，请向这些命令添加 `-StartupProjectName EF6`。
- 在 Core 项目中，添加对类库项目的项目引用。
- 在 Core 项目的 `Startup.cs` 中，为 DI 注册上下文。

- 在 Core 项目的 appsettings.json 中，添加连接字符串。
- 在 Core 项目中，添加控制器和视图以验证可读取和写入数据。（请注意，ASP.NET Core MVC 基架不会使用从类库引用的 EF6 上下文。）

总结

本文提供了在 ASP.NET Core 应用程序中使用 Entity Framework 6 的基本指南。

其他资源

- [Entity Framework - 基于代码的配置](#)

ASP.NET Core 中的 Azure 存储

2018/4/10 • 1 min to read • [Edit Online](#)

- [使用 Visual Studio 连接服务添加 Azure 存储](#)
- [开始使用 Blob 存储和 Visual Studio 连接服务](#)
- [开始使用队列存储和 Visual Studio 连接服务](#)
- [开始使用表存储和 Visual Studio 连接服务](#)

ASP.NET Core 中的客户端开发

2018/2/26 • 1 min to read • [Edit Online](#)

- [使用 Gulp](#)
- [使用 Grunt](#)
- [使用 Bower 管理客户端包](#)
- [使用 Bootstrap 构建响应式站点](#)
- [使用 LESS、Sass 和 Font Awesome 为应用设置样式](#)
- [捆绑和缩小](#)
- [TypeScript](#)
- [使用浏览器链接](#)
- [对 SPA 使用 JavaScriptServices](#)
- [使用 SPA 项目模板](#)
 - [Angular 项目模板](#)
 - [React 项目模板](#)
 - [带 Redux 的 React 项目模板](#)

在 ASP.NET 核心中使用 Gulp

2018/5/14 • 11 min to read • [Edit Online](#)

通过[艾力克 Reitan](#), [Scott Addie](#), [Daniel Roth](#), 和[Shayne 贝叶](#)

在典型的现代 web 应用中, 可能会生成过程:

- 捆绑和 minify JavaScript 和 CSS 文件。
- 运行工具以调用之前每个生成的绑定和缩减任务。
- 小于编译或 SASS 文件复制到 CSS。
- 编译 CoffeeScript 或 TypeScript 文件添加到 JavaScript。

A 任务运行程序是一种工具, 可以自动进行这些例程开发任务和的详细信息。Visual Studio 为两个流行的基于 JavaScript 的任务流道提供内置支持: [Gulp](#) 和 [Grunt](#)。

gulp

Gulp 是一个基于 JavaScript 的流式处理生成工具包, 客户端代码。它通常用于在生成环境中触发特定事件时流通过一系列的进程的客户端文件。例如, 使用 Gulp 来自动执行[绑定和缩减](#)或清理新生成前的开发环境。

在中定义一组 Gulp 任务`gulpfile.js`。以下 JavaScript 包括 Gulp 模块, 并指定文件路径, 以在即将推出的任务中引用:

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
    webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";
```

上面的代码中指定的节点模块所需。`require` 函数导入每个模块, 以便依赖任务可以使用其功能。每个导入的模块被分配给变量。模块可以位于按名称或路径。在此示例中, 模块名为 `gulp`, `rimraf`, `gulp-concat`, `gulp-cssmin`, 和 `gulp-uglify` 按名称检索。此外, 创建一系列的路径, 以便可以重复使用并在任务中引用的 CSS 和 JavaScript 文件的位置。下表提供了的模块中包含的描述`gulpfile.js`。

模块名	描述
gulp	Gulp 流式处理生成系统中。有关详细信息, 请参阅 gulp 。
rimraf	节点删除模块。有关详细信息, 请参阅 rimraf 。

模块名	描述
gulp concat	一个连接基于操作系统的换行字符的文件的模块。有关详细信息, 请参阅 gulp concat 。
gulp-cssmin	Minifies CSS 文件模块。有关详细信息, 请参阅 gulp cssmin 。
gulp uglify	Minifies 模块 <i>js</i> 文件。有关详细信息, 请参阅 gulp uglify 。

必备项的模块将导入后, 可以指定任务。此处有六项任务注册, 表示通过以下代码:

```
gulp.task("clean:js", function (cb) {
  rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
  rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
  return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

下表提供了在上面的代码中指定的任务的说明:

任务名称	描述
干净: js	使用 rimraf 节点删除模块删除 site.js 文件的缩减的版本的任务。
干净: css	使用 rimraf 节点删除模块删除 site.css 文件的缩减的版本的任务。
清理	一个任务, 它调用 <code>clean:js</code> 任务后, 跟 <code>clean:css</code> 任务。
min:js	Minifies 和串联的 js 文件夹中的所有.js 文件的任务。。排除 min.js 文件。
min:css	Minifies 和串联的 css 文件夹中的所有.css 文件的任务。。排除 min.css 文件。
min	一个任务, 它调用 <code>min:js</code> 任务后, 跟 <code>min:css</code> 任务。

正在运行的默认任务

如果你尚未创建新的 Web 应用程序, 请在 Visual Studio 中创建新的 ASP.NET Web 应用程序项目。

- 将新的 JavaScript 文件添加到你的项目并将其命名*gulpfile.js*, 然后将以下代码复制。

```
/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
    webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {
    rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
    rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

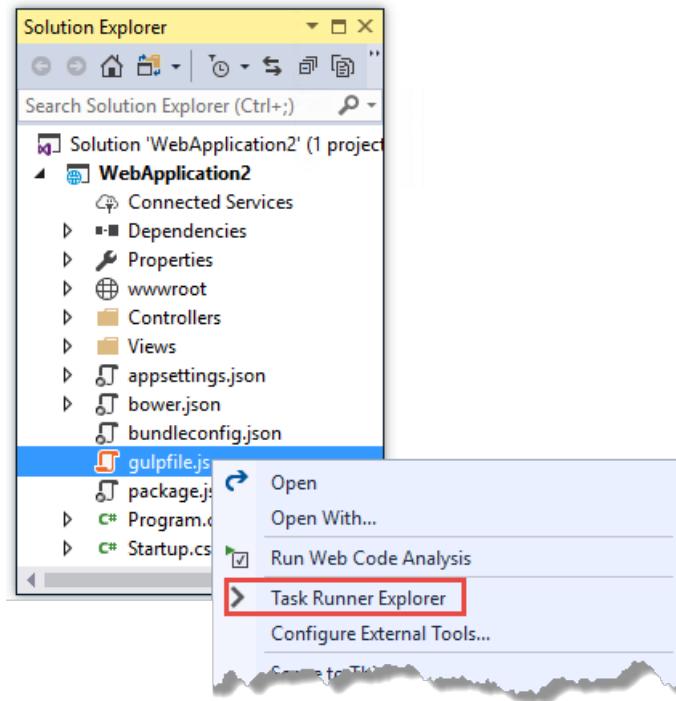
gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

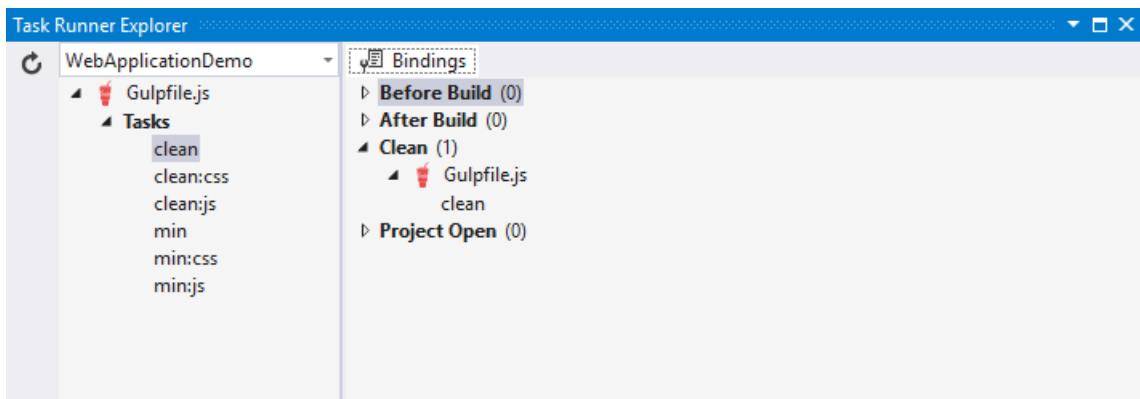
- 打开*package.json*文件 (添加如果不是存在) 并添加以下。

```
{
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "2.0.1",
    "rimraf": "2.6.1"
  }
}
```

3. 在解决方案资源管理器，右键单击*gulpfile.js*，然后选择任务运行程序资源管理器。



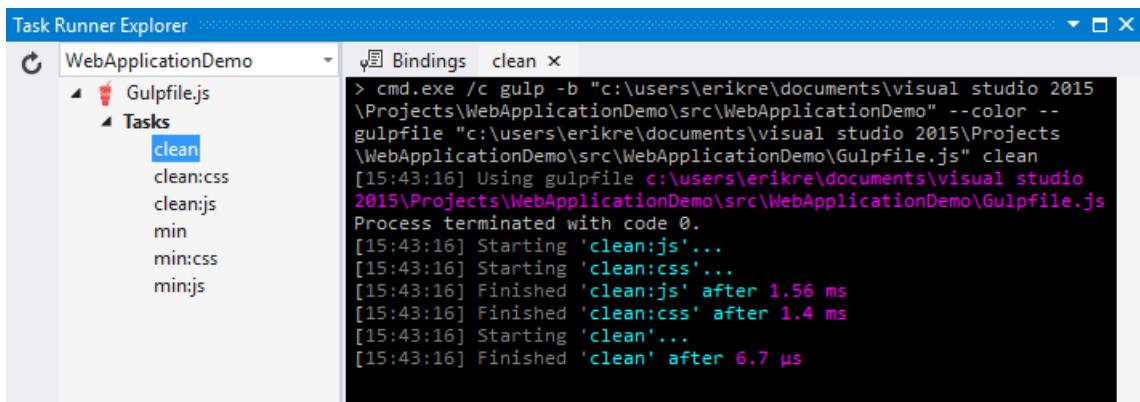
任务运行程序资源管理器显示 Gulp 任务的列表。(你可能需要单击刷新显示项目名称左侧的按钮。)



重要事项

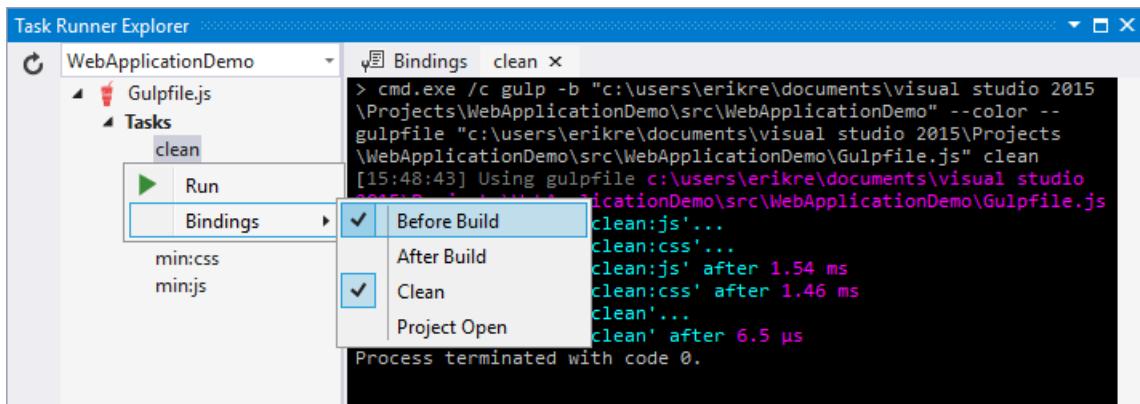
任务运行程序资源管理器，才会显示上下文菜单项*gulpfile.js*是在根项目目录中。

4. 在此之下任务中任务运行程序资源管理器，右键单击干净，然后选择运行从弹出菜单。



任务运行程序资源管理器将创建名为的新选项卡干净和执行清理任务，如中定义*gulpfile.js*。

5. 右键单击干净任务，然后选择绑定 > 之前生成。



之前生成绑定将配置 clean 任务，以在每次生成项目之前自动运行。

绑定您设置任务运行程序资源管理器顶部的注释的形式存储在你 `gulpfile.js` 和仅在 Visual Studio 中有效。不需要 Visual Studio 的替代方法是配置 gulp 中的任务自动执行你 `.csproj` 文件。例如，将这个放你 `.csproj` 文件：

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp clean" />
</Target>
```

现在 Visual Studio 中或从命令提示符处使用运行项目时执行清理任务 `dotnet run` 命令 (运行 `npm install` 第一个)。

定义和运行新任务

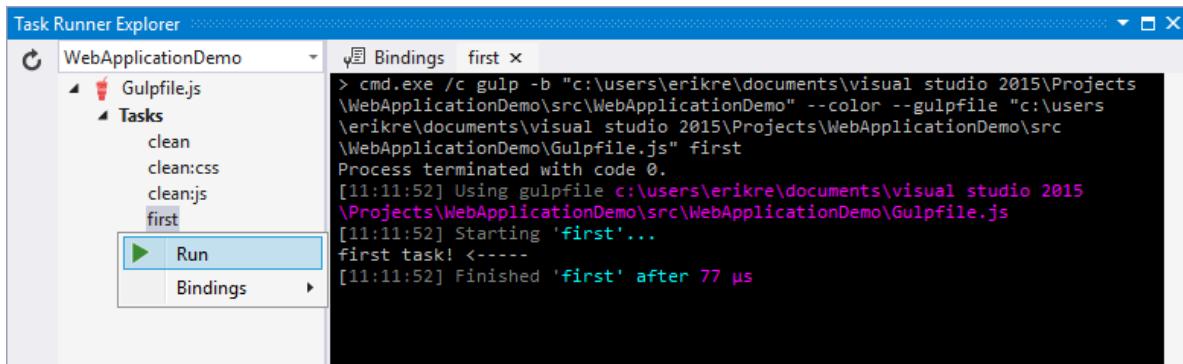
若要定义新的 Gulp 任务，修改 `gulpfile.js`。

1. 末尾添加以下 JavaScript `gulpfile.js`:

```
gulp.task("first", function () {
  console.log('first task! <----');
});
```

此任务名为 `first`，和它只需显示的字符串。

2. 保存 `gulpfile.js`。
3. 在解决方案资源管理器，右键单击 `gulpfile.js`，然后选择 **任务运行程序资源管理器**。
4. 在任务运行程序资源管理器，右键单击 **第一个**，然后选择 **运行**。



输出文本显示。若要查看基于常见方案的示例，请参阅 [Gulp 配方](#)。

定义和一系列中运行任务

当你运行多个任务时，任务将默认情况下同时运行。但是，如果你需要以特定顺序运行任务，你必须指定每个任务过程何时完成，以及为哪些任务依赖于另一个任务完成。

- 若要定义要按顺序运行的任务的一系列，替换 `first` 中前面添加的任务 `gulpfile.js` 替换为以下：

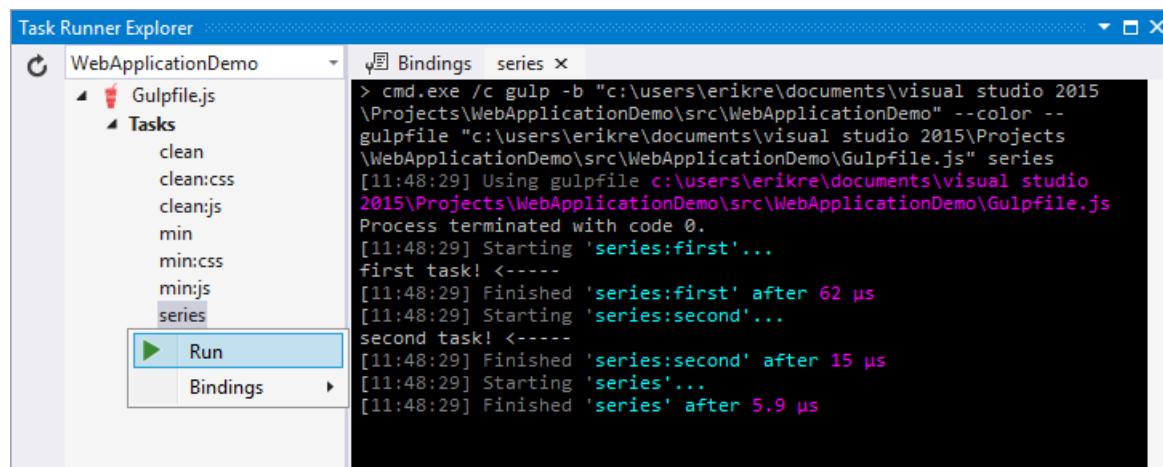
```
gulp.task("series:first", function () {
  console.log('first task! <----');
});

gulp.task("series:second", ["series:first"], function () {
  console.log('second task! <----');
});

gulp.task("series", ["series:first", "series:second"], function () {});
```

你现在具有三个任务：`series:first`，`series:second`，和`series`。`series:second` 任务包括指定任务要运行和完成之前的数组的第二个参数。`series:second` 任务将运行。根据上面，唯一的代码中的指定 `series:first` 任务必须完成之前 `series:second` 任务将运行。

- 保存 `gulpfile.js`。
- 在解决方案资源管理器，右键单击 `gulpfile.js` 和选择 **任务运行程序** 资源管理器如果尚未打开。
- 在任务运行程序资源管理器，右键单击 **系列** 和选择 **运行**。



IntelliSense

IntelliSense 提供了代码完成、参数说明和其他功能来提高工作效率，并减少错误。用 JavaScript 编写 gulp 任务因此，IntelliSense 可以提供开发时的协助。当你使用 JavaScript，IntelliSense 也会列出对象、函数、属性和可用的参数基于当前的上下文。从 intellisense 来完成代码提供的弹出列表中选择编码选项。

The screenshot shows the Visual Studio code editor with the file 'gulpfile.js' open. A tooltip is displayed over the 'task' method call, providing documentation for it. The tooltip includes the signature 'task(String name, [Array deps], [Function fn])', a description 'Registers a Gulp task.', and a note about the function's purpose: 'fn: The function that performs the task's operations. Generally this takes the form of gulp.src().pipe(someplugin()).' The code itself contains several Gulp tasks, including 'min:css', 'min', 'series:first', 'series:second', and 'series'.

```
gulp.task("min:css", function () {
  gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest(".")));
});

gulp.task("min", ["min:js", "min:css"]);

gulp.task("series:first", function () {
  console.log('first task! <-----');
});

gulp.task("series:second", ["series:first"], function () {
  console.log('second task! <-----');
});

gulp.task("series", ["series:first", "series:second"], function () {});
```

Intellisense 的详细信息, 请参阅[JavaScript IntelliSense](#)。

开发、过渡和生产环境

当 Gulp 用于优化过渡和生产客户端文件时, 处理的文件将保存到本地的过渡和生产位置。`_Layout.cshtml`文件使用环境标记帮助器提供两个不同版本的 CSS 文件。CSS 文件的一个版本是用于开发和另一个版本进行了优化的过渡和生产。在 Visual Studio 2017, 当你更改**ASPNETCORE_ENVIRONMENT**环境变量 `Production`, Visual Studio 将生成 Web 应用程序和链接到的最小化 CSS 文件。下面的标记演示环境标记帮助程序包含将标记与 `Development` CSS 文件和缩减 `Staging, Production` CSS 文件。

```
<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
  <script src("~/js/site.js" asp-append-version="true")></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery"
    crossorigin="anonymous"
    integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfhIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
    crossorigin="anonymous"
    integrity="sha384-Tc5IQib027qvyjSMFHjOMaLkfuWVxZxUPnCJA712mCWNIPG9mGCD8wGNICPD7Txa">
  </script>
  <script src "~/js/site.min.js" asp-append-version="true"></script>
</environment>
```

环境之间切换

若要针对不同的环境编译之间切换, 请修改**ASPNETCORE_ENVIRONMENT**环境变量的值。

1. 在任务运行程序资源管理器, 验证 `min` 任务已设置为运行之前生成。
2. 在解决方案资源管理器, 右键单击项目名称并选择属性。

在显示属性表为 Web 应用。

3. 单击“调试”选项卡。
4. 设置的值宿主：环境环境变量 `Production`。
5. 按 **F5** 在浏览器中运行该应用程序。
6. 在浏览器窗口中，右击该页并选择查看源若要查看的 HTML 页。

请注意，样式表链接指向的缩减的 CSS 文件。

7. 关闭浏览器来停止 Web 应用。
8. 在 Visual Studio 中，返回到 Web 应用的属性表，并更改宿主：环境环境变量回 `Development`。
9. 按 **F5** 再次在浏览器中运行该应用程序。
10. 在浏览器窗口中，右击该页并选择查看源若要查看的 HTML 页。

请注意，样式表链接指向 CSS 文件的 unminified 版本。

有关 ASP.NET 核心中的环境的详细信息，请参阅[使用多个环境](#)。

任务和模块的详细信息

Gulp 任务已注册到的函数名称。如果其他任务都必须运行在当前任务之前，你可以指定依赖关系。其他函数，您可以运行和监视 Gulp 任务，以及将源设置 (*src*) 和目标 (*dest*) 正在修改的文件。以下是主 Gulp API 函数：

GULP 函数	语法	描述
任务	<code>gulp.task(name[, deps], fn) { }</code>	<code>task</code> 函数创建的任务。 <code>name</code> 参数定义任务的名称。 <code>deps</code> 参数包含要运行此任务之前完成的任务的数组。 <code>fn</code> 参数表示执行任务的操作的回调函数。
监视	<code>gulp.watch(glob [, opts], tasks)</code> <code>{ }</code>	<code>watch</code> 发生文件更改时，函数会监视文件和运行任务。 <code>glob</code> 参数是 <code>string</code> 或 <code>array</code> ，它确定要监视哪些文件。 <code>opts</code> 参数提供了监视选项的其他文件。
src	<code>gulp.src(globs[, options]) { }</code>	<code>src</code> 函数提供了 <code>glob</code> 值匹配的文件。 <code>glob</code> 参数是 <code>string</code> 或 <code>array</code> ，它确定哪些文件读取。 <code>options</code> 参数提供附加文件选项。
目标	<code>gulp.dest(path[, options]) { }</code>	<code>dest</code> 函数定义可以向其写入文件的位置。 <code>path</code> 参数是字符串或确定目标文件夹的函数。 <code>options</code> 参数是一个对象，指定输出文件夹的选项。

有关其他的 Gulp API 参考信息，请参阅[Gulp 文档 API](#)。

Gulp 配方

Gulp 社区提供 Gulp 配方。这些配方包含 Gulp 任务用于针对常见情况。

其他资源

- [Gulp 文档](#)
- [绑定和缩减中 ASP.NET 核心](#)
- [在 ASP.NET 核心中使用 Grunt](#)

在 ASP.NET 核心中使用 Grunt

2018/4/10 • 10 min to read • [Edit Online](#)

通过了米

Grunt 是自动化脚本缩减、TypeScript 编译、代码质量"链接形式"工具、CSS 预处理器和几乎任何重复繁琐的工作，需要采取措施来支持客户端开发的 JavaScript 任务运行程序。尽管 ASP.NET 项目模板默认情况下使用 Gulp 在 Visual Studio 中，完全支持 grunt (请参阅[使用 Gulp](#))。

此示例使用一个空的 ASP.NET Core 项目作为起始点，以显示如何从零开始的客户端生成过程中自动运行。

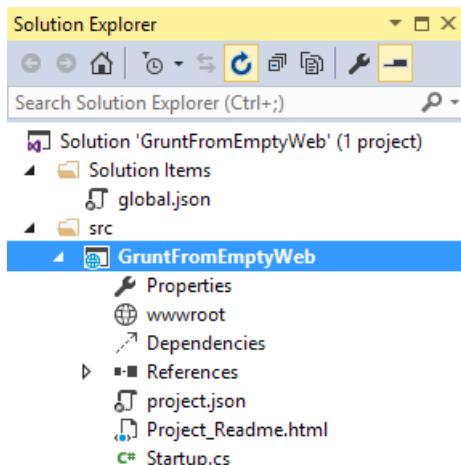
完成的示例清除目标部署目录、将合并 JavaScript 文件，检查代码质量，会将 JavaScript 文件内容和将部署到 web 应用程序的根目录。我们将使用下列包：

- **grunt**: Grunt 任务运行程序包。
- **grunt contrib 清理**: 删除文件或目录的插件。
- **grunt contrib jshint**: 查看 JavaScript 代码质量的插件。
- **grunt contrib concat**: 将文件联接成单个文件的插件。
- **grunt contrib uglify**: minifies JavaScript 以减少大小的插件。
- **grunt contrib 监视**: 监视文件活动的插件。

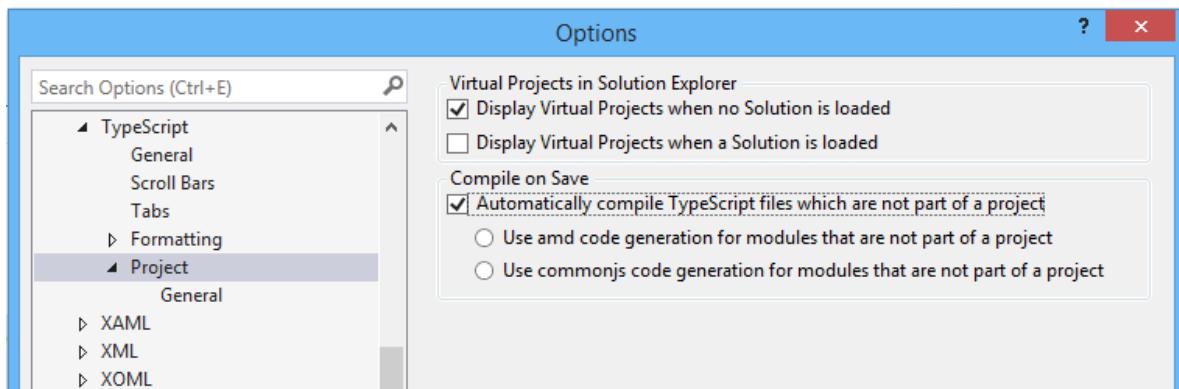
准备好应用程序

若要开始，设置新的空 web 应用程序，并添加 TypeScript 示例文件。TypeScript 文件将会自动编译到 JavaScript 使用默认 Visual Studio 设置，并且将是我们原材料处理使用 Grunt。

1. 在 Visual Studio 中，创建一个新 **ASP.NET Web Application**。
2. 在**新建 ASP.NET 项目**对话框中，选择 **ASP.NET Core 空模板**，然后单击确定按钮。
3. 在解决方案资源管理器，查看的项目结构。`\src` 文件夹包含空 `wwwroot` 和 `Dependencies` 节点。



4. 添加一个名为的新文件夹 `TypeScript` 到你的项目目录。
5. 在添加之前的任何文件，请确保 Visual Studio 具有选项编译保存 TypeScript 文件检查。导航到工具 > 选项 > 文本编辑器 > **TypeScript** > 项目：



6. 右键单击 **TypeScript** 目录，然后选择 **添加 > 新项** 从上下文菜单。选择 **JavaScript 文件** 项并将该文件命名为 *Tastes.ts* (请注意*.ts 扩展)。将以下 TypeScript 代码的行复制到文件 (在您保存，新 *Tastes.js* 文件将出现与 JavaScript 源)。

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. 添加的第二个文件，以便 **TypeScript** 目录并将其命名 **Food.ts**。将下面的代码复制到文件。

```
class Food {
    constructor(name: string, calories: number) {
        this._name = name;
        this._calories = calories;
    }

    private _name: string;
    get Name() {
        return this._name;
    }

    private _calories: number;
    get Calories() {
        return this._calories;
    }

    private _taste: Tastes;
    get Taste(): Tastes { return this._taste }
    set Taste(value: Tastes) {
        this._taste = value;
    }
}
```

配置 NPM

接下来，配置 NPM 来下载 grunt 和 grunt 任务。

1. 在解决方案资源管理器，右键单击该项目并选择 **添加 > 新项** 从上下文菜单。选择 **NPM 配置文件** 项，保留默认名称，*package.json*，然后单击 **添加** 按钮。
2. 在 *package.json* 文件内，**devDependencies** 对象大括号中，输入 "grunt"。选择 **grunt** 从智能感知列表并按 Enter 键。Visual Studio 将 quote grunt 包名称，并添加一个冒号。从 Intellisense 列表的顶部中的冒号右侧，选择包的最新稳定版本 (按 **ctrl-Space** 如果 Intellisense 不显示)。

```
"devDependencies": {
    "grunt": "0.4.5"
}
```

注意

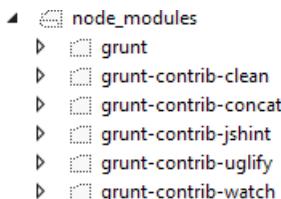
使用 NPM 语义版本控制来组织依赖关系。语义版本控制，也称为 SemVer，标识包的编号方案。.Intellisense 显示仅几个常用的选项，从而简化了语义版本控制。Intellisense 列表（在上面的示例 0.4.5）中的顶级项被视为包的最新稳定版本。脱字号 (^) 符号匹配的最新的主版本和波形符 () 与最新次要版本匹配。请参阅 [NPM semver 版本分析器参考](#) 作为 SemVer 提供其完整表达的指南。

- 添加更多的依赖关系，以加载 grunt-contrib-* 打包以干净，*jshint*，*concat*，*uglify*，和监视下面的示例中所示。版本不需要与示例匹配。

```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-clean": "0.6.0",
  "grunt-contrib-jshint": "0.11.0",
  "grunt-contrib-concat": "0.5.1",
  "grunt-contrib-uglify": "0.8.0",
  "grunt-contrib-watch": "0.6.1"
}
```

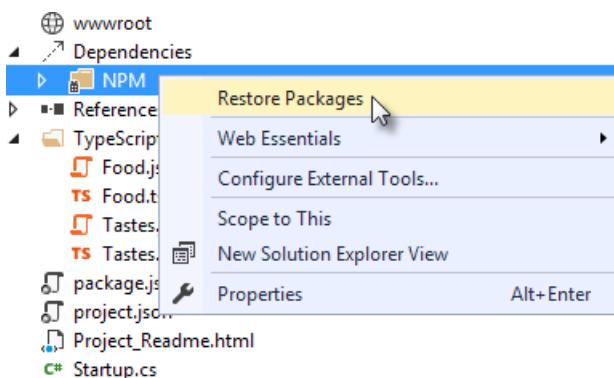
- 保存 *package.json* 文件。

将下载的每个 *devDependencies* 项包，以及每个包需要的任何文件。你可以查找中的包文件 `node_modules` 通过启用目录显示所有文件在解决方案资源管理器的按钮。



注意

如果需要可以通过右键单击来手动还原解决方案资源管理器中的依赖关系 `Dependencies\NPM` 并选择还原包菜单选项。



配置 Grunt

使用名为的清单配置 *Gruntfile.js* 的定义、加载和注册任务可以手动运行或配置以自动基于运行 Visual Studio 中的事件。

- 右键单击项目并选择 **添加 > 新项**。选择 **Grunt 配置文件** 选项，请保留默认名称，*Gruntfile.js*，然后单击 **添加** 按钮。

初始代码包含了一个模块定义和 `grunt.initConfig()` 方法。`initConfig()` 用于为每个包中，设置选项和模块的剩余部分将会加载和注册任务。

```
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

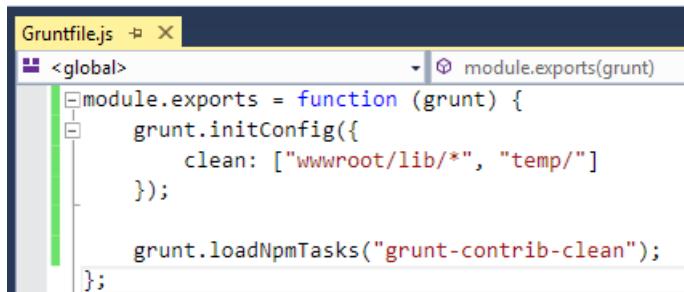
2. 内部 `initConfig()` 方法，添加用于 `clean` 任务示例中所示 `Gruntfile.js` 下面。Clean 任务，接受目录字符串的数组。此任务从 `wwwroot/lib` 中删除文件，并将移除整个临时目录。

```
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
  });
};
```

3. 下面 `initConfig()` 方法中，添加对的调用 `grunt.loadNpmTasks()`。这将从 Visual Studio 来使该任务可运行。

```
grunt.loadNpmTasks("grunt-contrib-clean");
```

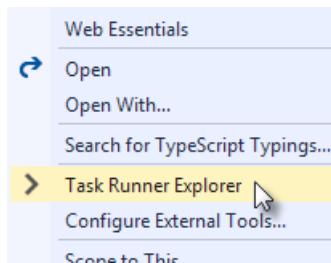
4. 保存 `Gruntfile.js`。该文件应类似于下面的屏幕截图。



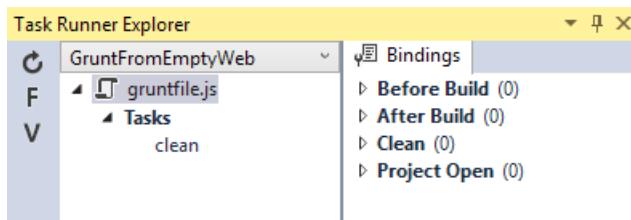
```
Gruntfile.js ✘ X
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"]
  });

    grunt.loadNpmTasks("grunt-contrib-clean");
};
```

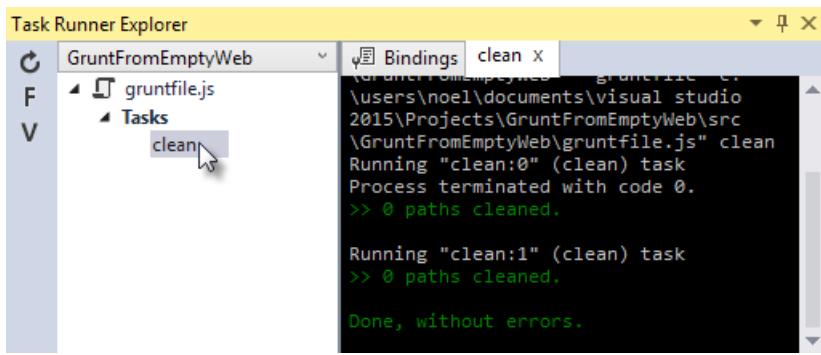
5. 右键单击 `Gruntfile.js` 和选择 **任务运行程序资源管理器** 从上下文菜单。任务运行程序资源管理器窗口将打开。



6. 验证 `clean` 下显示任务在任务运行程序资源管理器。



7. 右键单击 `clean` 任务，然后选择 **运行** 从上下文菜单。命令窗口将显示任务的进度。



注意

没有任何文件或目录尚未清理。如果您愿意，你可以在解决方案资源管理器中手动创建它们，然后运行为测试的 clean 任务。

- 在 initConfig() 方法中，添加一个条目 concat 使用下面的代码。

`src` 属性数组列出文件合并，它们应合并的顺序。`dest` 属性将路径分配给组合生成的文件。

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

注意

`all` 在上面的代码的属性为目标的名称。某些 Grunt 任务中使用的目标用于允许多个生成环境。你可以查看内置的目标使用 Intellisense，或指定你自己。

- 添加 `jshint` 任务使用下面的代码。

Jshint 代码质量实用工具运行的临时目录中找到的每个 JavaScript 文件。

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

注意

选项 "-W069" 一个错误时产生的 jshint JavaScript 使用方括号语法，即分配而不是点表示法，属性 `Tastes["Sweet"]` 而不是 `Tastes.Sweet`。选项关闭了警告，以允许进程继续的其余部分。

- 添加 `uglify` 任务使用下面的代码。

任务 minifies `combined.js` 文件的临时目录中找到并在 `wwwroot/lib` 以下标准命名约定中创建结果文件* <文件名>。`min.js*`。

```

uglify: {
  all: {
    src: ['temp/combined.js'],
    dest: 'wwwroot/lib/combined.min.js'
  }
},

```

11. 在加载 grunt contrib 清理调用 `grunt.loadNpmTasks()`, 包括相同的调用, 用于 jshint, concat 并 uglify 使用下面的代码使用。

```

grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');

```

12. 保存`Gruntfile.js`。该文件应类似于下面的示例。



```

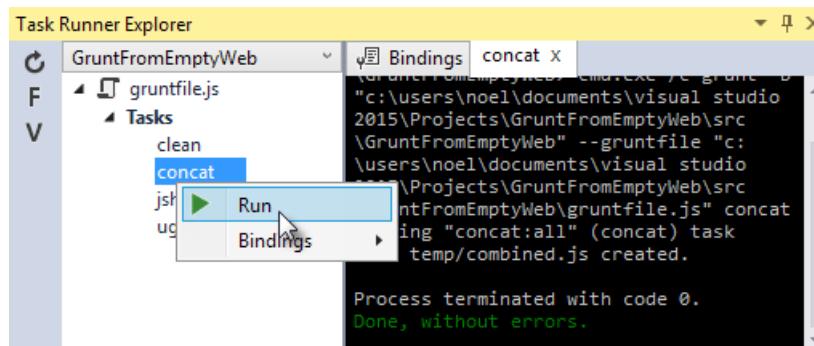
Gruntfile.js  X
{} module      exports(grunt)
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/*"],
    concat: {
      all: {
        src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
        dest: 'temp/combined.js'
      }
    },
    jshint: { files: ['temp/*.js'], options: { '-W069': false } },
    uglify: {
      all: {
        src: ['temp/combined.js'],
        dest: 'wwwroot/lib/combined.min.js'
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-clean');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-uglify');
};

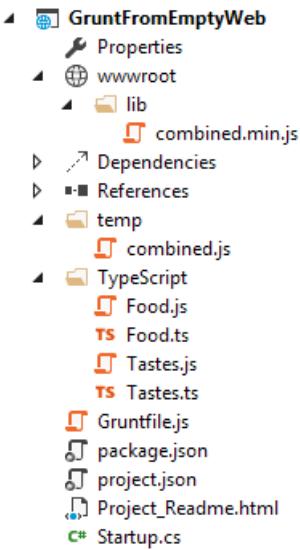
110 %

```

13. 请注意, 该任务运行程序资源管理器任务列表包括 `clean` , `concat` , `jshint` 和 `uglify` 任务。按顺序运行每个任务, 并观察解决方案资源管理器中的结果。每个任务应运行且未发生错误。



Concat 任务创建一个新`combined.js`文件并将其放到临时目录。Jshint 任务只需运行, 但不生成输出。Uglify 任务创建一个新`combined.min.js`文件并将其放到 `wwwroot/lib`。完成时, 解决方案应类似下面的屏幕截图:



注意

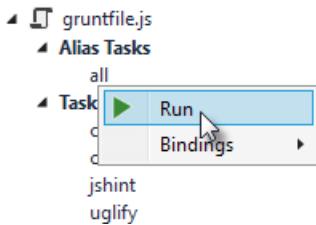
每个包的选项的详细信息, 请访问 <https://www.npmjs.com/> 和查找在主页上的搜索框中的包名称。例如, 你可以查找 `grunt contrib` 清理包以获取说明它的所有参数的文档链接。

总的来说

使用 Grunt `registerTask()` 方法以按特定顺序运行一系列任务。例如, 若要运行示例上述顺序干净的步骤-> `concat->jshint->uglify`, 将下面的代码添加到该模块。应将代码添加到与外部 `initConfig loadNpmTasks()` 调用相同的级别。

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

新任务将显示在任务运行程序资源管理器的别名任务中。您可以右键单击, 并运行它, 就像其他任务。`all` 任务将运行 `clean`, `concat`, `jshint` 和 `uglify`, 按顺序。



监视更改

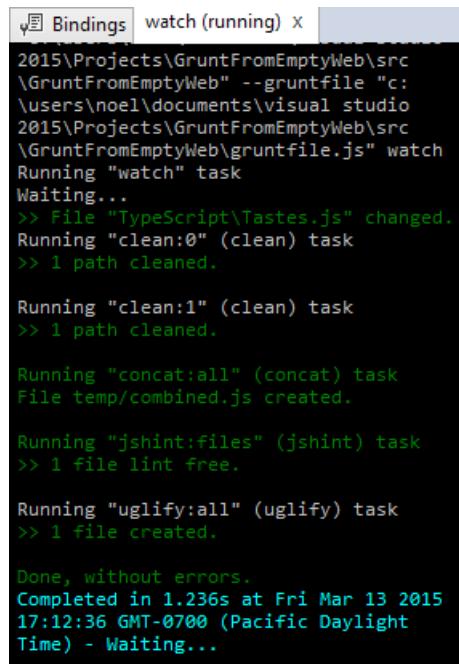
A `watch` 任务保留文件和目录。如果它检测到更改, 监视将自动触发任务。将下面的代码添加到 `initConfig`, 若要监视的更改*TypeScript 目录中的.js 文件。如果更改 JavaScript 文件, `watch` 将运行 `all` 任务。

```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

添加对的调用 `loadNpmTasks()` 以显示 `watch` 任务运行程序资源管理器中的任务。

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

右键单击任务运行程序资源管理器中的监视任务，并从上下文菜单中选择运行。显示运行的监视任务的命令窗口将显示“等待...”消息。打开一个 TypeScript 文件，添加一个空格，然后保存该文件。这将触发监视任务并触发其他任务以按顺序运行。下面的屏幕截图显示了示例运行。



```
Bindings watch (running) x
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" watch
Running "watch" task
Waiting...
>> File "TypeScript\Tastes.js" changed.
Running "clean:0" (clean) task
>> 1 path cleaned.

Running "clean:1" (clean) task
>> 1 path cleaned.

Running "concat:all" (concat) task
File temp/combined.js created.

Running "jshint:files" (jshint) task
>> 1 file lint free.

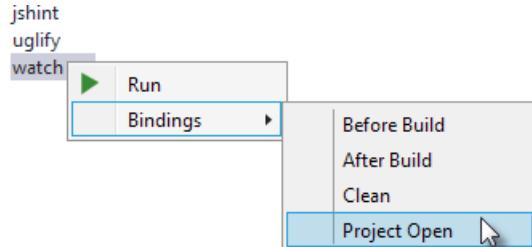
Running "uglify:all" (uglify) task
>> 1 file created.

Done, without errors.
Completed in 1.236s at Fri Mar 13 2015
17:12:36 GMT-0700 (Pacific Daylight
Time) - Waiting...
```

绑定到 Visual Studio 事件

除非你想要手动启动你的任务，每次在 Visual Studio 中工作时，你可以将绑定到的任务之前生成，后生成，清理，和项目打开事件。

接下来将绑定 `watch` 以便使其运行每次 Visual Studio 将打开。在任务运行程序资源管理器，右键单击监视任务并选择绑定 > 项目打开从上下文菜单。



卸载并重新加载项目。在项目加载后再次，监视任务将开始自动运行。

总结

Grunt 是可以用于自动执行大多数客户端生成任务的功能强大的任务运行程序。Grunt 利用 NPM 来提供其包和工具与 Visual Studio 的集成的功能。Visual Studio 的任务运行程序资源管理器检测到配置文件的更改，并提供方便的界面以运行任务，查看正在运行的任务，并将任务绑定到 Visual Studio 事件。

其他资源

- [使用 Gulp](#)

管理 ASP.NET Core 中的 Bower 的客户端包

2018/5/14 • 6 min to read • [Edit Online](#)

通过[Rick Anderson, 了米](#), 和[Scott Addie](#)

重要事项

而维持 Bower, 其 maintainer 建议使用另一种解决方案。[库管理器](#)(简称 LibMan)是 Visual Studio 的新客户端的静态内容管理系统。与 Webpack yarn 是一个常用的替代项为其[迁移说明](#)可用。

Bower 调用自身"web 程序包管理器"。内部.NET 生态系统, 它将填入 void 留下的 NuGet 的无法传送静态内容的文件。对于 ASP.NET Core 项目, 这些静态文件, 则所固有的客户端库, 如[jQuery](#)和[Bootstrap](#)。对于.NET 库, 你仍然使用[NuGet](#)程序包管理器。

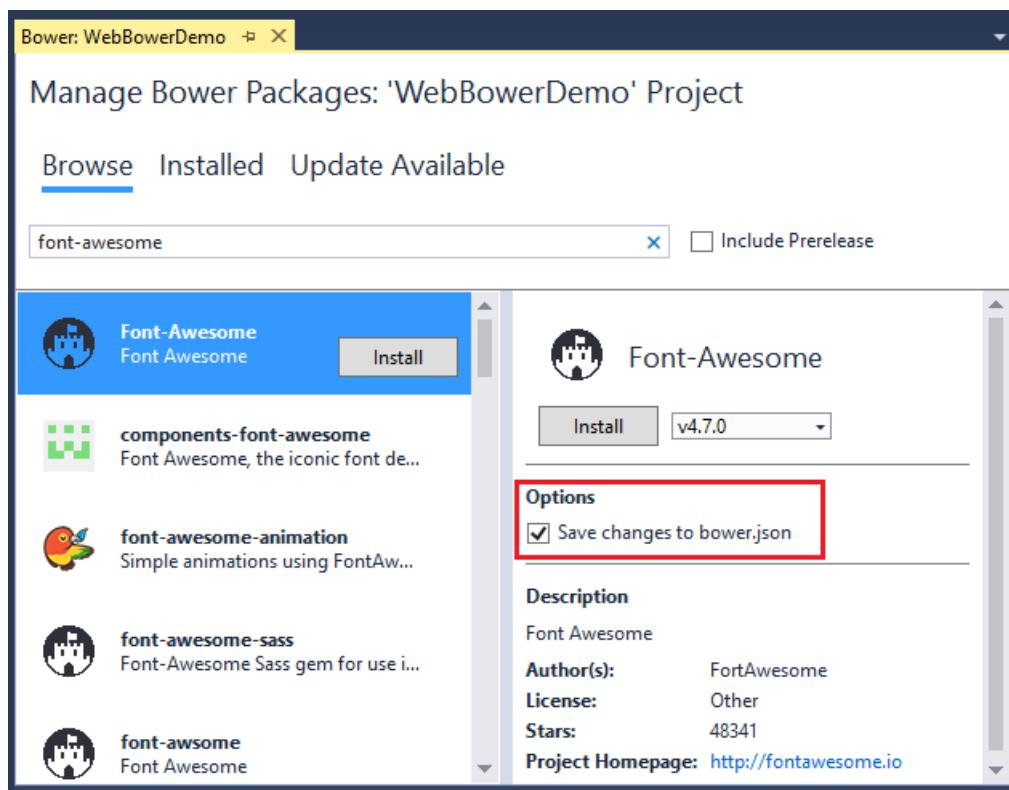
设置客户端的 ASP.NET Core 项目模板创建的新项目生成过程。[jQuery](#)和[Bootstrap](#)安装, 并支持 Bower。

在列出客户端包**bower.json**文件。ASP.NET 核心项目模板配置**bower.json** jQuery、jQuery 验证与 Bootstrap。

在本教程中, 我们将添加对支持[字体出色](#)。可以使用安装 bower 包管理 **Bower** 包 UI 或手动在**bower.json**文件。

通过管理 Bower 包 UI 的安装

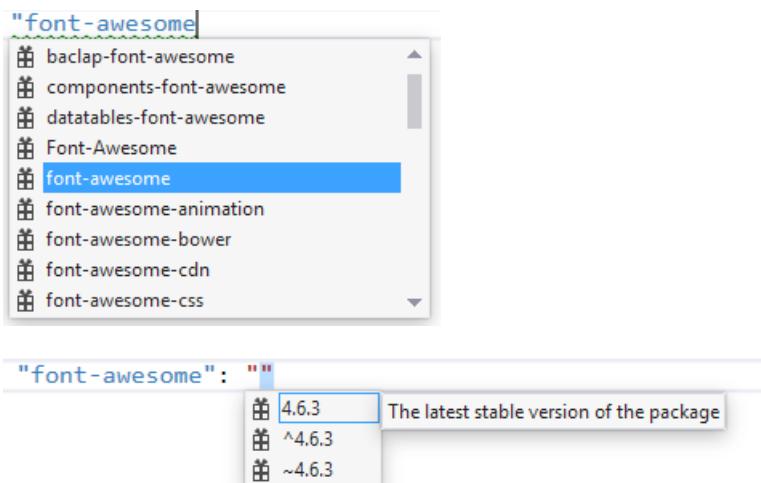
- 创建新的 ASP.NET 核心 Web 应用程序与[ASP.NET 核心 Web 应用程序 \(.NET Core\)](#) 模板。选择[Web 应用程序](#)和[无身份验证](#)。
- 右键单击解决方案资源管理器中的项目并选择[管理 Bower 包](#)(或者从主菜单中, 项目 > 管理 Bower 包)。
- 在**Bower:<项目名称>**窗口中, 单击"浏览"选项卡, 并输入, 然后筛选包列表 `font-awesome` 的搜索框中:



- 确认"保存更改为**bower.json**"复选框已选中。从下拉列表中选择一个版本, 然后单击安装按钮。输出窗口显示的安装详细信息。

手动安装在 bower.json

打开**bower.json**文件并添加到的依赖项的"字体出色"。IntelliSense 会显示可用的包。某个包被选中，将显示可用的版本。下面的映像不较旧，并不会匹配你看到的内容。



Bower 使用语义版本控制来组织依赖关系。语义版本控制，也称为 SemVer，标识包的编号方案<主要>。<次要>。<修补程序>。IntelliSense 显示仅几个常用的选项，从而简化了语义版本控制。IntelliSense 列表（在上面的示例 4.6.3）中的顶级项被视为包的最新稳定版本。脱字号（^）符号匹配的最新的主版本和波形符（~）与最新次要版本匹配。

保存**bower.json**文件。Visual Studio 监视**bower.json**文件的更改。保存后，**bower** 安装执行命令。请参见输出窗口**Bower/npm**视图执行的确切命令。

打开**.bowerrc**文件下**bower.json**。**directory** 属性设置为**wwwroot/lib**指示的位置 Bower 将安装包资产。

```
{  
  "directory": "wwwroot/lib"  
}
```

在解决方案资源管理器搜索框中可用于查找并显示字体出色的包。

打开**views/shared/_Layout.cshtml**文件并将字体出色的 CSS 文件添加到环境标记帮助器为 **Development**。从解决方案资源管理器，将拖字体 **awesome.css** 内 **<environment names="Development">** 元素。

```
<environment names="Development">  
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />  
  <link rel="stylesheet" href="~/css/site.css" />  
  <link href="~/lib/font-awesome/css/font-awesome.css" rel="stylesheet" />  
</environment>
```

在生产应用程序会添加字体 **awesome.min.css** 到环境标记帮助器 **Staging,Production**。

内容替换 **Views\Home\About.cshtml** Razor 文件替换为以下标记：

```
@{  
  ViewData["Title"] = "About";  
}  
  
<div class="list-group">  
  <a class="list-group-item" href="#"></i>&nbsp; Home</a>  
  <a class="list-group-item" href="#"></i>&nbsp; Library</a>  
    <a class="list-group-item" href="#"></i>&nbsp;  
  Applications</a>  
    <a class="list-group-item" href="#"></i>&nbsp; Settings</a>  
</div>
```

运行应用程序并导航到关于视图，以验证字体出色包正常运行。

浏览客户端生成过程

大多数 ASP.NET Core 项目模板已配置为使用 Bower。此下一个演练中创建空的 ASP.NET Core 项目启动，并手动添加每个部分，以便您可以如何在项目中使用 Bower 获得感觉。你可以查看到的项目结构和运行时，输出会和每个配置更改会发生什么情况。

将客户端生成过程用于 Bower 的常规步骤如下：

- 定义项目中使用的包。
- 从 web 页面的引用包。

定义包

一旦列表中的包**bower.json**文件，Visual Studio 将下载它们。下面的示例使用 Bower 加载 jQuery 和引导定向到**wwwroot**文件夹。

- 创建新的 ASP.NET 核心 Web 应用程序与**ASP.NET 核心 Web 应用程序 (.NET Core)** 模板。选择**空**项目模板，然后单击**确定**。
- 在解决方案资源管理器，右键单击项目 >**添加新项**和选择**Bower 配置文件**。注意：A **.bowerrc**还添加文件。
- 打开**bower.json**，并添加 jquery 和引导到 **dependencies** 部分。生成**bower.json**文件将如下所示下面的示例。版本将会发生更改，并且可能不匹配下图所示。

```
{  
  "name": "asp.net",  
  "private": true,  
  "dependencies": {  
    "jquery": "3.1.1",  
    "bootstrap": "3.3.7"  
  }  
}
```

- 保存**bower.json**文件。

验证该项目包括**bootstrap**和**jQuery**中的目录**wwwroot/lib**。Bower 使用 **.bowerrc**文件以安装中的资产**wwwroot/lib**。

注意：“管理 Bower 包”UI 提供手动文件编辑的替代方法。

启用静态文件

- 添加 **Microsoft.AspNetCore.StaticFiles** 到项目的 NuGet 包。
- 启用静态文件提供与 **静态文件中间件**。添加对的调用 **UseStaticFiles** 到 **Configure** 方法 **Startup**。

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

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    }
}
```

引用包

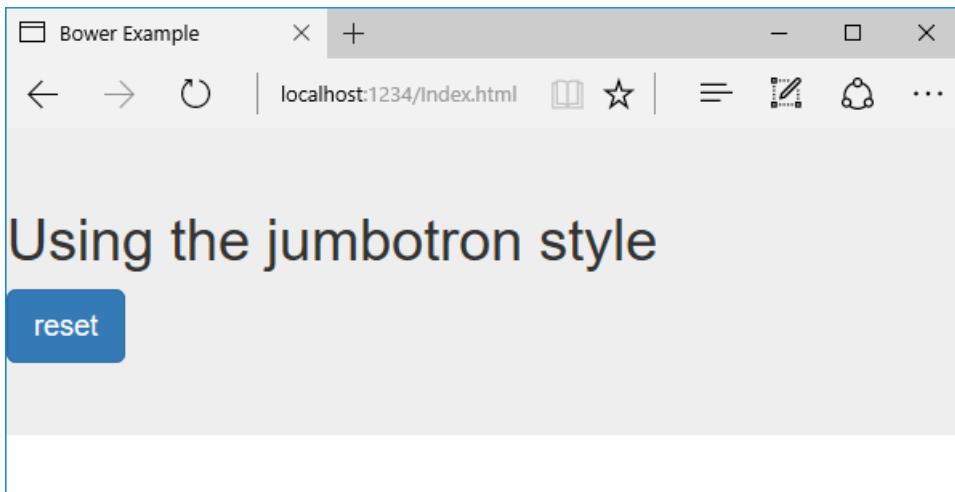
在本部分中，你将创建 HTML 页以验证它可以访问已部署的包。

- 添加一个名为的新 HTML 页*Index.html*到*wwwroot*文件夹。注意：你必须将添加到的 HTML 文件*wwwroot*文件夹。默认情况下，静态内容无法提供外部*wwwroot*。请参阅[静态文件](#)有关详细信息。

内容替换*Index.html*替换为以下标记：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Bower Example</title>
    <link href="lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
    <div class="jumbotron">
        <h1>Using the jumbotron style</h1>
        <p>
            <a class="btn btn-primary btn-lg" role="button">Stateful button</a>
        </p>
    </div>
    <script src="lib/jquery/dist/jquery.js"></script>
    <script src="lib/bootstrap/dist/js/bootstrap.js"></script>
    <script>
        $(".btn").click(function () {
            $(this).text('loading')
                .delay(1000)
                .queue(function () {
                    $(this).text('reset');
                    $(this).dequeue();
                });
        });
    </script>
</body>
</html>
```

- 运行应用程序并导航到 `http://localhost:<port>/Index.html`。或者，使用*Index.html*打开，按 `Ctrl+Shift+W`。验证应用 jumbotron 样式，jQuery 代码响应时单击该按钮，以及启动按钮更改状态。



生成带有 Bootstrap 和 ASP.NET Core 美观、响应迅速网站

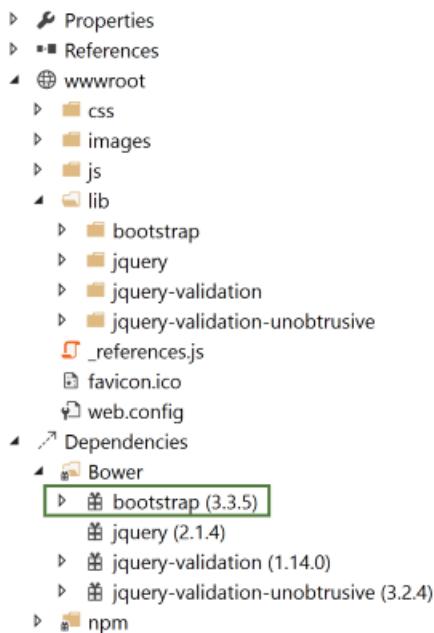
2018/4/27 • 13 min to read • [Edit Online](#)

作者: Steve Smith

Bootstrap 目前最常用的 web 框架开发响应式 web 应用程序。无论您是在前端设计和开发或方面的专家新手，它提供大量的功能和优势，这样可以提高您的网站，用户的体验。Bootstrap 部署为一组 CSS 和 JavaScript 文件，并旨在从手机有效地帮助你的网站或应用程序缩放，平板电脑和桌面。

入门

有多种，若要开始使用 Bootstrap。如果你在 Visual Studio 中开始新的 web 应用程序，则可以为 ASP.NET Core，区分大小的 Bootstrap 会预安装选择默认初学者模板：



添加 ASP.NET Core Bootstrap 项目是只需将其添加到 `bower.json` 作为依赖项：

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

这是 Bootstrap 添加到 ASP.NET 核心项目的方法。

你还可以安装使用多个包管理器，例如 Bower、npm 或 NuGet 之一的 bootstrap。在每个情况下，该过程是实质上是相同的：

Bower

```
bower install bootstrap
```

npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

注意

安装客户端依赖关系，如 ASP.NET Core 中的 Bootstrap 为通过 Bower 的建议的方法（使用 `bower.json`，如上所示）。使用 npm/NuGet 显示来演示如何轻松地 Bootstrap 可以添加到其他类型的 web 应用程序，包括 ASP.NET 的早期版本。

如果您要引用的 Bootstrap 您自己的本地版本，你将需要在将使用它的所有页中引用它们。在生产中应引用使用 CDN 的 bootstrap。在默认 ASP.NET 站点模板中，`_Layout.cshtml` 文件因此像这样：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ ViewData["Title"] - WebApplication1</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
              value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-
brand">WebApplication1</a>
            </div>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
            </ul>
            @await Html.PartialAsync("_LoginPartial")
        </div>
    </div>
</body>
```

```

</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2016 - WebApplication1</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src("~/js/site.js" asp-append-version="true")></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
    </script>
    <script src "~/js/site.min.js" asp-append-version="true"></script>
</environment>

@RenderSection("scripts", required: false)
</body>
</html>

```

注意

如果你要使用的任何 Bootstrap 的 jQuery 插件, 你将需要引用 jQuery。

基本的模板和功能

最基本启动模板看起来非常相似 `_Layout.cshtml` 所示的文件更高版本, 并只包括基本菜单用于导航和呈现页面的其余部分的一个位置。

基本导航

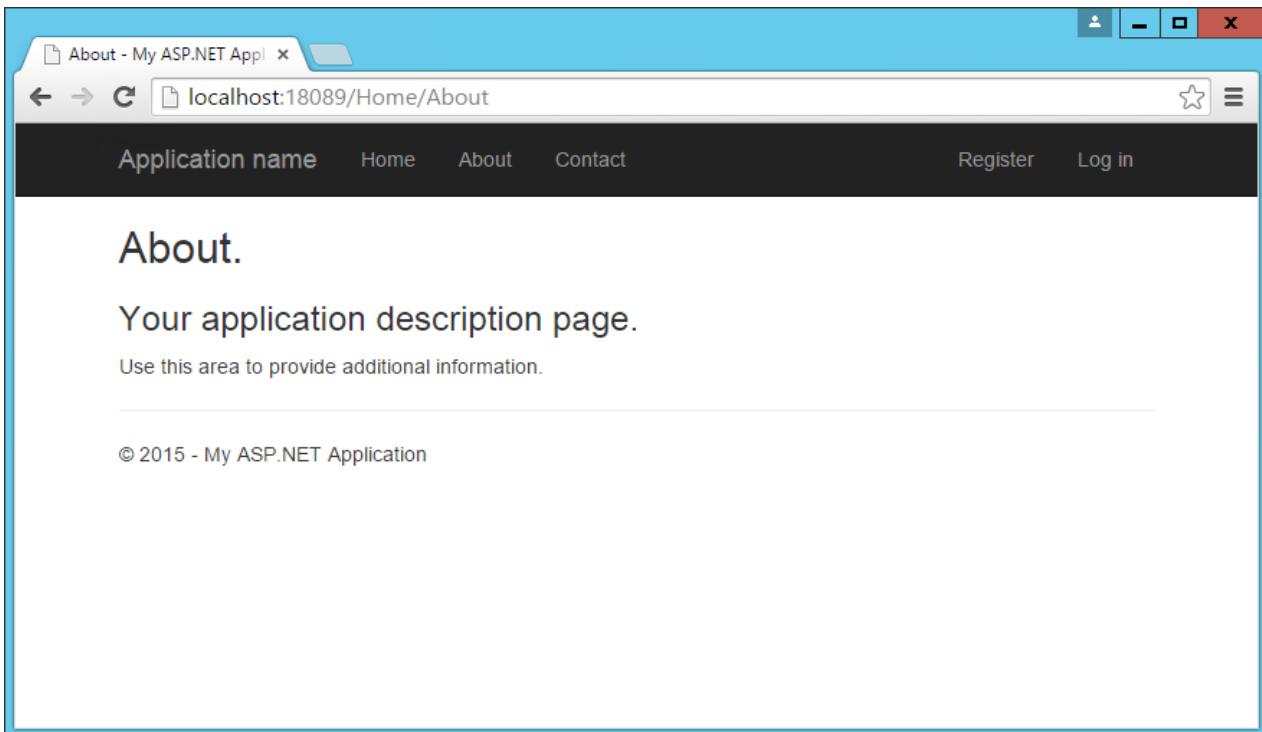
默认模板使用的一组 `<div>` 元素来呈现顶部导航栏和页的正文。如果你使用 HTML5, 则可以将第一个 `<div>` 使用标记 `<nav>` 标记来获取相同的效果, 但具有更精确的语义。此第一部分中 `<div>` 可以看到有多个其他人。首先, `<div>` 与类的"容器", 然后在中, 两个多 `<div>` 元素:"导航条标头"和"导航栏折叠"。导航条标头 div 包括时一定最小宽度, 显示 3 水平行下面的屏幕, 将出现一个按钮 (所谓"汉堡图标")。使用纯 HTML 和 CSS; 呈现图标没有图像 是必需的。这是显示的图标, 与每个代码标记呈现白色条之一:

```

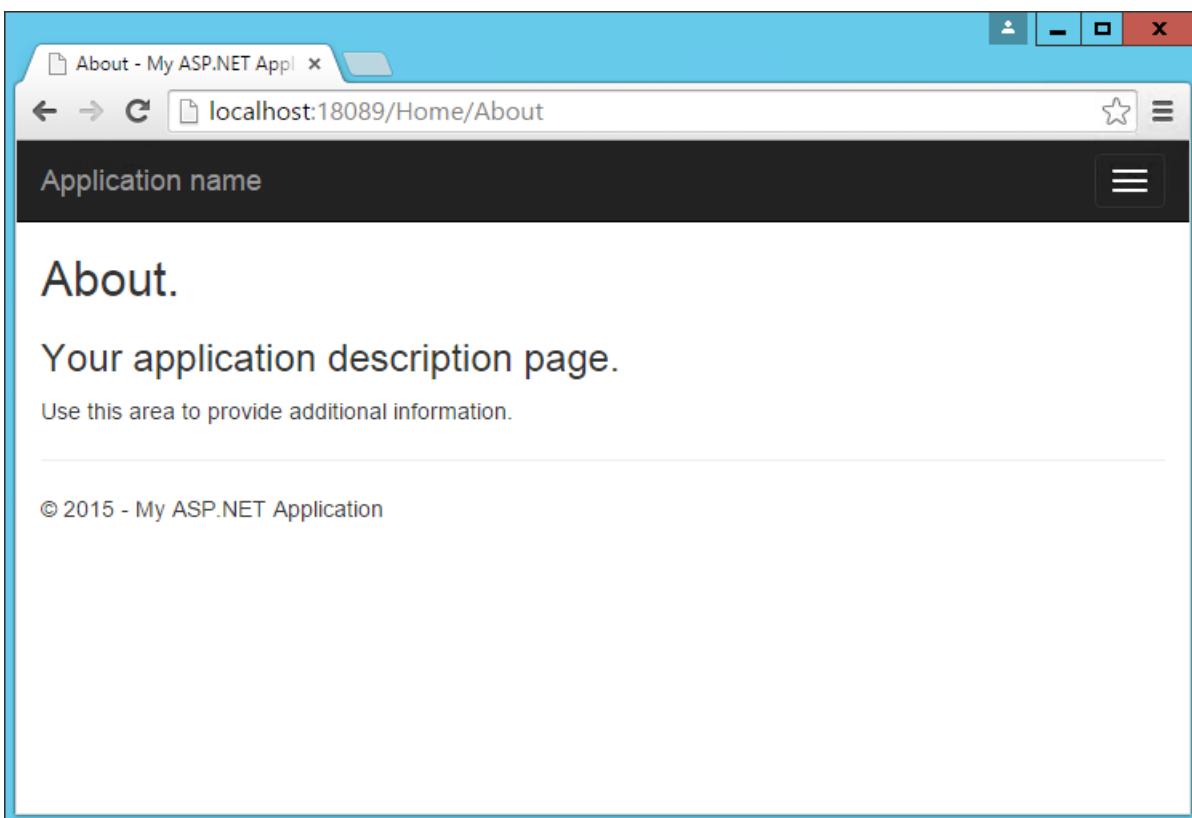
<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>

```

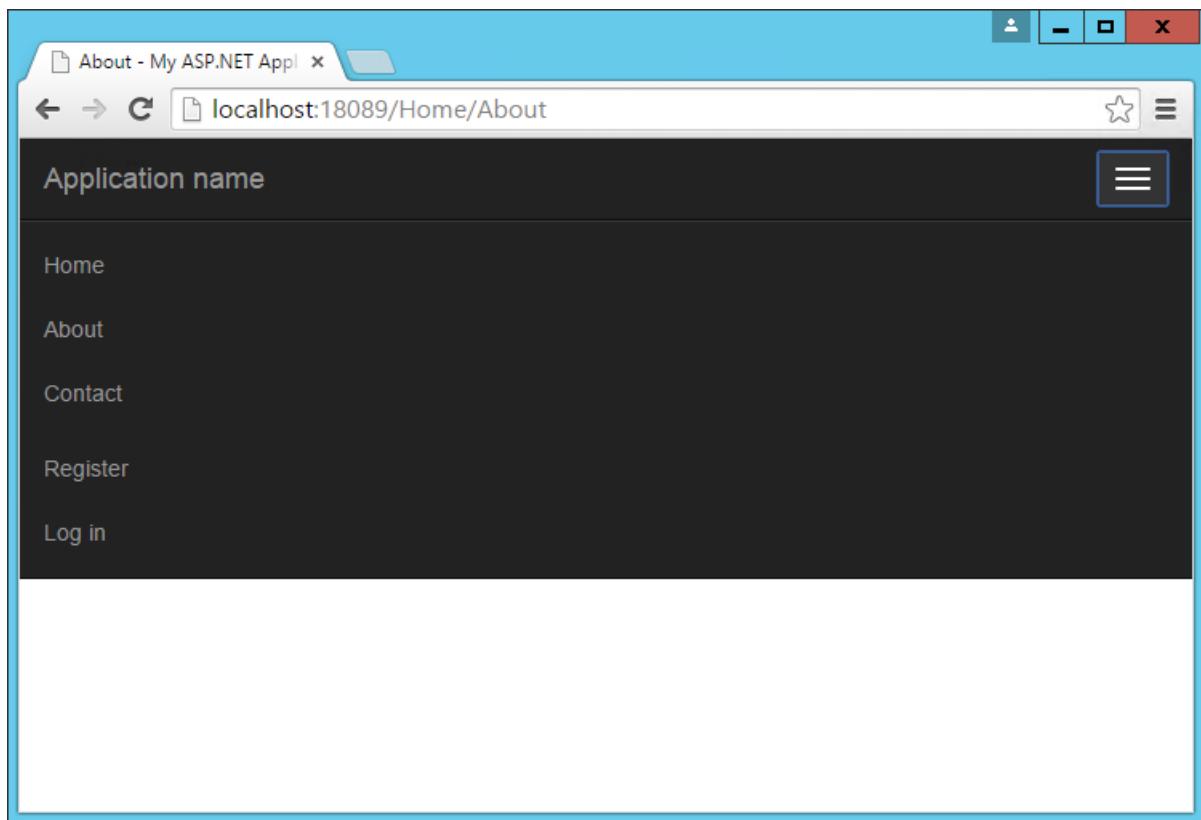
它还包括应用程序名称, 将显示在左上角。由呈现的主导航菜单 `` 中的第二个 `div` 元素并包括指向链接为主页, 关于, 和联系人。下面的导航窗格中, 主正文中的每个页面呈现在另一个 `<div>`、标记与"容器"和"正文内容"类。在此处显示简单的默认 `_Layout` 文件中, 页的内容所呈现页面上, , 然后选择一个简单与关联的特定视图 `<footer>` 添加到末尾 `<div>` 元素。你可以看到有关页面内置将显示使用此模板:



窗口低于一定宽度时，将显示折叠的导航栏中，与在右上角，“汉堡”按钮：



单击图标将显示从页面顶部向下滑垂直抽屉中的菜单项：



版式和链接

Bootstrap 设置了站点的基本版式、颜色和其 CSS 文件中的格式设置的链接。此 CSS 文件包括表、按钮、窗体元素、图像和的详细信息的默认样式 ([了解](#))。一个特别有用功能是，接下来涵盖的网格布局系统。

网格

Bootstrap 的最流行的功能之一是其网格布局系统。现代 web 应用程序应避免使用 `<table>` 布局，而将此元素的用途限制为实际的表格数据的标记。相反，列和行可以进行布局使用一系列 `<div>` 元素和相应的 CSS 类。有许多好处包括能够调整以显示垂直屏幕窄，如在手机上的网格布局，这种方法。

[Bootstrap 的网格布局系统](#)基于 12 个列。已选择此数字，原因是它可以为 1、2、3 或 4 列均匀划分和到之间变化的列宽可以在 1/12 的垂直屏幕的宽度。若要开始使用网格布局系统，你应该开始使用容器 `<div>`，然后添加行 `<div>`，如下所示：

```
<div class="container">
  <div class="row">
    ...
  </div>
</div>
```

接下来，添加其他 `<div>` 元素为每个列，并指定的列数，`<div>` 应占用 (带 12)"列-md-"开头的 CSS 类的一部分。例如，如果你想要只具有大小相等的两个列，你将为每个使用"列-md-6"的类。在这种情况下"md"是短，无法用于"medium"并引用标准尺寸台式计算机的显示大小。有四个不同选项，你可以选择从，和每个将使用更高版本的宽度除非重写 (因此，如果你想要无论屏幕宽度固定的布局，你可以只需指定 xs 类)。

CSS 类前缀	设备层	宽度
col-xs-	手机	< 768px
列-sm-	平板电脑	>= 768px
列-md-	桌面	>= 992px

CSS 类前缀	设备层	宽度
列-lg-	更大的桌面显示	> = 1200px

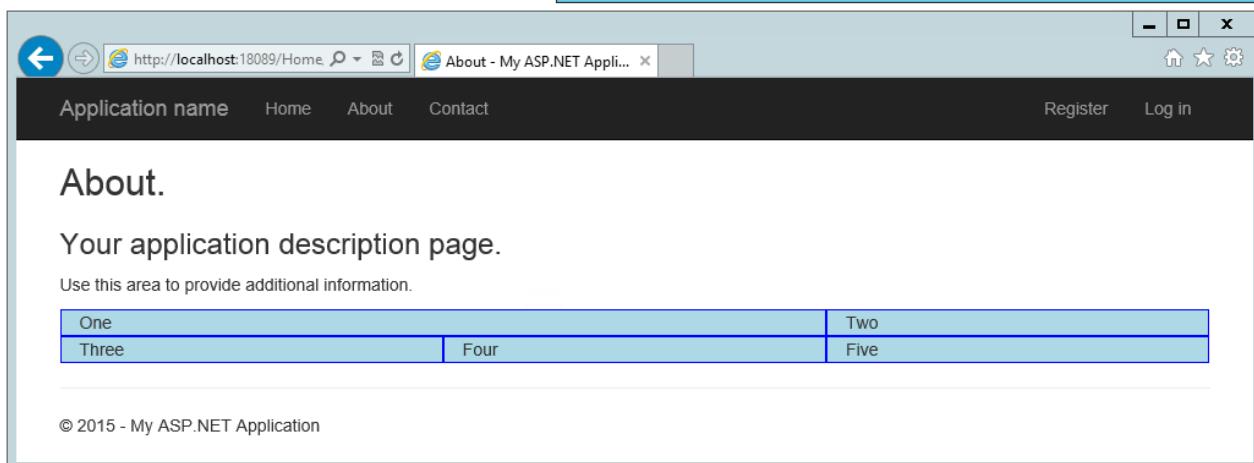
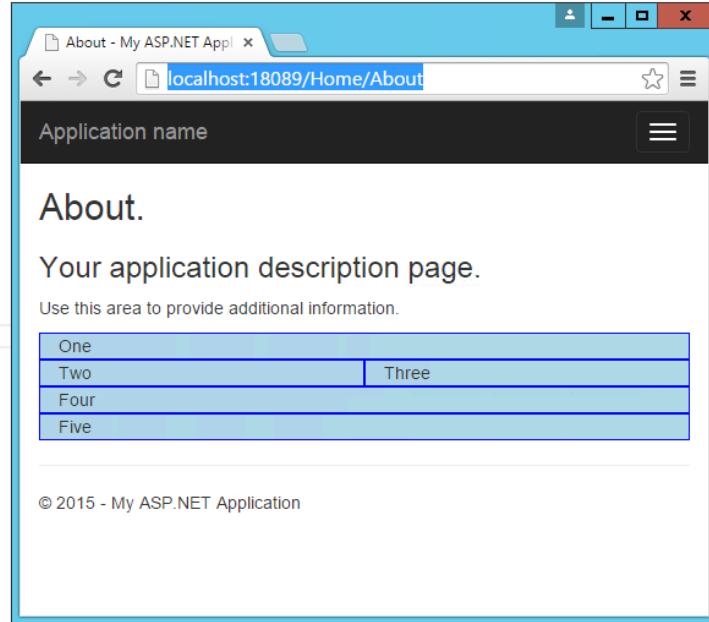
指定的两个列这两个"列-md-6"生成的布局中不会在桌面的解决方法的两个列，但较小的设备（或在桌面上较窄的浏览器窗口），允许用户轻松地查看上呈现时，这两列将垂直堆叠时而无需水平滚动的内容。

Bootstrap 始终默认为单列布局，以便只需指定列，如果希望多个列。你想要显式指定的唯一时间 `<div>` 占用所有 12 列可以重写更大的设备层的行为。在指定多个设备层类时，你可能需要重置在某些点列呈现。添加只会显示某些视区内 clearfix div 可以实现此目的，如下所示：

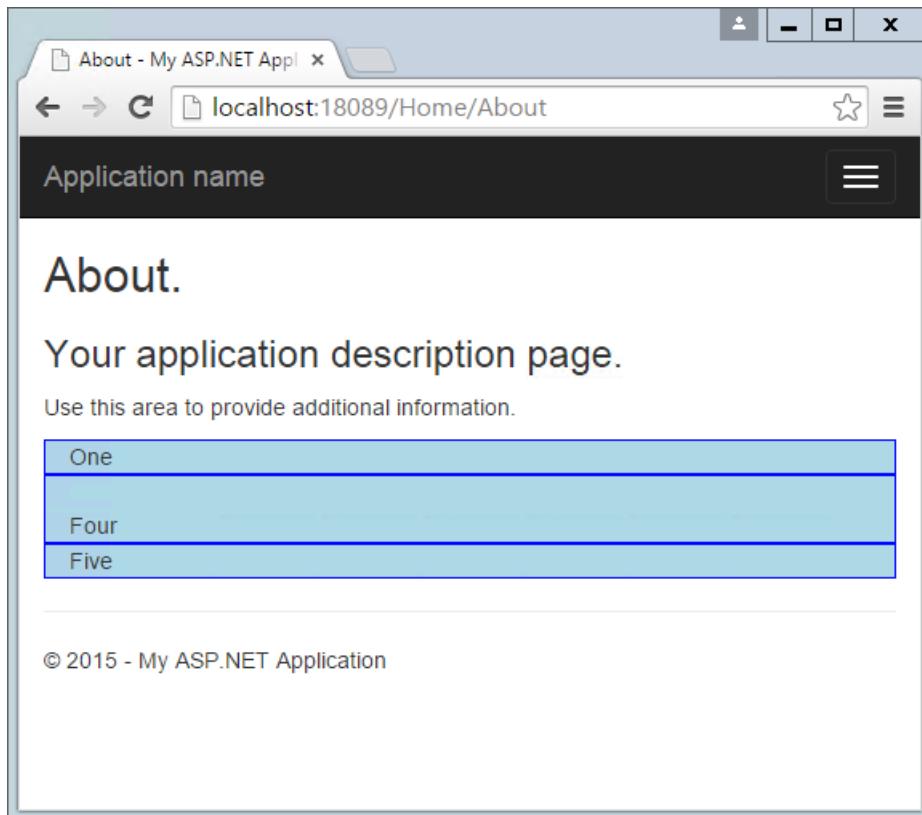
```

<p>Use this area to provide additional information.</p>
<style>
[class*="col-"] {
background-color: lightblue;
border: 1px solid blue;
}
</style>
<div class="container">
<div class="row">
<div class="col-xs-12 col-md-8">
    One
</div>
<div class="col-xs-6 col-md-4">
    Two
</div>
<div class="col-xs-6 col-md-4">
    Three
</div>
<div class="clearfix visible-xs"></div>
<div class="col-xs-12 col-md-4">
    Four
</div>
<div class="col-xs-12 col-md-4">
    Five
</div>
</div>
</div>

```



在上面的示例中，一个和第二个共享中行"md"布局中，而两个和第三共享"xs"布局中的行。而无需 clearfix `<div>`，2 和 3 中未显示正确的"xs"视图（请注意，只有一个、4 和 5 所示）：



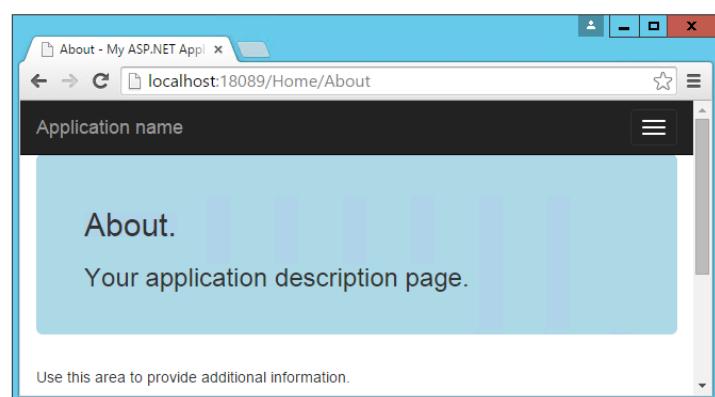
在此示例中，只有一行 `<div>` 所使用的并且启动仍然主要未正确的操作方面的布局和堆叠的列。通常情况下，应指定行 `<div>` 对于每个水平行布局要求，和当然可以嵌套在另一个的 Bootstrap 网格。执行操作时，每个嵌套的网格将占用 100%的元素在其中放置它，然后可以通过使用列类细分的宽度。

Jumbotron

如果你已使用 Visual Studio 2012 或 2013年中的默认 ASP.NET MVC 模板，你可能已了解 Jumbotron 操作中。它将引用到大型全角部分可以用于显示较大的背景图像，操作、旋转或类似的元素调用的页。若要添加到页面 jumbotron，只需添加 `<div>` 并为其提供的类"jumbotron"，然后放置容器 `<div>` 内并添加你的内容。我们可以轻松地调整有关页后，可以使用它显示的主要标题 jumbotron 标准：

```
<style>
    .jumbotron {
        background-color: lightblue;
    }
</style>

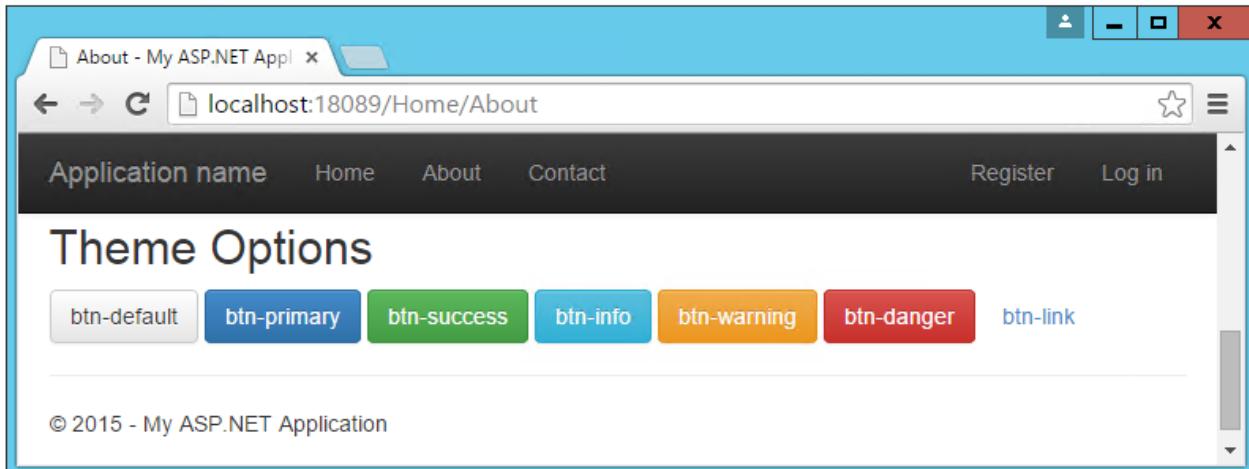
<div class="jumbotron">
    <div class="container">
        <h2>@ViewBag.Title.</h2>
        <h3>@ViewBag.Message</h3>
    </div>
</div>
<p>Use this area to provide additional information.</p>
```



按钮

在下图中显示的默认按钮类和它们的颜色。

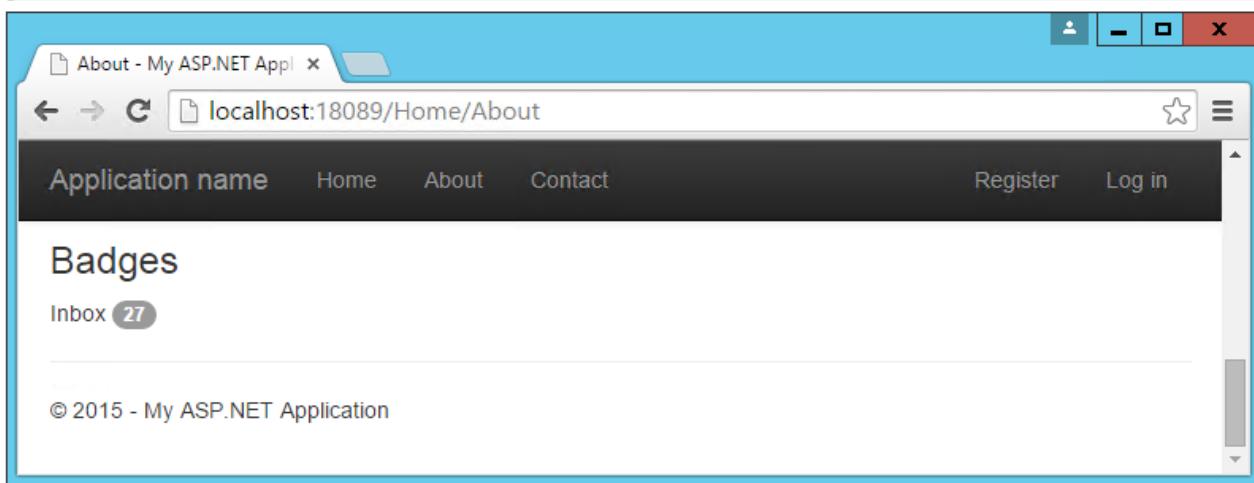
```
<h2>Theme Options</h2>
<p>
    <button type="button" class="btn btn-default">btn-default</button>
    <button type="button" class="btn btn-primary">btn-primary</button>
    <button type="button" class="btn btn-success">btn-success</button>
    <button type="button" class="btn btn-info">btn-info</button>
    <button type="button" class="btn btn-warning">btn-warning</button>
    <button type="button" class="btn btn-danger">btn-danger</button>
    <button type="button" class="btn btn-link">btn-link</button>
</p>
```



徽章

徽章是指导航项旁边的小，通常为数值标注。它们可以指明大量消息或通知等待或更新的状态。指定此类徽章非常简单，只添加 `` 包含文本，与"徽章"的类：

```
<h3>Badges</h3>
<p>
    Inbox <span class="badge">27</span>
</p>
```



警报

你可能需要向应用程序的用户显示通知、警报或错误消息的某种类型。这是标准的警报类是很有用。有四个不同的严重级别与关联的颜色方案：

```

<h3>Alerts</h3>


<strong>Success!</strong> Well done.



<strong>FYI</strong> You might need to know this.

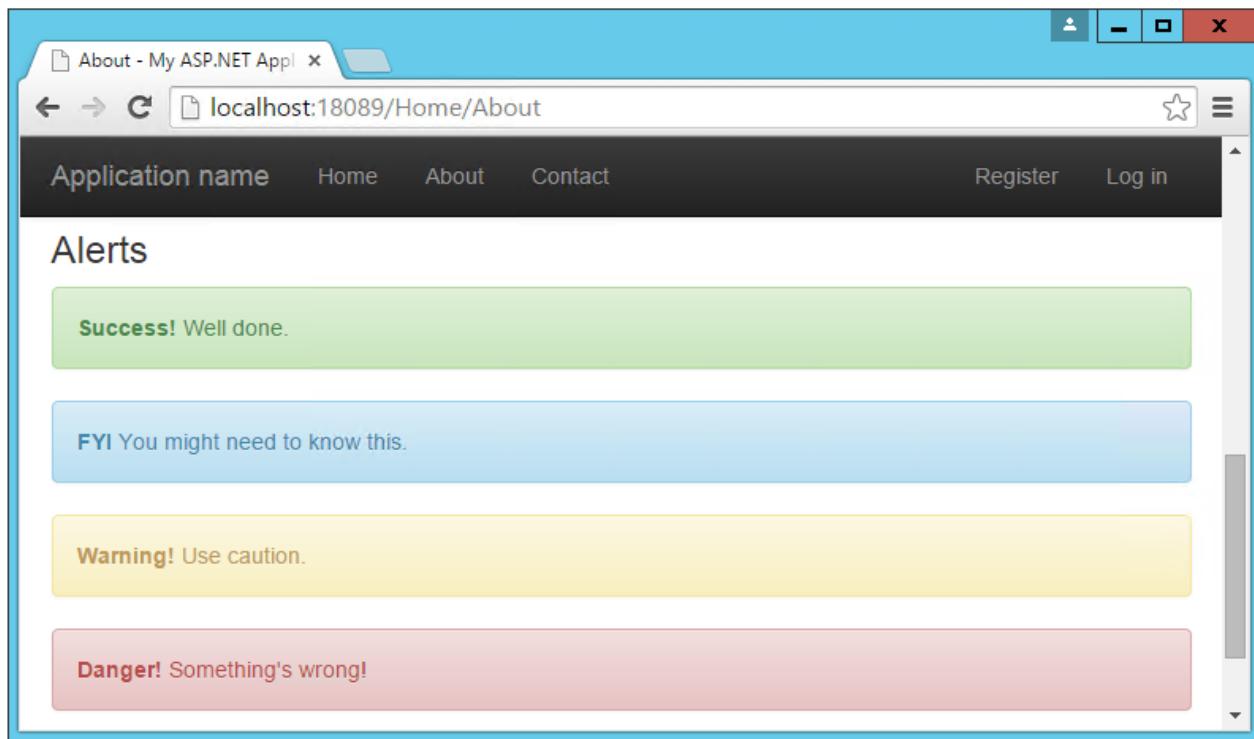


<strong>Warning!</strong> Use caution.



<strong>Danger!</strong> Something's wrong!


```



Navbars 和菜单

我们布局已经包括标准的导航栏中，但是的 Bootstrap 主题支持其他样式选项。我们也很容易可以选择垂直显示导航栏，而不是水平如果，具有首选，以及为添加的子导航中的项弹出菜单。简单导航菜单，选项卡条带，如生成的顶部 `` 元素。这些内容可以创建非常只需通过只需为他们提供的 CSS 类"导航"和"导航选项卡":

```

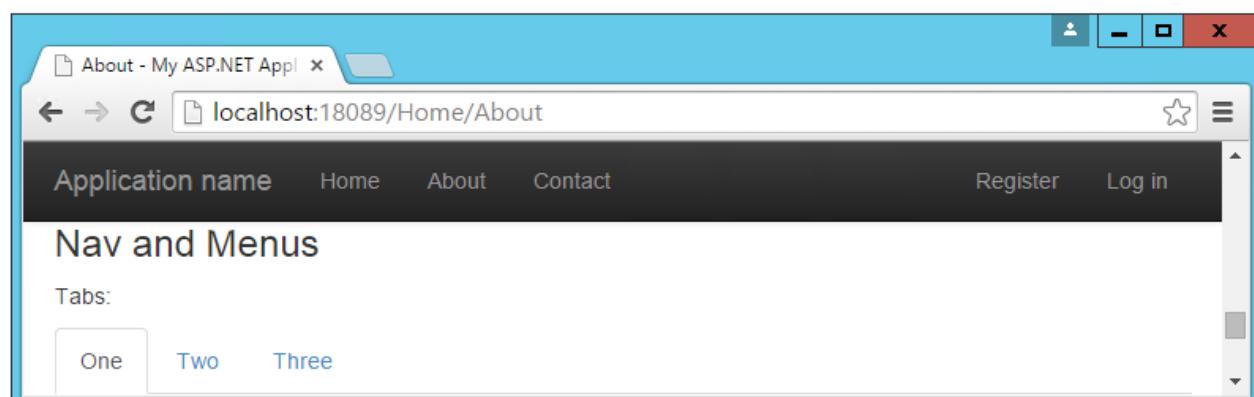
<h3>Nav and Menus</h3>


Tabs:



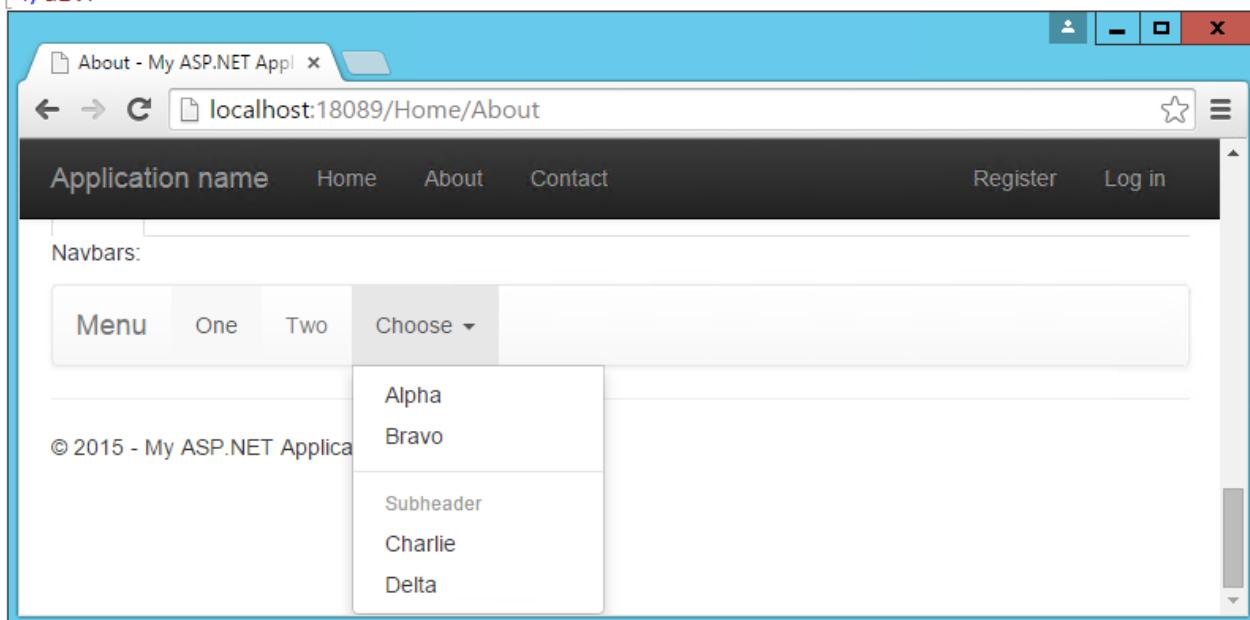
- One
- Two
- Three

```



Nvbars 同样，内置，但较之前更复杂。启动时出现 `<nav>` 或 `<div>` "导航栏"容器 div 其中包含的元素的其余部分的类。我们的页面导航栏在其标题中已包括 – 所示只需对此进行扩展，添加对下拉菜单中的支持：

```
<p>Nvbars:</p>
<div class="navbar navbar-default">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target=".navbar-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Menu</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">One</a></li>
                <li><a href="#">Two</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false">Choose <span class="caret"></span>
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Alpha</a></li>
                        <li><a href="#">Bravo</a></li>
                        <li class="divider"></li>
                        <li class="dropdown-header">Subheader</li>
                        <li><a href="#">Charlie</a></li>
                        <li><a href="#">Delta</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</div>
```



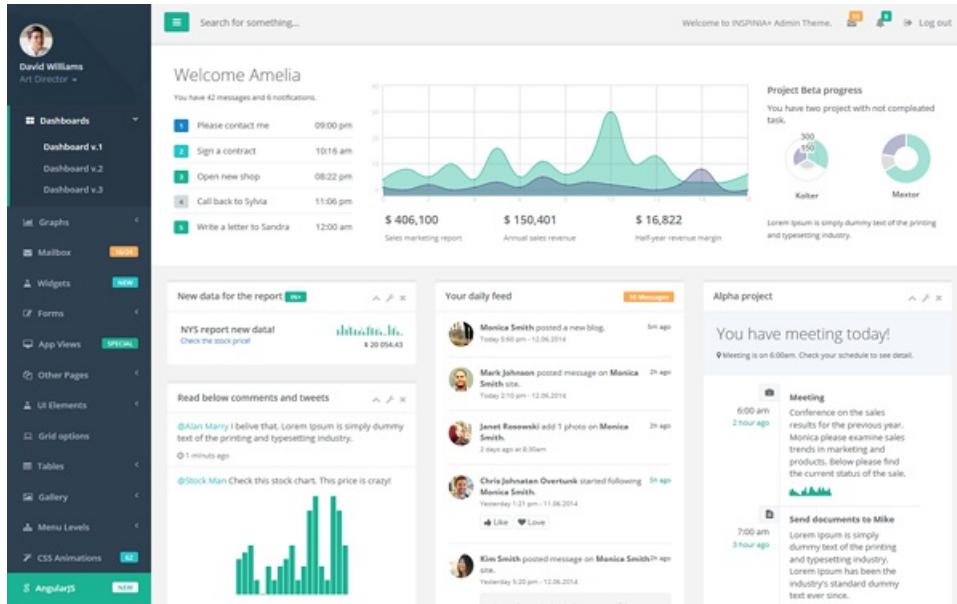
其他元素

此外可以使用默认主题在格式设置精美的样式，包括支持条带化的视图中呈现 HTML 表。有类似的按钮的样式的标签。你可以创建自定义支持的标准 HTML 之外的其他样式选项的下拉列表菜单 `<select>` 元素及其 Navbars 我们默认入门站点已在使用所示。如果你需要一个进度栏，有几种样式可供选择，以及列出组和面板，包括标题和内容。浏览标准的 Bootstrap 主题中的其他选项：

<http://getbootstrap.com/examples/theme/>

更多主题

你可以通过重写某些或所有其 CSS，扩展的标准的 Bootstrap 主题调整颜色和样式，以满足自己的应用程序的需要。如果你想要从头现成的主题，有几个主题库可用联机，在启动主题，例如 WrapBootstrap.com（其的各种商业主题）和 Bootswatch.com（它提供了可用的主题）的专用化。某些付费的可用模板提供大量之上的基本的 Bootstrap 主题，如对管理菜单和仪表板的丰富支持的功能丰富的图表和仪表。常用的付费模板示例目前将 Inspinia, \$18，其中包括除了 AngularJS 和静态 HTML 版本的 ASP.NET MVC5 模板的销售。示例屏幕快照所示。



如果你想要更改您的 Bootstrap 主题，请将放bootstrap.css主题中所需的文件wwwroot/css文件夹并将更改中的引用_Layout.cshtml以将其指向。更改所有环境的链接：

```
<environment names="Development">
    <link rel="stylesheet" href="~/css/bootstrap.css" />
```

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

如果你想要生成你自己的仪表板，你可以从提供的免费示例从这里开始：

<http://getbootstrap.com/examples/dashboard/>。

组件数

Bootstrap 除了已经讨论了这些元素，包括对各种支持[内置 UI 组件](#)。

Glyphicons

Bootstrap 包括从 Glyphicons 图标集 (<http://glyphicons.com>)，与超过 200 个自由可用于启用 Bootstrap 的 web 应用程序中使用的图标。下面是只是一个小的示例：

indent-right	facetime-video	picture	map-marker	adjust			
glyphicon glyphicon-check	glyphicon glyphicon-move	glyphicon glyphicon-step-backward	glyphicon glyphicon-fast-backward	glyphicon glyphicon-backward	glyphicon glyphicon-play	glyphicon glyphicon-pause	glyphicon glyphicon-stop
glyphicon glyphicon-forward	glyphicon glyphicon-fast-forward	glyphicon glyphicon-step-forward	glyphicon glyphicon-eject	glyphicon glyphicon-chevron-left	glyphicon glyphicon-chevron-right	glyphicon glyphicon-plus-sign	glyphicon glyphicon-minus-sign
glyphicon glyphicon-remove-sign	glyphicon glyphicon-ok-sign	glyphicon glyphicon-question-sign	glyphicon glyphicon-info-sign	glyphicon glyphicon-screenshot	glyphicon glyphicon-remove-circle	glyphicon glyphicon-ok-circle	glyphicon glyphicon-ban-circle
glyphicon glyphicon-arrow-left	glyphicon glyphicon-arrow-right	glyphicon glyphicon-arrow-up	glyphicon glyphicon-arrow-down	glyphicon glyphicon-share-alt	glyphicon glyphicon-resize-full	glyphicon glyphicon-resize-small	glyphicon glyphicon-exclamation-sign
glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon

输入的组

输入的组允许绑定的附加文本或按钮使用输入元素，从而为用户提供更直观的体验：

Recipient's username	@example.com
----------------------	--------------

痕迹导航

痕迹导航是用于显示用户，其最新历史记录或在站点的导航层次结构中的深度的常见 UI 组件。通过将"痕迹导航"类应用于任何轻松地添加 `` 列表元素。包括内置支持分页上使用"分页"类 `` 中的元素 `<nav>`。通过使用添加响应嵌入的幻灯片和视频 `<iframe>`，`<embed>`，`<video>`，或 `<object>` Bootstrap 将自动设置样式的元素。通过使用特定的类，如"嵌入的响应性-16by9"中指定特定的纵横比。

JavaScript 支持

Bootstrap 的 JavaScript 库包括对包含的组件，使您可以控制其行为以编程方式在你的应用程序的 API 支持。此外，`bootstrap.js` 包括超过 12 个自定义 jQuery 插件，提供其他功能，例如转换，模式对话框，向下滚动（更新样式基于用户具有滚动文档中的何处）的检测，因此，它们不会滚动出屏幕折叠到窗口的行为、行李传送带和将附加菜单。没有足够的空间来涵盖所有 Bootstrap – 若要了解详细信息，请访问内置的 JavaScript 外接程序 <http://getbootstrap.com/javascript/>。

总结

Bootstrap 提供一个可用来快速和高效布局和样式各种网站和应用程序的 web 框架。其基本版式和样式提供通过可以手工编写或商业上购买的自定义主题支持可以容易地操作获得愉快外观和感觉。它支持的 web 组件在过去会要求昂贵的第三方控件，若要完成，同时支持现代和打开 web 标准的主机。

小子, Sass 和 ASP.NET Core 中出色的字体

2018/4/10 • 14 min to read • [Edit Online](#)

作者:Steve Smith

Web 应用程序的用户有越来越高的期望时设置的样式和总体体验。现代 web 应用程序频繁地利用丰富的工具和框架用于定义和管理其外观和感觉以一致的方式。框架喜欢Bootstrap可以地长定义一组通用的样式和网站的布局选项。但是, 大多数重要站点还受益于能够有效地定义和维护样式和级联样式表 (CSS) 文件, 以及能够轻松访问帮助使站点的接口更直观的非图像图标。就是在此处的支持语言和工具较少和Sass, 库等, 以及将字体出色, 进入。

CSS 预处理器语言

为了改进的体验的使用基础的语言中, 编译成其它语言的语言被称为预处理器。有两个常用的预处理器针对 CSS: 不太和 Sass。这些预处理器将功能添加到 CSS, 如对变量和嵌套的规则, 以改进可维护性的大型、复杂的样式表的支持。作为一种语言的 CSS 是非常基本, 缺少那样简单变量, 即使对于支持, 这往往会重复和臃肿使 CSS 文件。添加通过预处理器的实际编程语言功能可以帮助减少重复, 提供更好地组织样式规则。Visual Studio 提供这两个较低的内置支持和 Sass, 以及可以进一步提高开发体验, 使用这些语言时的扩展。

作为预处理器可以如何改进可读性和可维护性的样式信息的快速示例, 请考虑以下 CSS:

```
.header {  
    color: black;  
    font-weight: bold;  
    font-size: 18px;  
    font-family: Helvetica, Arial, sans-serif;  
}  
  
.small-header {  
    color: black;  
    font-weight: bold;  
    font-size: 14px;  
    font-family: Helvetica, Arial, sans-serif;  
}
```

使用更少, 这可以重写以消除所有重复, 使用*mixin* (这样命名是因为它允许你"中混合使用"从一个类或到另一个规则集的属性):

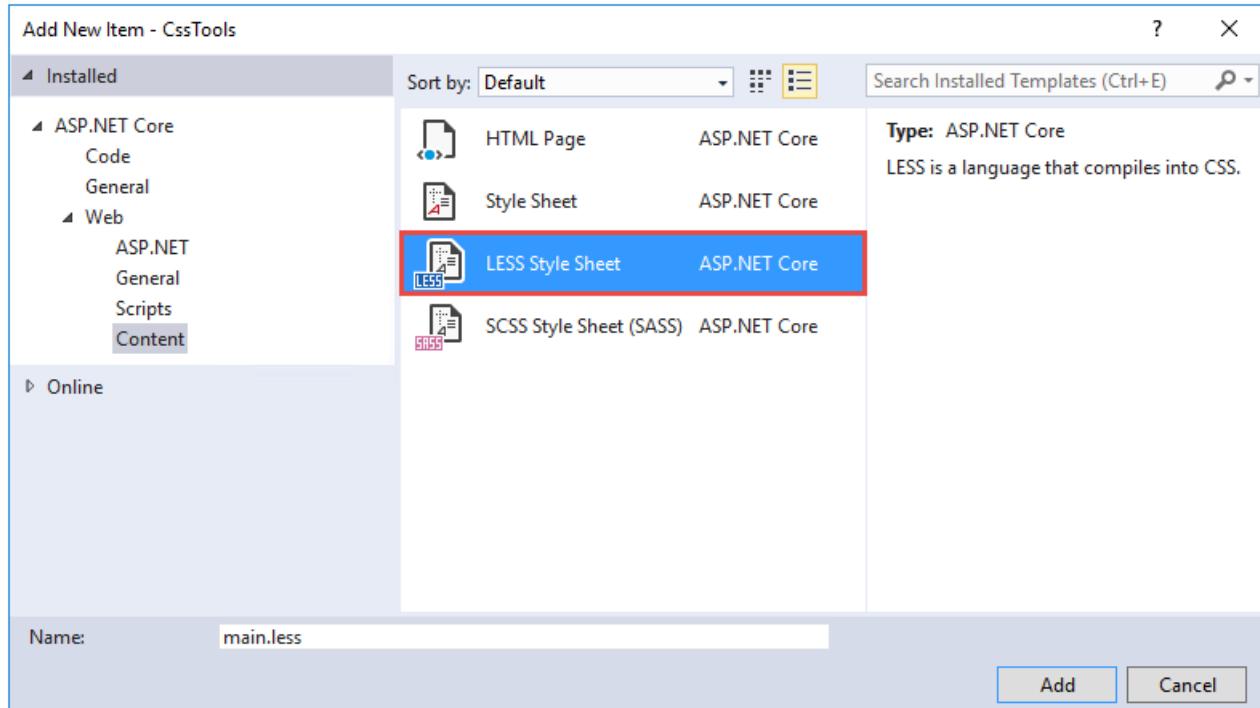
```
.header {  
    color: black;  
    font-weight: bold;  
    font-size: 18px;  
    font-family: Helvetica, Arial, sans-serif;  
}  
  
.small-header {  
    .header;  
    font-size: 14px;  
}
```

小子

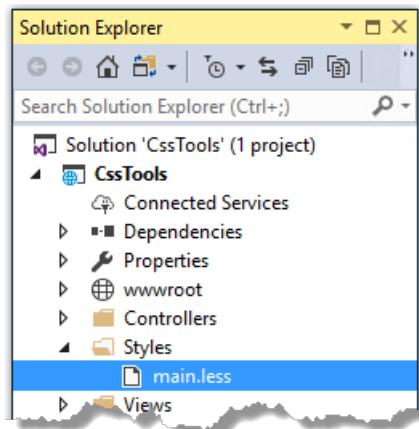
使用 Node.js 不太 CSS 预处理器运行。若要安装小子, 使用 Node 程序包管理器 (npm) 从命令提示符 (-g 意味着"全局"):

```
npm install -g less
```

如果你使用 Visual Studio, 你可以开始使用小于通过将一个或多个更少文件添加到你的项目, 然后配置 Gulp (或 Grunt) 在编译时处理它们。添加样式到你的项目的文件夹, 然后添加新的名为的文件小于*main.less*到此文件夹。



添加后, 您的文件夹结构应如下所示:



现在你可以添加到文件中, 这将编译到 CSS 并部署到的 wwwroot 文件夹 Gulp 的一些基本的样式。

修改*main.less*为包括以下内容, 这将从一种基颜色创建一个简单的调色板。

```

@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
    background-color:@background;
}
.baseColor {color:@base}
.bgLight {color:@lighter}
.bgLight2 {color:@lighter2}
.bgDark {color:@darker}
.bgDark2 {color:@darker2}

```

`@base` 和其他`@-prefixed`项是变量。每个表示一种颜色。除 `@base`，设置它们并使用颜色函数：加亮、变暗，和旋转。淡化和加深执行几乎你将预期；数值调节钮调整颜色色调的大量度（围绕颜色盘中）。较少处理器是足够智能，可忽略不使用的变量，因此若要展示了这些变量的工作原理，我们需要某个位置使用它们。类 `.baseColor`，等将演示每个生成的 CSS 文件中的变量的计算的值。

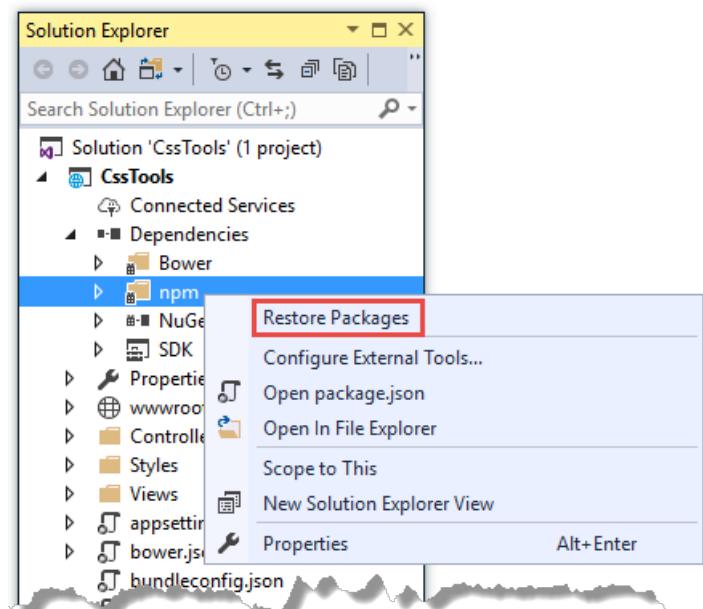
入门

创建npm 配置文件(`package.json`) 在你的项目文件夹并编辑它以引用 `gulp` 和 `gulp-less`：

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-less": "3.3.0"
  }
}
```

在你的项目文件夹，或在 Visual Studio 中安装的依赖关系，请在命令提示符下解决方案资源管理器(依赖项 > npm > 还原程序包)。

```
npm install
```



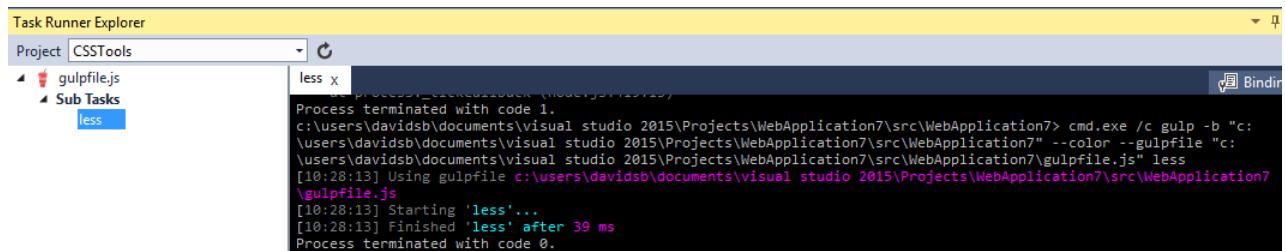
在项目文件夹中，创建**Gulp 配置文件**(*gulpfile.js*)以定义自动的过程。在更少，表示的文件及要运行更少的任务的顶部添加一个变量：

```
var gulp = require("gulp"),
  fs = require("fs"),
  less = require("gulp-less");

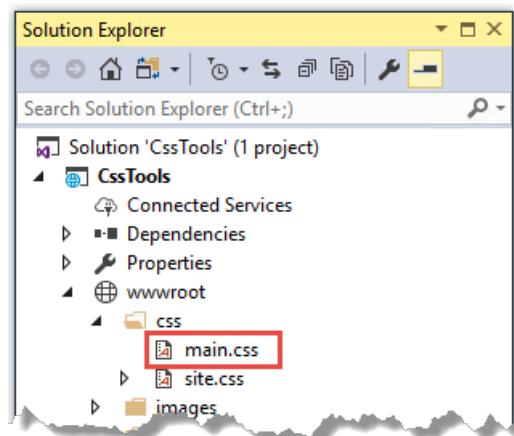
gulp.task("less", function () {
  return gulp.src('Styles/main.less')
    .pipe(less())
    .pipe(gulp.dest('wwwroot/css'));
});
```

打开任务运行程序资源管理器(视图 > 其他 Windows > 任务运行程序资源管理器)。在任务之间，你应看到一个名为的新任务 **less**。你可能必须刷新窗口。

运行 **less** 任务，并看到类似于此处所示的输出：



*Wwwroot/css*文件夹现在包含一个新的文件，*main.css*：



打开*main.css*，你将看到与下面类似：

```

body {
    background-color: #336666;
}
.baseColor {
    color: #663333;
}
.bgLight {
    color: #884a44;
}
.bgLight2 {
    color: #aa6355;
}
.bgDark {
    color: #442225;
}
.bgDark2 {
    color: #221114;
}

```

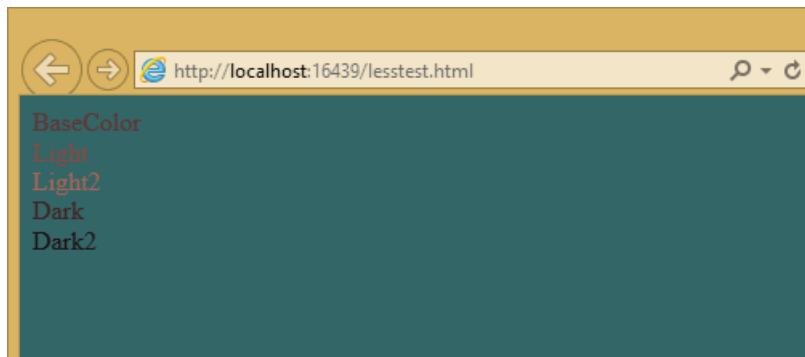
添加到一个简单的 HTML 页面wwwroot文件夹, 然后引用main.css若要查看操作中的调色板。

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="css/main.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        <div class="baseColor">BaseColor</div>
        <div class="bgLight">Light</div>
        <div class="bgLight2">Light2</div>
        <div class="bgDark">Dark</div>
        <div class="bgDark2">Dark2</div>
    </div>
</body>
</html>

```

你可以看到上旋转 180 度 @base 用于生成 @background 导致相反颜色的颜色盘 @base :



小于还提供对嵌套的规则以及嵌套的媒体查询的支持。例如, 像菜单可能会导致详细 CSS 规则的定义嵌套层次结构像这样:

```

nav {
    height: 40px;
    width: 100%;
}
nav li {
    height: 38px;
    width: 100px;
}
nav li a:link {
    color: #000;
    text-decoration: none;
}
nav li a:visited {
    text-decoration: none;
    color: #CC3333;
}
nav li a:hover {
    text-decoration: underline;
    font-weight: bold;
}
nav li a:active {
    text-decoration: underline;
}

```

理想情况下的所有相关的样式规则将被放在一起在 CSS 文件中，但在实践中没有任何强制实施此规则约定和可能块注释除外。

定义使用小于这些相同的规则类似如下所示：

```

nav {
    height: 40px;
    width: 100%;
    li {
        height: 38px;
        width: 100px;
        a {
            color: #000;
            &:link { text-decoration:none}
            &:visited { color: #CC3333; text-decoration:none}
            &:hover { text-decoration:underline; font-weight:bold}
            &:active {text-decoration:underline}
        }
    }
}

```

请注意，在此情况下，所有的从属元素 `nav` 包含在其作用域内。不再父元素的任何重复 (`nav`, `li`, `a`)，并且总的行计数也已删除（尽管某些就是将值放在第二个示例的同一行上的结果）。它可以是非常有帮助，组织，若要查看的所有规则的给定的用户界面元素在显式限定范围内，在这种情况下设置从文件的其余部分由大括号。

`&` 语法是较少的选择器功能，与 `(&)` 表示当前的选择器父级。这样，在 `{...}` 块中，`&` 表示 `a` 标记，因此 `&:link` 等效于 `a:link`。

创建响应式设计非常有用的媒体查询还可能影响很大程度重复和 CSS 中的复杂性。小于允许媒体查询嵌套在类，以便整个类定义不需要重复内不同顶级 `@media` 元素。例如，下面是响应式菜单的 CSS:

```

.navigation {
    margin-top: 30%;
    width: 100%;
}
@media screen and (min-width: 40em) {
    .navigation {
        margin: 0;
    }
}
@media screen and (min-width: 62em) {
    .navigation {
        width: 960px;
        margin: 0;
    }
}

```

这可以更好地定义以秒为：

```

.navigation {
    margin-top: 30%;
    width: 100%;
    @media screen and (min-width: 40em) {
        margin: 0;
    }
    @media screen and (min-width: 62em) {
        width: 960px;
        margin: 0;
    }
}

```

小于我们已经看到的另一个功能是其支持的数学运算，允许从预定义的变量构造的样式特性。这使得更新相关的样式容易得多，因为可以修改基变量和所有依赖值自动更改。

CSS 文件，尤其是对于大型站点（和尤其是如果正在使用媒体查询），就会变得很大随着时间推移，从而使用它们难以操作。Less 文件可以单独定义，然后一起使用，取出 `@import` 指令。小于还可导入单个 CSS 文件，同样，如果需要。

*Mixins*可以接受参数，并且小于 mixin 防护，提供声明性方式定义某些 mixins 生效的形式支持条件逻辑。Mixin 临界条件的一个常见用途是调整颜色如何基于或深色的源颜色是。Mixin 防护给定 mixin 接受颜色的参数，用于修改基于该颜色 mixin：

```

.box (@color) when (lightness(@color) >= 50%) {
    background-color: #000;
}
.box (@color) when (lightness(@color) < 50%) {
    background-color: #FFF;
}
.box (@color) {
    color: @color;
}

.feature {
    .box (@base);
}

```

给定我们当前 `@base` 值 `#663333`，小于此脚本将生成以下 CSS：

```
.feature {  
    background-color: #FFF;  
    color: #663333;  
}
```

小于提供许多其他功能，但这应为你提供此强大的一些思路预处理语言。

Sass

Sass 类似于较少，是为许多相同功能，但使用略有不同的语法提供支持。它使用 Ruby，而不是 JavaScript 中，生成，因此不同的安装要求。原始 Sass 语言未使用大括号或分号，但改为定义使用空白和缩进的作用域。在 Sass 3 版本中，引入了新的语法，**SCSS** ("Sassy CSS")。SCSS 是类似于 CSS，可忽略缩进级别和空格，并改为使用分号和大括号。

若要安装 Sass，通常你将首先安装 Ruby（预安装在 macOS 上），然后运行：

```
gem install sass
```

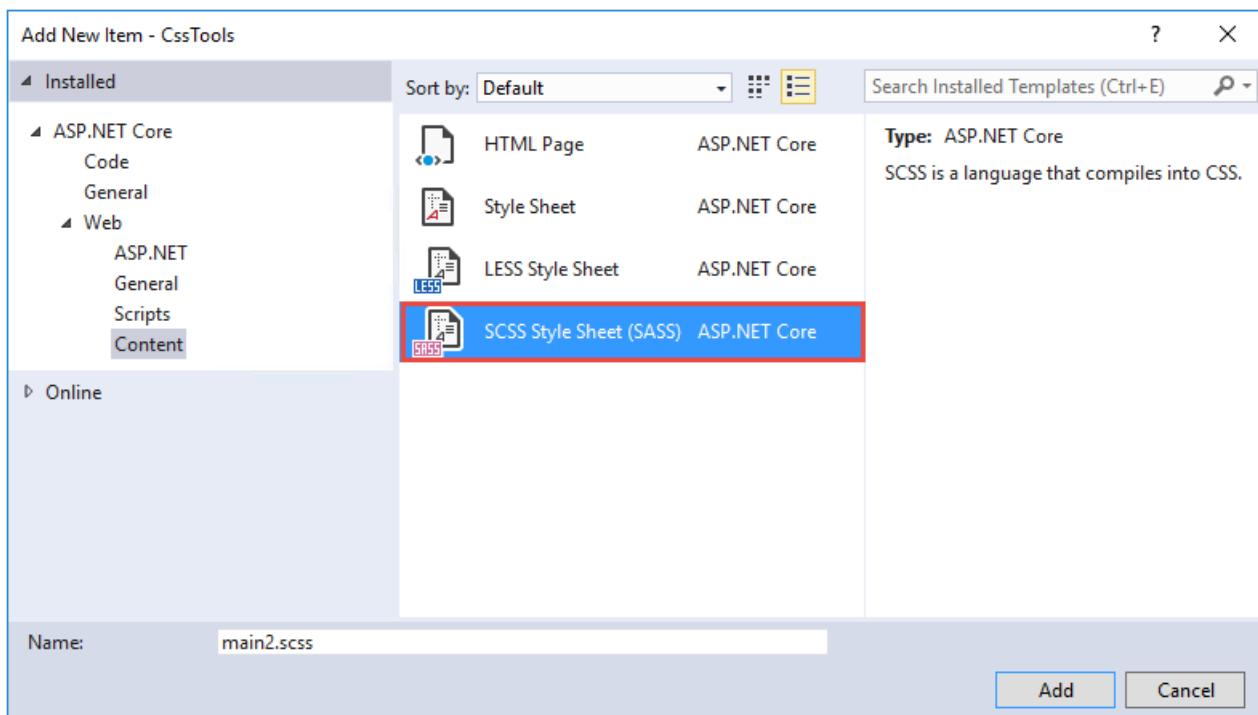
但是，如果要运行 Visual Studio，你可以开始使用 Sass 在很大程度的相同方式就像对待小于。打开 *package.json* 并添加到 "gulp sass" 包 `devDependencies`：

```
"devDependencies": {  
    "gulp": "3.9.1",  
    "gulp-less": "3.3.0",  
    "gulp-sass": "3.1.0"  
}
```

接下来，修改 *gulpfile.js* 添加 sass 变量和编译 Sass 文件并将结果放的 wwwroot 文件夹中的任务：

```
var gulp = require("gulp"),  
    fs = require("fs"),  
    less = require("gulp-less"),  
    sass = require("gulp-sass");  
  
// other content removed  
  
gulp.task("sass", function () {  
    return gulp.src('Styles/main2.scss')  
        .pipe(sass())  
        .pipe(gulp.dest('wwwroot/css'));  
});
```

现在，您可以将 Sass 文件 *main2.scss* 到样式项目的根目录中的文件夹：



打开 `main2.scss`，添加以下内容：

```
$base: #CC0000;
body {
    background-color: $base;
}
```

保存你所有文件。现在刷新任务运行程序资源管理器，你看到 `sass` 任务。运行它，并查找 `*/wwwroot/css` 文件夹。现在有了 `main2.css*` 文件，使用以下内容：

```
body {
    background-color: #CC0000;
}
```

Sass 支持嵌套大体相同时，小于存在，提供类似的好处。文件可以由函数拆分和包含使用 `@import` 指令：

```
@import 'anotherfile';
```

Sass 支持 mixins 同样，使用 `@mixin` 关键字定义它们和 `@include` 以便将它们，包含从如此示例所示 [sass lang.com](#)：

```
@mixin border-radius($radius) {
    -webkit-border-radius: $radius;
    -moz-border-radius: $radius;
    -ms-border-radius: $radius;
    border-radius: $radius;
}

.box { @include border-radius(10px); }
```

Sass 除了 mixins，它还支持继承，的概念允许一个类，以扩展另一个。它是从概念上讲类似于 mixin，但在更少的 CSS 代码中的结果。它通过 `@extend` 关键字。若要尝试 mixins，将以下代码添加到你 `main2.scss` 文件：

```

@mixin alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @include alert;
    border-color: green;
}

.error {
    @include alert;
    color: red;
    border-color: red;
    font-weight:bold;
}

```

检查在输出 `main2.css` 运行之后 `sass` 任务中任务运行程序资源管理器:

```

.success {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
    border-color: green;
}

.error {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
    color: red;
    border-color: red;
    font-weight: bold;
}

```

请注意所有警报 mixin 的常见属性在每个类中有重复。Mixin 好地帮助消除重复在开发时，但它仍包含中的重复，从而导致大于必要 CSS 文件-一个潜在的性能问题很多要创建 CSS。

现在将与警报 mixin `.alert` 类，并更改 `@include` 到 `@extend` (记住来扩展 `.alert`，而不 `alert`):

```

.alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @extend .alert;
    border-color: green;
}

.error {
    @extend .alert;
    color: red;
    border-color: red;
    font-weight:bold;
}

```

运行 Sass 次，并检查生成的 CSS:

```
.alert, .success, .error {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    border-color: green;
}

.error {
    color: red;
    border-color: red;
    font-weight: bold;
}
```

现在仅为根据需要多次定义属性，并更好地生成 CSS。

Sass 还包括函数和条件逻辑操作，类似于小。事实上，这两种语言的功能是非常相似。

较低或 Sass？

仍没有关于是否通常更好的做法将占用更少或 Sass 没有共识（或甚至是否首选原始 Sass 中的较新 SCSS 语法）。可能的最重要的决定是使用这些工具之一，而不是只使用手工编码 CSS 文件。一旦你所做的决策，这两个不太和 Sass 是不错的选择。

出色的字体

除了 CSS 预处理器样式现代 web 应用程序的另一个很好的办法是字体出色。字体 Awesome 是一个工具包，提供 500 多个可以自由地使用 web 应用程序中的可缩放的向量图标。它最初设计成可使用 Bootstrap，但它并不依赖于该框架或在任何 JavaScript 库。

若要开始使用字体出色的最简单方法是添加对它，使用其公用内容交付网络 (CDN) 位置的引用：

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

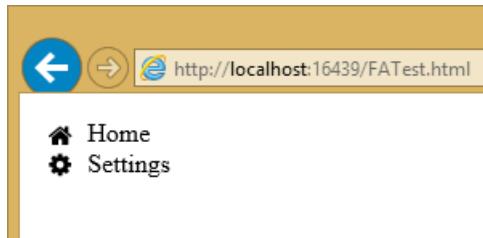
你可以还将其添加到你的 Visual Studio 项目通过将它添加到“依赖关系”中 bower.json：

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.0.0",
    "jquery": "1.10.2",
    "jquery-validation": "1.11.1",
    "jquery-validation-unobtrusive": "3.2.2",
    "hammer.js": "2.0.4",
    "bootstrap-touch-carousel": "0.8.0",
    "Font-Awesome": "4.3.0"
  }
}
```

指向字体出色页上的引用后，你可以将图标添加到你的应用程序通过应用字体出色的类，通常前缀为“fa-”，为嵌入式 HTML 元素（如 `` 或 `<i>`）。例如，你可以将图标添加到简单列表和菜单使用如下代码：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
    <ul class="fa-ul">
        <li><i class="fa fa-li fa-home"></i> Home</li>
        <li><i class="fa fa-li fa-cog"></i> Settings</li>
    </ul>
</body>
</html>
```

这将生成浏览器中的以下-请注意每个项旁边的图标：



你可以查看可用的图标的完整列表：

<http://fontawesome.io/icons/>

总结

现代 web 应用程序越来越要求清晰、直观，编写和更易于使用的各种设备从的响应速度快、流体设计。管理实现这些目标所需的 CSS 样式表的复杂性最好的做法是不太使用预处理器类似或 Sass。此外，如字体出色的工具包快速提供的已知图标添加到文本导航菜单和按钮，提高整体用户体验的应用程序。

在 ASP.NET 核心中的捆绑和 minify 静态资产

2018/5/17 • 12 min to read • [Edit Online](#)

作者: Scott Addie

本文介绍了应用绑定和缩减，包括如何使用 ASP.NET Core web apps 使用这些功能的好处。

绑定和缩减是什么？

绑定和缩减是可以应用的 web 应用中的两个不同的性能优化。一起使用时，绑定和缩减提高性能通过减少服务器请求数和减少的请求的静态资产的大小。

绑定和缩减主要提高第一个页面请求加载时间。一旦已请求网页上，浏览器缓存静态资产（JavaScript、CSS 和图像）。因此，绑定和缩减不时提高性能请求的同一页面或页，请求相同的资产在同一站点上。如果过期在资产上未正确设置标头和如果未使用绑定和缩减，浏览器的新鲜度试探方法将标记资产陈旧在几天后。此外，浏览器需要为每个资产的验证请求。在这种情况下，绑定和缩减提供在第一个页面请求后的提高性能。

绑定

绑定将多个文件合并到单个文件。绑定可减少的所需呈现 web 资产，例如 web 页的服务器请求数。可以专门为 CSS、JavaScript 等创建任意数量的各项捆绑。少选一些文件意味着从浏览器到服务器或提供你的应用程序的服务的少数几个 HTTP 请求。第一个页面加载性能改善中的此结果。

缩减

缩减从代码中移除而无需更改功能的不必要的字符。结果是显著的大小减少请求资产（如 CSS、映像和 JavaScript 文件）中。常见的缩减的负面影响包括缩短为一个字符的变量名和删除注释和多余的空格。

请考虑下面的 JavaScript 函数：

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
    //<signature>
    //<summary> Adds an alt tab to the image
    // </summary>
    //<param name="imgElement" type="String">The image selector.</param>
    //<param name="ContextForImage" type="String">The image context.</param>
    //</signature>
    var imageElement = $(imageTagAndImageID, imageContext);
    imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

缩减缩小为以下函数：

```
AddAltToImg=function(n,t){var i=$(n,t);i.attr("alt",i.attr("id").replace(/ID/,""))};
```

除了删除注释和多余的空格，则以下参数和变量已重命名名称，如下所示：

原始	重命名
imageTagAndImageID	t
imageContext	a

原始	重命名
imageElement	r

绑定和缩减的影响

下表概述了单独加载资产与使用绑定和缩减之间的差异：

操作	与 B/M	而无需 B/M	更改
文件请求	7	18	157%
传输的 KB	156	264.68	70%
加载时间 (ms)	885	2360	167%

浏览器是相当详细方面 HTTP 请求标头。发送的总字节数度量值绑定时见到显著减少。加载时显示的重大进步，但此示例中本地运行。使用资产的绑定和缩减通过网络传输时，在实现更高的性能增益。

选择绑定和缩减策略

MVC 和 Razor 页项目模板提供的绑定和缩减包含的 JSON 配置文件的现成可用解决方案。第三方工具，如[Gulp](#)和[Grunt](#)任务流道，完成相同任务的更多的复杂性。第三方工具开发工作流需要超出绑定和缩减的处理时非常适合—如 linting 和映像的优化。通过使用设计时绑定和缩减，在应用程序的部署之前创建缩减的文件。绑定和在部署前贴图层提供减少的服务器负载的优点。但是，务必要识别该设计时绑定，缩减会增加生成复杂性，并且仅适用于静态文件。

配置绑定和缩减

MVC 和 Razor 页项目模板提供了 `bundleconfig.json` 配置文件用于定义每个捆绑包的选项。默认情况下，一个捆绑包配置定义的自定义 javascript (`wwwroot/js/site.js`) 和样式表 (`wwwroot/css/site.css`) 文件：

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

配置选项包括：

- `outputFileName`：要输出的捆绑文件名称。可以包含中的相对路径 `bundleconfig.json` 文件。**必填**
- `inputFiles`：要将捆绑在一起的文件的数组。这些是配置文件的相对路径。**可选**，* 空值会在空的输出文件。

组合支持模式。

- `minify`：输出类型缩减选项。可选，**默认值**- `minify: { enabled: true }`
 - 每个输出文件类型有配置选项。
 - [CSS Minifier](#)
 - [JavaScript Minifier](#)
 - [HTML Minifier](#)
- `includeInProject`：指示是否将生成的文件添加到项目文件的标志。可选，**默认-false**
- `sourceMap`：指示是否生成捆绑的文件的源映射的标志。可选，**默认-false**
- `sourceMapRootPath`：用于存储生成的源代码映射文件的根路径。

生成时执行的绑定和缩减

[BuildBundlerMinifier](#) NuGet 包启用的绑定执行并在生成时的缩减。包插入[MSBuild 目标](#)在生成和清理时间运行。

`Bundleconfig.json`文件分析由生成过程以生成基于定义的配置的输出文件。

注意

[BuildBundlerMinifier](#) 属于在为其 Microsoft 不提供支持的 GitHub 上的社区主导项目。应归档问题[此处](#)。

- [Visual Studio](#)
- [.NET Core CLI](#)

添加[BuildBundlerMinifier](#)包到你的项目。

生成项目。以下内容出现在输出窗口：

```
1>----- Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>
1>Bundler: Begin processing bundleconfig.json
1>  Minified wwwroot/css/site.min.css
1>  Minified wwwroot/js/site.min.js
1>Bundler: Done processing bundleconfig.json
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

清除该项目。以下内容出现在输出窗口：

```
1>----- Clean started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>
1>Bundler: Cleaning output from bundleconfig.json
1>Bundler: Done cleaning output file from bundleconfig.json
===== Clean: 1 succeeded, 0 failed, 0 skipped =====
```

绑定和缩减的即席执行

它是无法在临时上, 运行的绑定和缩减的任务, 而不生成项目。添加[BundlerMinifier.Core](#)到你的项目的 NuGet 包：

```
<DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.6.362" />
```

注意

[BundlerMinifier.Core](#) 属于在为其 Microsoft 不提供支持的 GitHub 上的社区主导项目。应归档问题[此处](#)。

此包扩展以包括.NET 核心 CLI `dotnet` 捆绑工具。在包管理器控制台 (PMC) 窗口中或在命令行界面，可以执行以下命令：

```
dotnet bundle
```

重要事项

NuGet 包管理器将依赖项添加到 *.csproj 文件作为 `<PackageReference />` 节点。`dotnet bundle` 命令注册.NET 核心 CLI 时，才 `<DotNetCliToolReference />` 使用节点。相应地修改 *.csproj 文件。

将文件添加到工作流

考虑在其中一个示例附加 `custom.css` 文件会添加与下面类似的：

```
.about, [role=main], [role=complementary] {
    margin-top: 60px;
}

footer {
    margin-top: 10px;
}
```

若要 minify `custom.css` 和捆绑其与 `site.css` 到 `site.min.css` 文件中，添加的相对路径 `bundleconfig.json`：

```
[
{
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
        "wwwroot/css/site.css",
        "wwwroot/css/custom.css"
    ]
},
{
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
        "wwwroot/js/site.js"
    ],
    "minify": {
        "enabled": true,
        "renameLocals": true
    },
    "sourceMap": false
}
]
```

注意

或者，可使用以下的组合模式：

```
"inputFiles": ["wwwroot/**/*(*.css|!(*.min.css))"]
```

此组合模式匹配所有 CSS 文件，并排除缩减的文件模式。

生成应用程序。打开 `site.min.css`，并注意的内容 `custom.css` 追加到文件末尾。

基于环境的绑定和缩减

作为最佳做法，应在生产环境中使用你的应用的捆绑和缩减型文件。在开发期间，为方便调试应用程序将使原始文件。

指定要通过使用在您的网页中包括哪些文件[环境标记帮助器](#)在您的视图。环境标记帮助器只呈现其内容，在特定运行时[环境](#)。

以下`environment`标记将呈现的未处理的 CSS 文件，在运行时`Development`环境：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
<environment include="Development">
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
  <link rel="stylesheet" href("~/css/site.css" />
</environment>
```

以下`environment`标记呈现捆绑和缩减型 CSS 文件，而不在环境中运行时`Development`。例如，在运行`Production`或`Staging`触发这些样式表的呈现：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
<environment exclude="Development">
  <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
        asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
        asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
        value="absolute" />
  <link rel="stylesheet" href "~/css/site.min.css" asp-append-version="true" />
</environment>
```

使用从 Gulp bundleconfig.json

在情况下在其中应用的绑定和缩减工作流需要额外的处理。示例包括映像优化、缓存清除功能，和 CDN 资产处理。若要满足这些要求，则可以将绑定和缩减工作流使用 Gulp。

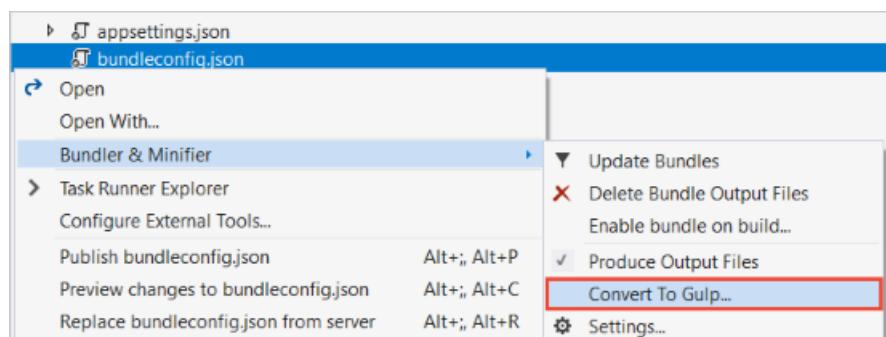
使用捆绑包 (&) Minifier 扩展

Visual Studio[捆绑包 \(&\) Minifier](#)扩展会转换为 Gulp。

注意

捆绑包 (&) Minifier 扩展属于在其 Microsoft 不提供支持的 GitHub 上的社区主导项目。应归档问题[此处](#)。

右键单击`bundleconfig.json`文件在解决方案资源管理器中，然后选择**捆绑包 (&) Minifier** > 转换到的 **Gulp...**：



*Gulpfile.js*和*package.json*文件添加到项目。支持npm中列出的包*package.json*文件的 `devDependencies` 部分安装。

为全局依赖项安装的 Gulp CLI PMC 窗口中运行以下命令：

```
npm i -g gulp-cli
```

*Gulpfile.js*文件读取*bundleconfig.json*输入、输出和设置的文件。

```
"use strict";

var gulp = require("gulp"),
concat = require("gulp-concat"),
cssmin = require("gulp-cssmin"),
htmlmin = require("gulp-htmlmin"),
uglify = require("gulp-uglify"),
merge = require("merge-stream"),
del = require("del"),
bundleconfig = require("./bundleconfig.json");

// Code omitted for brevity
```

手动转换

如果 Visual Studio 和/或捆绑包 (&) Minifier 扩展都不可用，将转换手动。

添加*package.json*文件中的，替换为以下 `devDependencies`，与项目根目录：

```
"devDependencies": {
  "del": "^3.0.0",
  "gulp": "^3.9.1",
  "gulp-concat": "^2.6.1",
  "gulp-cssmin": "^0.2.0",
  "gulp-htmlmin": "^3.0.0",
  "gulp-uglify": "^3.0.0",
  "merge-stream": "^1.0.1"
}
```

通过在相同的级别中运行以下命令安装依赖项*package.json*：

```
npm i
```

为全局依赖项安装的 Gulp CLI：

```
npm i -g gulp-cli
```

复制*gulpfile.js*到项目根目录下面文件：

```
"use strict";

var gulp = require("gulp"),
concat = require("gulp-concat"),
cssmin = require("gulp-cssmin"),
htmlmin = require("gulp-htmlmin"),
uglify = require("gulp-uglify"),
merge = require("merge-stream"),
del = require("del"),
bundleconfig = require("./bundleconfig.json");

var regex = {
```

```

css: /\.css$/,
html: /\.(html|htm)$/,
js: /\.js$/
};

gulp.task("min", ["min:js", "min:css", "min:html"]);

gulp.task("min:js", function () {
    var tasks = getBundles(regex.js).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(uglify())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
});

gulp.task("min:css", function () {
    var tasks = getBundles(regex.css).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(cssmin())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
});

gulp.task("min:html", function () {
    var tasks = getBundles(regex.html).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(htmlmin({ collapseWhitespace: true, minifyCSS: true, minifyJS: true }))
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
});

gulp.task("clean", function () {
    var files = bundleconfig.map(function (bundle) {
        return bundle.outputFileName;
    });

    return del(files);
});

gulp.task("watch", function () {
    getBundles(regex.js).forEach(function (bundle) {
        gulp.watch(bundle.inputFiles, ["min:js"]);
    });

    getBundles(regex.css).forEach(function (bundle) {
        gulp.watch(bundle.inputFiles, ["min:css"]);
    });

    getBundles(regex.html).forEach(function (bundle) {
        gulp.watch(bundle.inputFiles, ["min:html"]);
    });
});

function getBundles(regexPattern) {
    return bundleconfig.filter(function (bundle) {
        return regexPattern.test(bundle.outputFileName);
    });
}

```

运行 Gulp 任务

若要触发 Gulp 缩减任务之前生成 Visual Studio 中的项目，添加以下[MSBuild 目标](#)*.csproj 文件：

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>
```

在此示例中，在内定义的任何任务 `MyPreCompileTarget` 目标之前预定义运行 `Build` 目标。在 Visual Studio 输出窗口将显示类似于下面的输出：

```
1>----- Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU -----
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
1>[14:17:49] Using gulpfile C:\BuildBundlerMinifierApp\gulpfile.js
1>[14:17:49] Starting 'min:js'...
1>[14:17:49] Starting 'min:css'...
1>[14:17:49] Starting 'min:html'...
1>[14:17:49] Finished 'min:js' after 83 ms
1>[14:17:49] Finished 'min:css' after 88 ms
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

或者，Visual Studio 任务运行程序资源管理器可能用于将 Gulp 任务绑定到特定的 Visual Studio 事件。请参阅[正在运行的默认任务](#)有关执行此操作的说明。

其他资源

- [使用 Gulp](#)
- [使用 Grunt](#)
- [使用多个环境](#)
- [标记帮助程序](#)

在 ASP.NET 核心中的浏览器链接

2018/5/14 • 4 min to read • [Edit Online](#)

通过Nicolò Carandini, Mike Wasson, 和Tom Dykstra

浏览器链接是创建开发环境和一个或多个 web 浏览器之间的通信通道的 Visual Studio 中的功能。你可以使用浏览器链接刷新 web 应用程序在多个浏览器中的，这是适用于跨浏览器测试。

浏览器链接安装程序

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

ASP.NET 核心 2.x **Web 应用程序**, 空, 和**Web API**模板项目使用[Microsoft.AspNetCore.All](#)元包，其中包含的包引用[Microsoft.VisualStudio.Web.BrowserLink](#)。因此，使用[Microsoft.AspNetCore.All](#)元包无需任何进一步的操作，以使浏览器链接可供使用。

配置

在 `Configure` 方法 `Startup.cs` 文件：

```
app.UseBrowserLink();
```

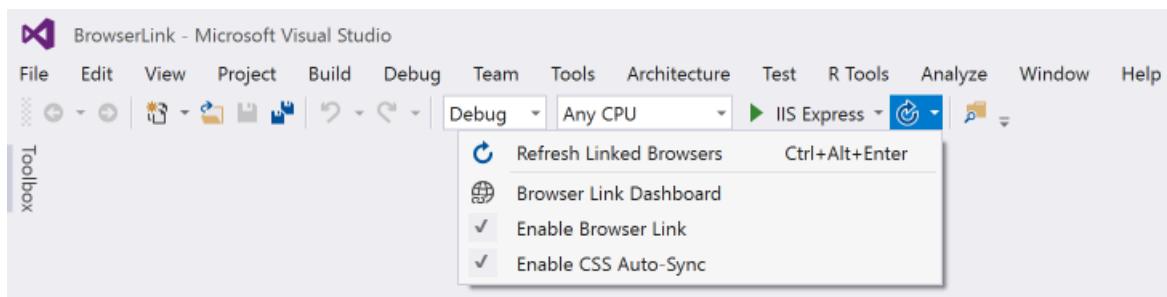
代码通常位于 `if` 块只在开发环境中，启用浏览器链接，如下所示：

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

有关详细信息，请参阅[使用多个环境](#)。

如何使用浏览器链接

如果你拥有打开 ASP.NET Core 项目，Visual Studio 将旁边显示浏览器链接工具栏控件调试目标工具栏控件：



通过浏览器链接工具栏控件，你可以：

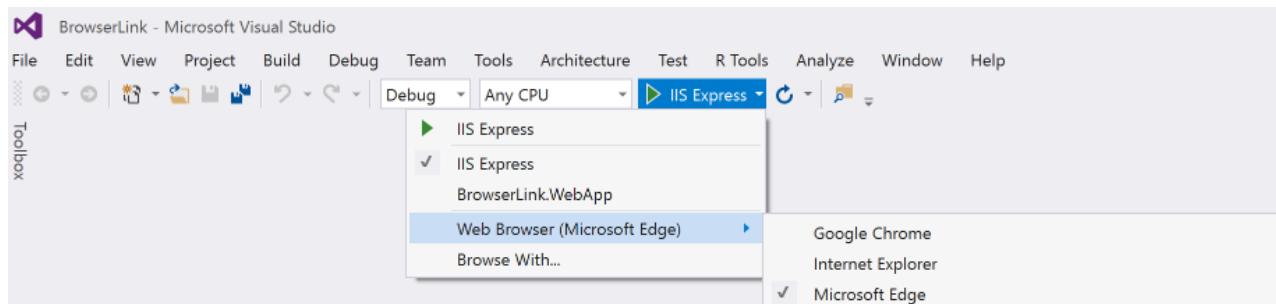
- 刷新 web 应用程序在多个浏览器中的一次。
- 打开浏览器链接仪表板。
- 启用或禁用**Browser Link**。注意：默认情况下，Visual Studio 2017 (15.3) 中禁用浏览器链接。
- 启用或禁用**CSS 自动同步**。

注意

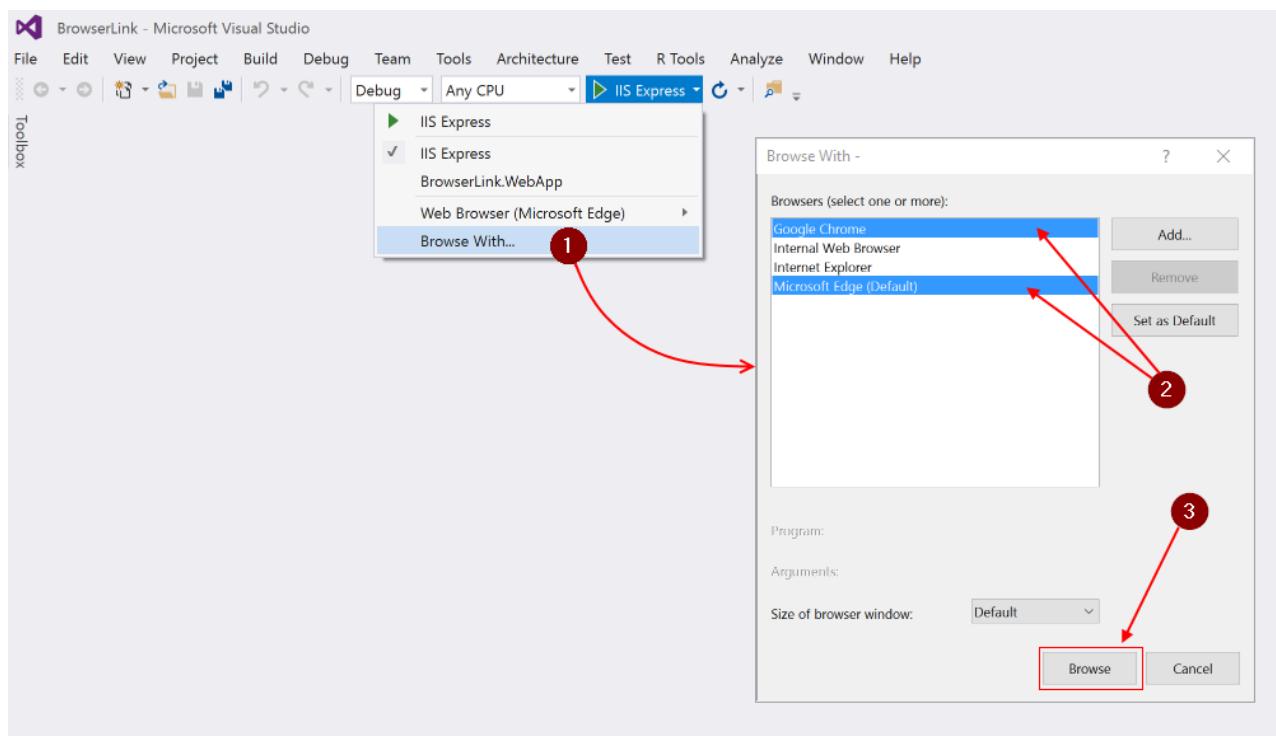
某些 Visual Studio 插件，最值得注意的是 *Web 扩展包 2015 年* 和 *Web 扩展包 2017 年*，提供扩展的功能的浏览器链接，但某些其他功能不与 ASP 配合使用。NET 核心项目。

刷新 web 应用程序在多个浏览器中的一次

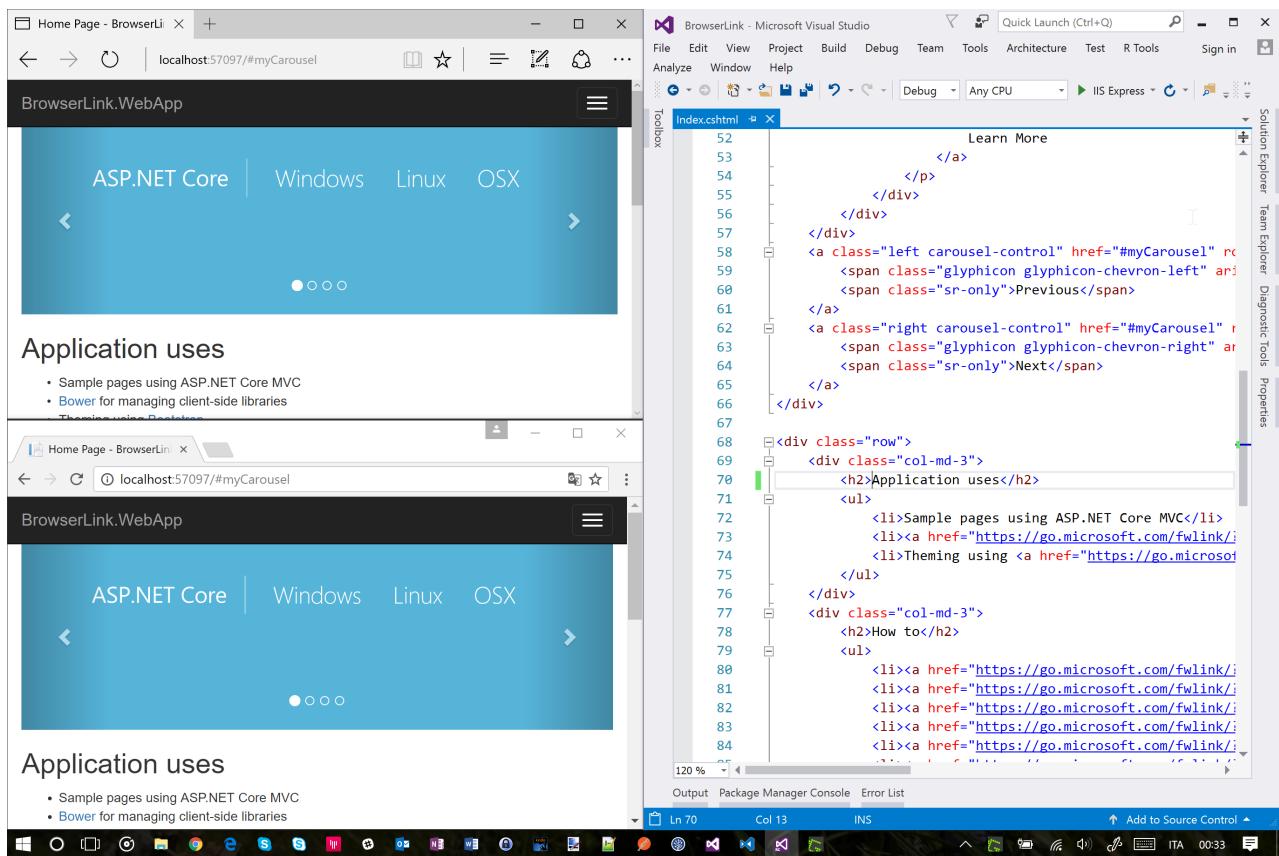
若要选择单个 web 浏览器以启动启动项目时，使用中的下拉列表菜单调试目标工具栏控件：



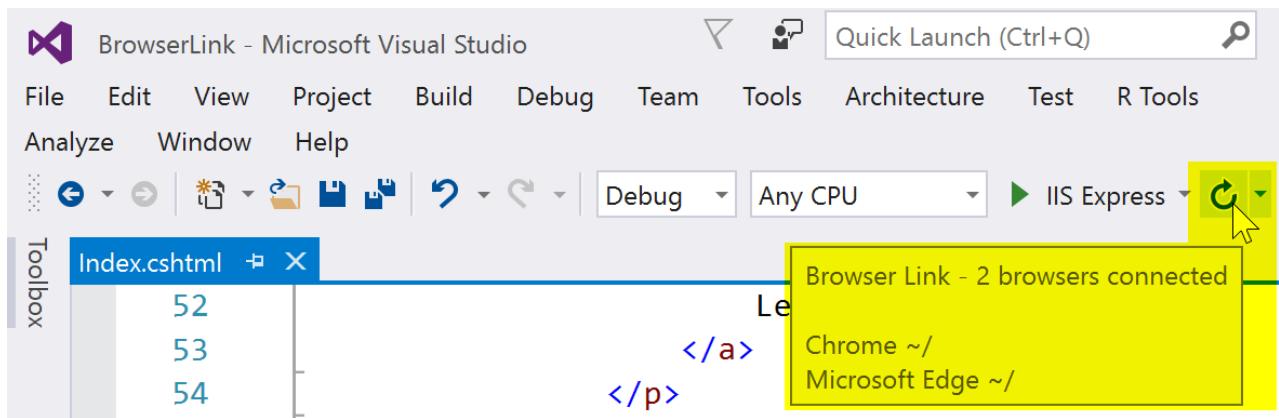
若要同时打开多个浏览器，选择浏览与... 从相同的下拉列表。按住 CTRL 键以选择所需的浏览器，然后单击浏览：



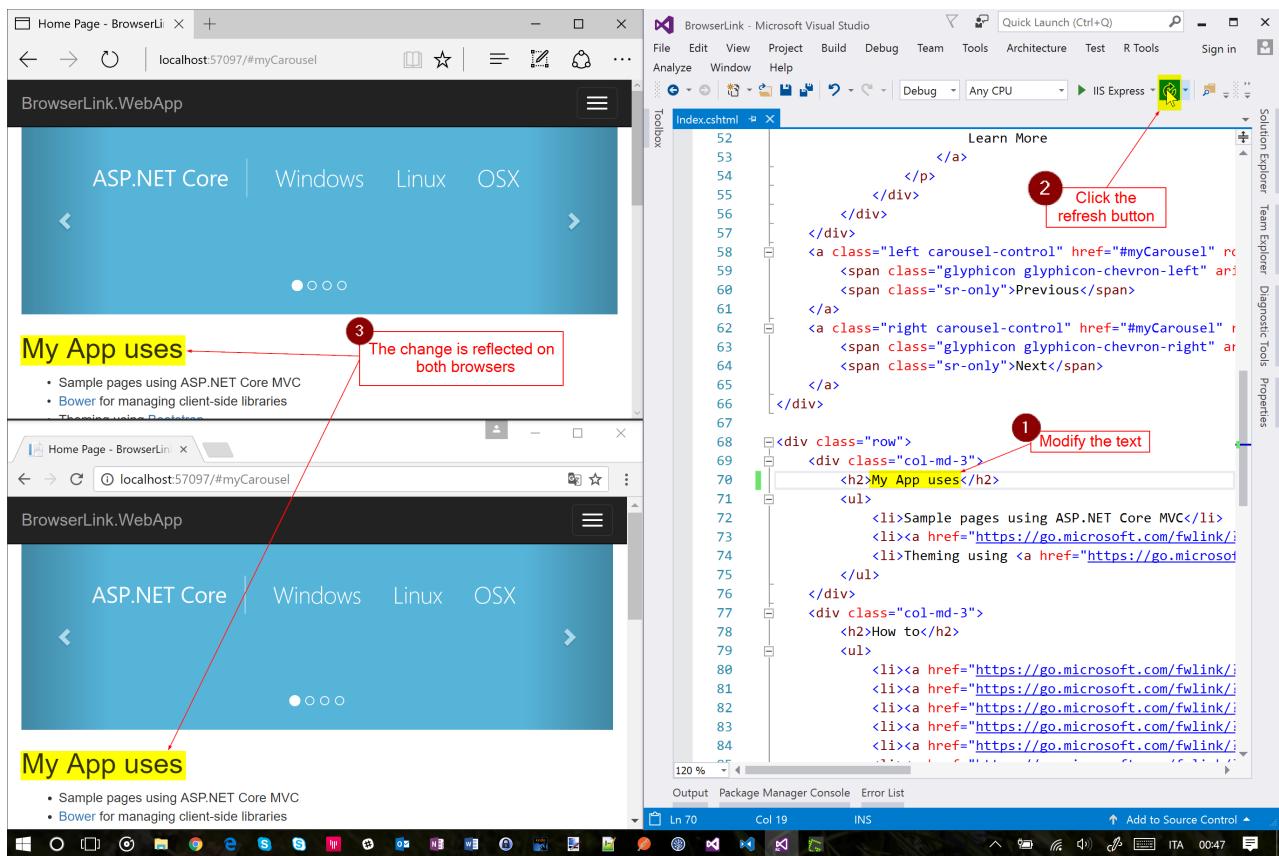
下面是 Visual Studio 显示与索引视图打开的屏幕快照以及两个打开的浏览器：



将鼠标悬停在该浏览器链接工具栏控件，以查看连接到的项目的浏览器：



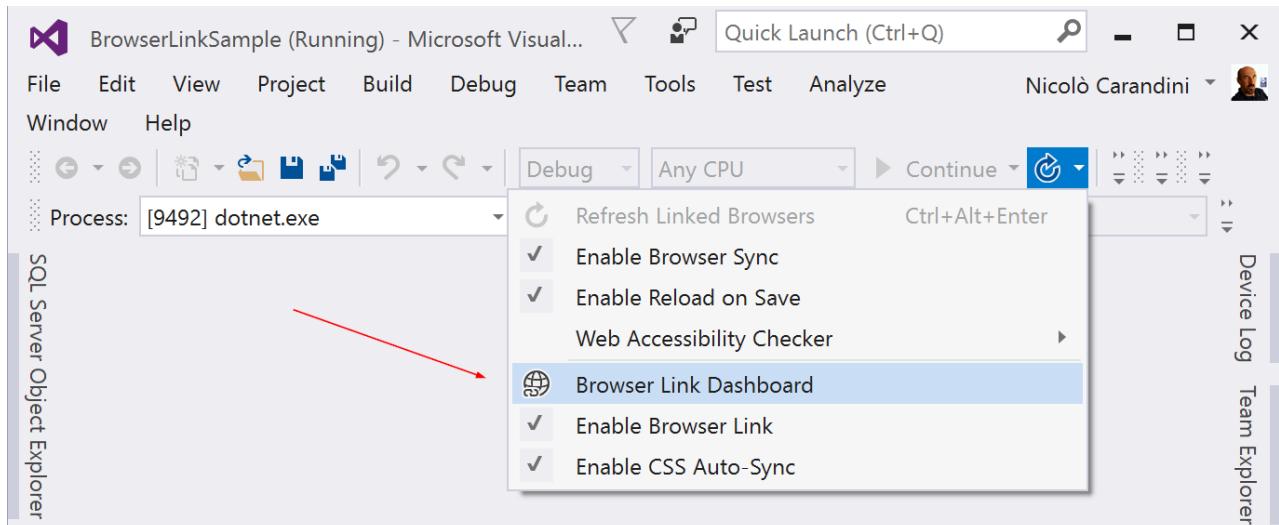
更改索引视图中，并单击浏览器链接刷新按钮后，将更新所有连接的浏览器：



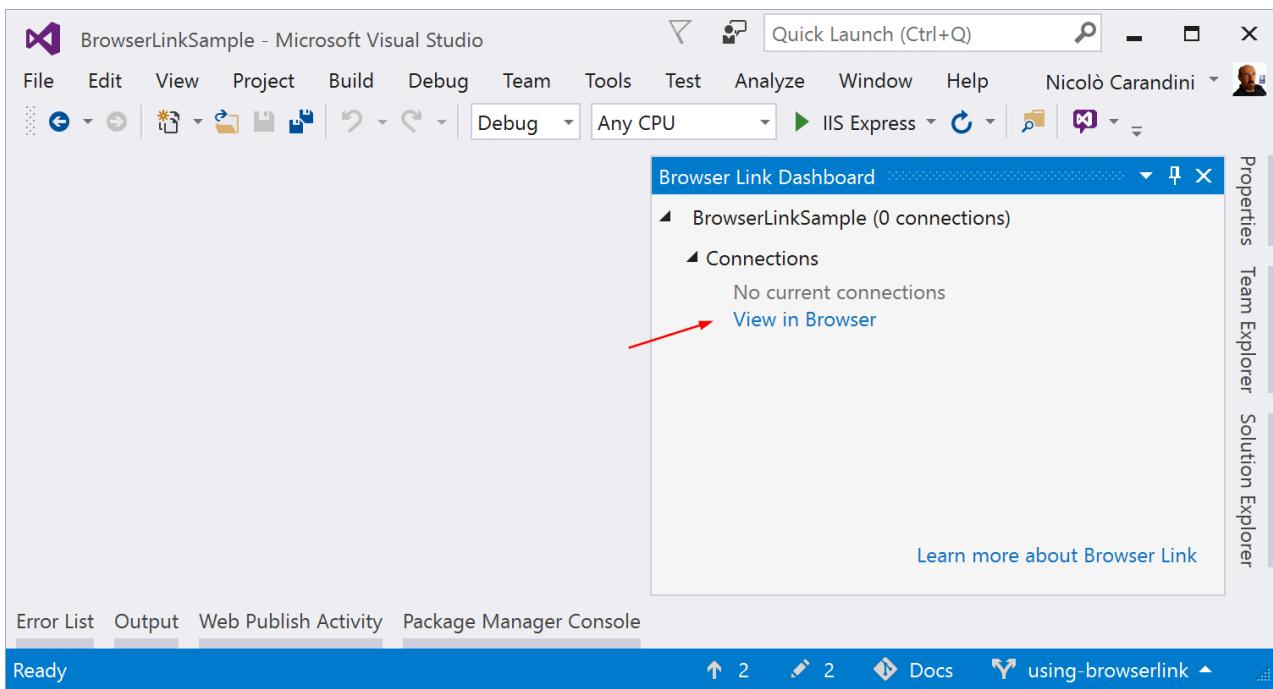
浏览器链接也适用于浏览器，从 Visual Studio 外部启动并导航到应用程序 URL。

浏览器链接仪表板

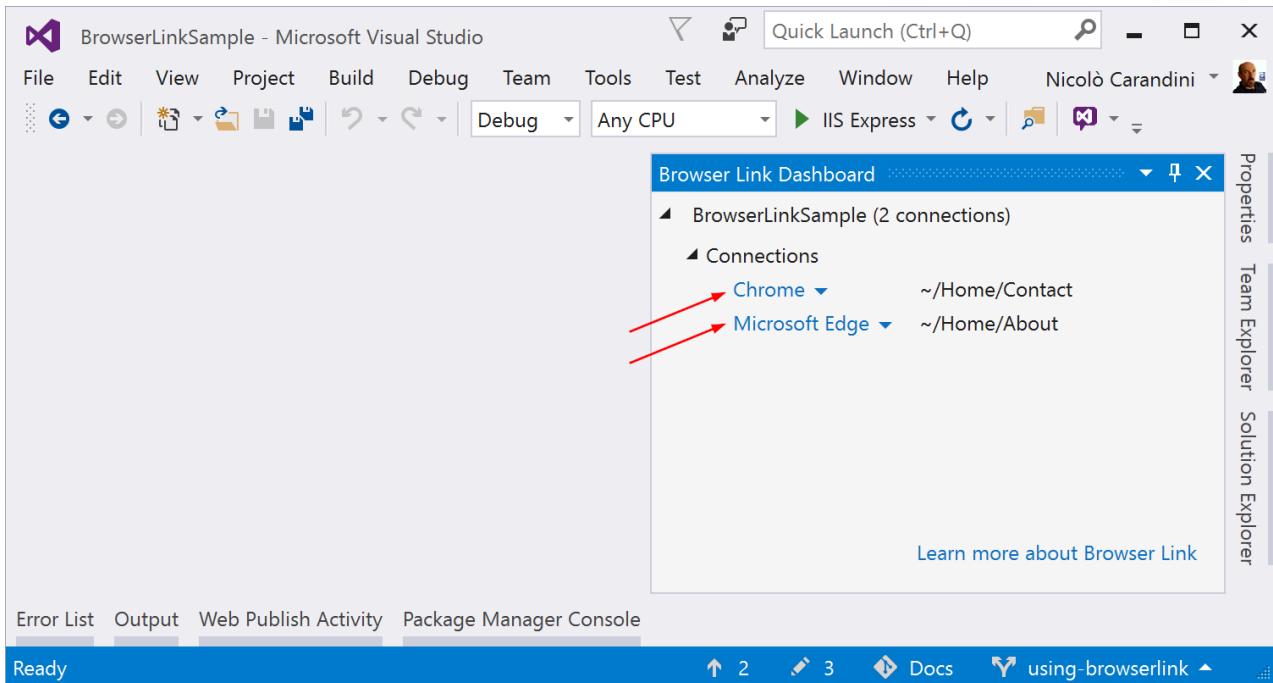
从浏览器链接下拉菜单来管理与打开的浏览器的连接打开浏览器链接仪表板：



如果连接没有浏览器，您可以通过选择启动非调试会话用浏览器查看链接：



否则，连接的浏览器在显示时带有显示每个浏览器的页面的路径：



如果您愿意，你可以单击要刷新该单个浏览器的列出的浏览器名称。

启用或禁用浏览器链接

当重新启用 Browser Link 禁用它之后时，您必须刷新浏览器以将它们重新连接。

启用或禁用 CSS 自动同步

启用 CSS 自动同步后，对 CSS 文件进行任何更改时自动刷新连接的浏览器。

Application Insights 如何工作？

浏览器链接使用 SignalR 来创建 Visual Studio 和浏览器之间的通信通道。启用浏览器链接后，Visual Studio 将充当多个客户端（浏览器）可以连接到的 SignalR 服务器。浏览器链接也在 ASP.NET 请求管道中注册的中间件组件。此组件将插入特殊 `<script>` 到每个页请求从服务器的引用。您可以通过选择查看脚本引用查看源在浏览器和左边边缘滚动 `<body>` 标记内容：

```
<!-- Visual Studio Browser Link -->
<script type="application/json" id="__browserLink_initializationData">
    {"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":false}
</script>
<script type="text/javascript" src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
<!-- End Browser Link -->
</body>
```

不修改你的源文件。中间件组件动态插入脚本引用。

由于浏览器端代码是所有 JavaScript，则它将适用于 SignalR 支持而无需浏览器插件的所有浏览器。

JavaScriptServices 用于在 ASP.NET Core 中创建单页面应用程序

2018/4/27 • 14 min to read • [Edit Online](#)

通过[Scott Addie](#)和[Fiyaz Hasan](#)

单页面应用程序 (SPA) 是一种流行的 web 应用程序，因为其固有的丰富用户体验。将客户端 SPA 框架或库，如集成角或做出反应，与服务器端框架，如 ASP.NET Core 可能很困难。[JavaScriptServices](#)开发的目的是减少在集成过程中的问题。它可让在不同的客户端和服务器技术堆栈之间的无缝操作。

[查看或下载示例代码 \(如何下载\)](#)

什么是 JavaScriptServices？

JavaScriptServices 是 ASP.NET Core 的客户端技术的集合。其目标是将 ASP.NET Core 定位为开发人员的首选服务器端平台，用于构建 Spas。

JavaScriptServices 包含三个不同的 NuGet 包：

- [Microsoft.AspNetCore.NodeServices](#) (NodeServices)
- [Microsoft.AspNetCore.SpaServices](#) (SpaServices)
- [Microsoft.AspNetCore.SpaTemplates](#) (SpaTemplates)

这些包的作用是如果你：

- 在服务器上运行 JavaScript
- 使用 SPA 框架或库
- 生成客户端 Webpack 资产

在本文中将焦点大部分位于使用 SpaServices 包。

什么是 SpaServices？

用于将 ASP.NET Core 定位为开发人员的首选服务器端平台，用于构建 Spas，SpaServices 而创建。SpaServices 不需要开发与 ASP.NET 核心的 Spas，它不会将您限制在一个特定的客户端框架。

SpaServices 提供有用的基础结构，如所示：

- [服务器端预呈现](#)
- [Webpack 开发人员中间件](#)
- [热模块更换](#)
- [路由的帮助器](#)

总体来说，这些基础结构组件来提高开发工作流和运行时体验。可以单独采用组件。

使用 SpaServices 的先决条件

若要使用 SpaServices，安装以下项：

- [Nodejs](#) (6 或更高版本) 与 npm
 - 若要验证这些组件安装，并找不到，运行以下命令从命令行：

```
node -v && npm -v
```

注意：如果你要部署到 Azure 网站，你不需要此处执行任何操作—Nodejs 已安装并且可用的服务器环境中。

- [.NET Core SDK 2.0 or later](#)
 - 如果你使用 Visual Studio 2017 在 Windows 上，通过选择安装 SDK [.NET 核心跨平台开发工作负载](#)。
- [Microsoft.AspNetCore.SpaServices](#) NuGet 包

服务器端预呈现

通用的（也称为 isomorphic）应用程序是能够在服务器和客户端上同时运行的 JavaScript 应用程序。角、响应和其他常用框架提供一个通用平台此应用程序的开发风格。目的是第一次呈现 Nodejs，通过在服务器上的 framework 组件，然后将进一步委托到客户端执行。

ASP.NET 核心[标记帮助程序](#)由 SpaServices 简化通过调用服务器上的 JavaScript 函数的服务器端预呈现的实现。

系统必备

安装以下组件：

- [aspnet 预呈现](#)npm 包：

```
npm i -S aspnet-prerendering
```

配置

标记帮助程序都在项目的命名空间注册通过可发现 `_ViewImports.cshtml` 文件：

```
@using SpaServicesSampleApp  
@addTagHelper "*", Microsoft.AspNetCore.Mvc.TagHelpers"  
@addTagHelper "*", Microsoft.AspNetCore.SpaServices"
```

这些标记帮助程序抽象化通过利用在 Razor 视图类似于 HTML 的语法直接与低级别 API 进行通信的复杂性：

```
<app asp-prerender-module="ClientApp/dist/main-server">Loading...</app>
```

asp-prerender-module 标记帮助器

`asp-prerender-module` 标记帮助器，使用在前面的代码示例中，执行 `ClientApp/dist/main-server.js` 通过 Nodejs 服务器上。为清晰起见，`main-server.js` 文件是 TypeScript JavaScript transpilation 任务中的项目 [Webpack](#) 生成过程。Webpack 定义的入口点别名 `main-server`；并且，在开始此别名的依赖项关系图的遍历 `ClientApp/启动 server.ts` 文件：

```
entry: { 'main-server': './ClientApp/boot-server.ts' },
```

在以下的角度示例中，`ClientApp/启动 server.ts` 文件利用 `createServerRenderer` 函数和 `RenderResult` 类型 `aspnet-prerendering` npm 包以配置通过 Nodejs 服务器呈现。发送到服务器端呈现传递给解析函数调用，从而将包装在强类型化的 JavaScript 中的 HTML 标记 `Promise` 对象。`Promise` 对象的基数是它以异步方式提供到 DOM 的占位符元素中注入的页的 HTML 标记。

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: state.renderToString()
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

asp-prerender-data 标记帮助器

结合了 `asp-prerender-module` 标记帮助器, `asp-prerender-data` 标记帮助器可以用于将上下文信息从 Razor 视图传递到服务器端 JavaScript。例如, 以下标记将传递到的用户数据 `main-server` 模块:

```
<app asp-prerender-module="ClientApp/dist/main-server"
      asp-prerender-data='new {
          UserName = "John Doe"
      }'>Loading...</app>
```

接收 `UserName` 自变量使用内置的 JSON 序列化程序序列化和存储在 `params.data` 对象。在以下的角度示例中, 数据用于构造中的个性化的问候语 `h1` 元素:

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            const result = `<h1>Hello, ${params.data.userName}</h1>`;

            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: result
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

注意：在标记帮助程序中传递的属性名称使用表示**PascalCase**表示法。相比之下，到 JavaScript，相同的属性名称由驼峰匹配。默认 JSON 序列化配置负责这种差异。

若要展开在前面的代码示例时，数据可以从服务器由传递给视图 hydrating `globals` 属性提供给 `resolve` 函数：

```

import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
  const providers = [
    { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
    { provide: 'ORIGIN_URL', useValue: params.origin }
  ];

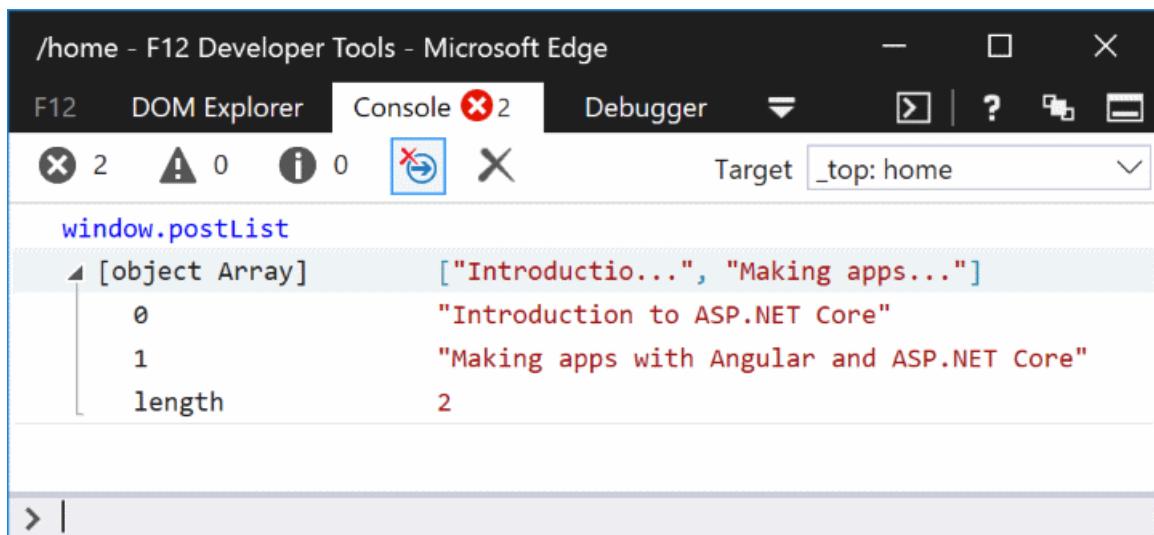
  return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
    const appRef = moduleRef.injector.get(ApplicationRef);
    const state = moduleRef.injector.get(PlatformState);
    const zone = moduleRef.injector.get(NgZone);

    return new Promise<RenderResult>((resolve, reject) => {
      const result = `<h1>Hello, ${params.data.userName}</h1>`;

      zone.onError.subscribe(errorInfo => reject(errorInfo));
      appRef.isStable.first(isStable => isStable).subscribe(() => {
        // Because 'onStable' fires before 'onError', we have to delay slightly before
        // completing the request in case there's an error to report
        setImmediate(() => {
          resolve({
            html: result,
            globals: {
              postList: [
                'Introduction to ASP.NET Core',
                'Making apps with Angular and ASP.NET Core'
              ]
            }
          });
        });
      });
    });
  });
});

```

`window.postList` 数组内部定义 `globals` 对象附加到浏览器的全局 `window` 对象。为全局作用域此变量提升消除重复的工作量，尤其当它与加载一次在服务器上，再次在客户端上相同的数据。



Webpack 开发人员中间件

[Webpack 开发人员中间件](#) 引入了 Webpack 按需生成资源的凭此简化的开发工作流。该中间件自动编译，并在浏览器中重新加载页面时提供客户端资源。一种替代方法是手动将 Webpack 通过项目的 npm 生成脚本调用，第三方依赖关系或自定义代码更改时。Npm 中生成脚本 `package.json` 文件显示在下面的示例：

```
"build": "npm run build:vendor && npm run build:custom",
```

系统必备

安装以下组件：

- [aspnet webpack](#) npm 包：

```
npm i -D aspnet-webpack
```

配置

Webpack 开发人员中间件注册到中的以下代码通过 HTTP 请求管道 `Startup.cs` 文件的 `Configure` 方法：

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseWebpackDevMiddleware();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Call UseWebpackDevMiddleware before UseStaticFiles
app.UseStaticFiles();
```

`UseWebpackDevMiddleware` 前，必须调用扩展方法[注册静态文件承载](#)通过 `UseStaticFiles` 扩展方法。出于安全原因，注册该中间件，仅当应用程序在开发模式下运行时。

`Webpack.config.js` 文件的 `output.publicPath` 属性告知要监视的中间件 `dist` 更改的文件夹：

```
module.exports = (env) => {
    output: {
        filename: '[name].js',
        publicPath: '/dist/' // Webpack dev middleware, if enabled, handles requests for this URL prefix
    },
}
```

热模块替换

思考的 Webpack 的[热模块替换\(HMR\)](#)功能作为演变而来的[Webpack 开发人员中间件](#)。HMR 引入了完全相同的好处，但它进一步，从而简化了开发工作流自动编译所做的更改后更新页面内容。不要混淆这与刷新浏览器中，这会干扰的当前内存中状态和 SPA 的调试会话。没有 Webpack 开发人员中间件服务与浏览器中，这意味着更改推送到浏览器之间的实时链接。

系统必备

安装以下组件：

- [webpack 热 middleware](#) npm 包：

```
npm i -D webpack-hot-middleware
```

配置

HMR 组件必须注册到 MVC 的 HTTP 请求管道中 `Configure` 方法：

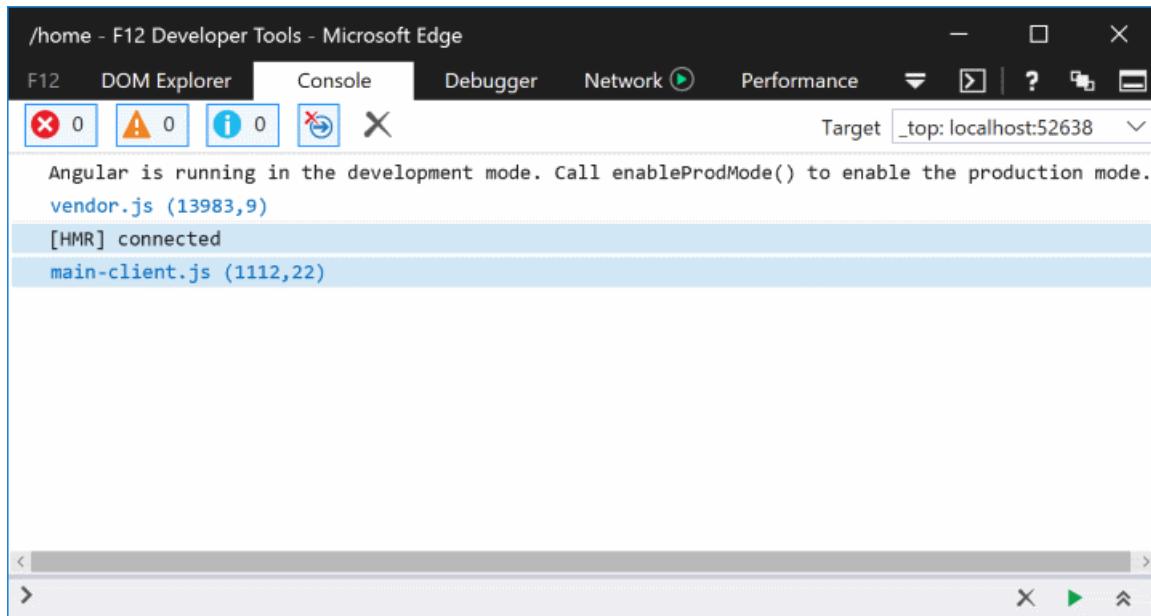
```
app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
    HotModuleReplacement = true
});
```

为已对如此[Webpack 开发人员中间件](#)、`UseWebpackDevMiddleware` 前，必须调用扩展方法 `UseStaticFiles` 扩展方法。出于安全原因，注册该中间件，仅当应用程序在开发模式下运行时。

`Webpack.config.js`文件必须定义 `plugins`，即使它保留为空数组：

```
module.exports = (env) => {
    plugins: [new CheckerPlugin()]
```

在加载浏览器中的应用程序之后，的开发人员工具的控制台选项卡提供 HMR 激活的确认：



路由的帮助器

在大多数基于 ASP.NET Core 的 Spa，您需要客户端路由除了服务器端路由。SPA 和 MVC 路由系统可以独立处理不受干扰。没有，但是，一个边缘案例是否会造造成面临的挑战：标识 404 HTTP 响应。

请考虑在该方案中的无扩展名路由 `/some/page` 使用。假定该请求不模式匹配的服务器端路由，但其模式匹配的客户端路由。现在请考虑对的传入请求 `/images/user-512.png`，它通常需要查找服务器上的图像文件。如果该请求的资源路径不匹配任何服务器端路由或静态文件，它不太客户端应用程序将处理它，你通常想要返回 HTTP 状态代码为 404。

系统必备

安装以下组件：

- 客户端路由 npm 包。使用角作为示例：

```
npm i -S @angular/router
```

配置

名为的扩展方法 `MapSpaFallbackRoute` 中使用 `Configure` 方法：

```

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

    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});

```

提示：路由中配置它们的顺序进行计算。因此，`default` 模式匹配的第一次使用在前面的代码示例中的路由。

创建新项目

JavaScriptServices 提供了预配置的应用程序模板。SpaServices 中这些模板，与不同的框架和库如角、响应和 Redux 结合使用。

可以通过.NET 核心 CLI 安装这些模板，通过运行以下命令：

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates::*
```

显示可用的 SPA 模板的列表：

模板	短名称	语言	TAGS
带有角度的 MVC ASP.NET 核心	angular	[C#]	Web/MVC/SPA
带有 React.js 的 MVC ASP.NET 核心	react	[C#]	Web/MVC/SPA
MVC ASP.NET Core React.js 和回顾	reactredux	[C#]	Web/MVC/SPA

若要创建新项目使用 SPA 模板之一时，包含短名称中的模板的[dotnet 新](#)命令。以下命令将创建与 ASP.NET 核心 MVC 配置为在服务器端角度的应用程序：

```
dotnet new angular
```

设置运行时配置模式

存在两个主运行时配置模式：

- **开发：**
 - 包括源映射，以便于调试。
 - 不优化性能的客户端代码。
- **生产：**
 - 排除源映射。
 - 优化通过绑定和缩减的客户端代码。

ASP.NET 核心使用名为的环境变量 `ASPNETCORE_ENVIRONMENT` 来存储配置模式。请参阅**[将环境设置](#)** 有关详细信息。

使用.NET Core CLI 运行

在项目根目录中运行以下命令，以还原所需的 NuGet 和 npm 包：

```
dotnet restore && npm i
```

生成并运行应用程序：

```
dotnet run
```

在根据本地主机上的应用程序启动[运行时配置模式下](#)。导航到 `http://localhost:5000` 浏览器中显示登录页。

使用 Visual Studio 2017 运行

打开 `.csproj` 生成文件 [dotnet 新](#) 命令。在项目打开时自动还原所需的 NuGet 和 npm 包。此还原过程可能需要几分钟时间，并已准备好它完成时要运行的应用程序。单击绿色运行的按钮或按 `ctrl + F5`，应用程序的登录页将打开浏览器。在根据本地主机上运行应用程序[运行时配置模式下](#)。

测试应用程序

SpaServices 模板是预配置为运行客户端测试使用 [Karma](#) 和 [Jasmine](#)。Jasmine 是常用于单元测试框架 JavaScript，而 Karma 是这些测试的测试运行程序。Karma 配置为使用 [Webpack 开发人员中间件](#) 以便开发人员不需要停止并运行测试，每次进行更改。无论是针对测试用例或测试用例本身运行的代码，则测试将自动运行。

使用作为示例的角度的应用程序，已为提供两个 Jasmine 测试用例 `CounterComponent` 中 `counter.component.spec.ts` 文件：

```
it('should display a title', async(() => {
  const titleText = fixture.nativeElement.querySelector('h1').textContent;
  expect(titleText).toEqual('Counter');
}));

it('should start with count 0, then increments by 1 when clicked', async(() => {
  const countElement = fixture.nativeElement.querySelector('strong');
  expect(countElement.textContent).toEqual('0');

  const incrementButton = fixture.nativeElement.querySelector('button');
  incrementButton.click();
  fixture.detectChanges();
  expect(countElement.textContent).toEqual('1');
}));
```

打开命令提示符中 `ClientApp` 目录。运行下面的命令：

```
npm test
```

该脚本将启动 Karma 测试运行程序，读取中定义的设置 `karma.conf.js` 文件。除了其他设置以外，`karma.conf.js` 标识要执行通过的测试文件其 `files` 数组：

```
module.exports = function (config) {
  config.set({
    files: [
      '../../wwwroot/dist/vendor.js',
      './boot-tests.ts'
    ],
  },
```

发布应用程序

将生成的客户端资产和已发布的 ASP.NET Core 项目合并为准备就绪，可以部署包可能会很麻烦。SpaServices 幸运的是，具有名为自定义 MSBuild 目标安排该整个发布进程 `RunWebpack`：

```
<Target Name="RunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config webpack.config.vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**; ClientApp\dist\**" />
    <ResolvedFileToPublish Include="@{DistFiles->'%(FullPath)'}" Exclude="@{ResolvedFileToPublish}">
      <RelativePath>%{DistFiles.Identity}</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>
```

MSBuild 目标具有以下职责：

1. 还原 npm 包
2. 创建第三方、客户端资产的生产级版本
3. 创建自定义客户端资产的生产级版本
4. 将 Webpack 生成资产复制到发布文件夹

运行时，将调用 MSBuild 目标：

```
dotnet publish -c Release
```

其他资源

- [角度文档](#)

通过 ASP.NET Core 使用单页应用程序模板

2018/4/10 • 1 min to read • [Edit Online](#)

注意

已发布的 .NET Core 2.0.x SDK 包括 Angular、React 以及带 Redux 的 React 早期项目模板。本文档并不涉及这些早期项目模板。本文档适用于最新的 Angular、React 以及带 Redux 的 React 模板，这些模板可手动安装到 ASP.NET Core 2.0 中。ASP.NET Core 2.1 中默认包含这些模板。

系统必备

- [.NET Core SDK 2.0 or later](#)
- [Node.js](#) 版本 6 或更高版本

安装

如果拥有 ASP.NET Core 2.0，请运行以下命令，安装 Angular、React 以及带 Redux 的 React 更新 ASP.NET Core 模板：

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

使用模板

- [使用 Angular 项目模板](#)
- [使用 React 项目模板](#)
- [使用带 Redux 的 React 项目模板](#)

使用 ASP.NET Core 角度项目模板

2018/5/17 • 13 min to read • [Edit Online](#)

注意

本文档不有关角度项目模板包括在 ASP.NET 核心 2.0。它是有关与其则可手动更新较新角度模板。默认情况下，该模板包含在 ASP.NET 核心 2.1。

更新的角度项目模板提供了 ASP.NET Core 应用使用角和角速度 CLI 可以实现的丰富的客户端用户界面 (UI) 的方便的起始点。

该模板是等效于创建充当 API 后端的 ASP.NET Core 项目和一个角度 CLI 项目，以充当用户界面。模板提供承载这两种项目类型中的单个应用程序项目的便利性。因此，可以生成应用程序项目并将其发布为单个单元。

创建新的应用程序

如果使用 ASP.NET 核心 2.0，请确保你已[安装更新的角度项目模板](#)。如果你有 ASP.NET 核心 2.1，则不需要安装它。

从命令提示符下使用命令创建新项目 `dotnet new angular` 空的目录中。例如，以下命令创建的应用程序中我新应用目录并切换到该目录：

```
dotnet new angular -o my-new-app  
cd my-new-app
```

从 Visual Studio 或.NET Core CLI 运行应用程序：

- [Visual Studio](#)
- [.NET Core CLI](#)

打开生成 `.csproj` 文件中，并从那里运行正常的应用程序。

生成过程还原首次运行，可能需要几分钟的 npm 依赖关系。后续的生成处于快得多。

项目模板创建 ASP.NET Core 应用和角速度应用。ASP.NET Core 应用旨在用于数据访问、授权和其他服务器端问题。角度的应用程序，驻留在 `ClientApp` 子目录，旨在用于所有 UI 问题。

添加页面、映像、样式、模块、等。

`ClientApp` 目录包含标准的角度 CLI 应用程序。请参阅官方[角度文档](#)有关详细信息。

略有差异由此模板创建的角度的应用程序和创建角度 cli 本身之间（通过 `ng new`）；但是，应用程序的功能保持不变。由模板创建此应用程序包含 [Bootstrap](#)-基于布局和一个基本的路由示例。

运行 ng 命令

在命令提示符下，切换到 `ClientApp` 子目录：

```
cd ClientApp
```

如果你有 `ng` 全局安装的工具，你可以运行任何其命令。例如，你可以运行 `ng lint`，`ng test`，或任何其他角度

CLI 命令。没有无需运行 `ng serve` 不过, 因为 ASP.NET Core 应用程序处理为服务器端和客户端部分你的应用程序提供服务。在内部, 它使用 `ng serve` 开发中。

如果你没有 `ng` 工具安装, 运行 `npm run ng` 相反。例如, 你可以运行 `npm run ng lint` 或 `npm run ng test`。

安装 npm 包

若要安装第三方 npm 包, 使用命令提示符处, `ClientApp` 子目录。例如:

```
cd ClientApp
npm install --save <package_name>
```

发布和部署

在开发中, 应用将在优化为开发人员方便起见模式下运行。例如, JavaScript 捆绑包包括源地图 (以便在调试时, 你可以看到原始 TypeScript 代码)。应用程序监视的磁盘上的 TypeScript、HTML 和 CSS 文件更改自动重新编译并重新加载当它发现更改这些文件。

在生产中, 提供您针对性能进行了优化的应用程序的版本。这被配置为自动发生这种情况。发布时, 生成配置发出缩减, 预时间的 (AoT) 编译生成的客户端代码。与开发内部版本, 不同生产生成不需要在服务器上安装的 Node.js (除非已启用 [服务器端预呈现](#))。

您可以使用标准 [ASP.NET Core 托管和部署方法](#)。

独立运行"ng 提供"

将项目配置为在开发模式下启动 ASP.NET Core 应用时在后台启动自己的角度 CLI 服务器实例。这很方便, 因为无需手动运行单独的服务器。

没有此默认设置的缺陷。每次修改你的 C# 代码和应用程序需要重新启动, 将 ASP.NET 核心角度 CLI 服务器重新启动。启动备份需要大约为 10 秒钟。如果您正在进行频繁的 C# 代码编辑, 并且不希望等待重新启动的角度 CLI, 运行角度 CLI 服务器外部, 独立于 ASP.NET 核心进程。若要这样做:

1. 在命令提示符下, 切换到 `ClientApp` 子目录, 并启动角度 CLI 开发服务器:

```
cd ClientApp
npm start
```

重要事项

使用 `npm start` 不启动的角度 CLI 开发服务器 `ng serve`, 以便中的配置 `package.json` 遵守。若要将附加参数传递到角度 CLI 服务器, 请将它们添加到相关 `scripts` 行中你 `package.json` 文件。

2. 修改 ASP.NET Core 应用程序而不是启动其自身的一个使用外部的角度 CLI 实例。在你启动类中, 替换 `spa.UseAngularCliServer` 替换为以下调用:

```
spa.UseProxyToSpaDevelopmentServer("http://localhost:4200");
```

当你启动 ASP.NET Core 应用程序时, 它不会启动角度 CLI 服务器。将改为使用手动启动实例。这使它能够启动并重新启动速度更快。它不再等待角度 CLI 来每次重新生成客户端应用。

服务器端呈现

作为一种性能功能，你可以选择预呈现上服务器以及运行客户端上的应用程序角度。这意味着浏览器接收表示您的应用程序的初始用户界面，使其能够显示它即使在下载并执行 JavaScript 捆绑包之前的 HTML 标记。大部分此操作的实现调用角度功能带来[角度通用](#)。

提示

启用服务器端呈现 (SSR) 引入了大量的额外复杂性同时在开发和部署过程。读取[SSR 缺点](#)确定 SSR 是否适合你的要求。

若要启用 SSR，你需要进行大量添加到你的项目。

在启动类，[后配置行](#) `spa.Options.SourcePath`，和[之前](#)调用 `UseAngularCliServer` 或 `UseProxyToSpaDevelopmentServer`，添加以下：

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "ClientApp";

    spa.UseSpaPrerendering(options =>
    {
        options.BootModulePath = `${spa.Options.SourcePath}/dist-server/main.bundle.js`;
        options.BootModuleBuilder = env.IsDevelopment()
            ? new AngularCliBuilder(npmScript: "build:ssr")
            : null;
        options.ExcludeUrls = new[] { "/sockjs-node" };
    });

    if (env.IsDevelopment())
    {
        spa.UseAngularCliServer(npmScript: "start");
    }
});
```

在开发模式下，此代码将尝试通过运行脚本创建 SSR 捆绑 `build:ssr` 中，定义 `ClientApp/package.json`。这将生成名为的角度应用 `ssr`，这并不尚未定义。

在结束 `apps` 数组中 `ClientApp/angular-clijson`，定义具有名称的额外应用 `ssr`。使用以下选项：

```
{
    "name": "ssr",
    "root": "src",
    "outDir": "dist-server",
    "assets": [
        "assets"
    ],
    "main": "main.server.ts",
    "tsconfig": "tsconfig.server.json",
    "prefix": "app",
    "scripts": [],
    "environmentSource": "environments/environment.ts",
    "environments": {
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
    },
    "platform": "server"
}
```

此新 SSR 启用应用程序配置需要两个其他文件：`tsconfig.server.json` 和 `main.server.ts`。`Tsconfig.server.json` 文件指定 TypeScript 编译选项。`Main.server.ts` 文件期间 SSR 作为代码入口点。

添加新的文件称为 `tsconfig.server.json` 内 `ClientApp/src` (与现有一起 `tsconfig.app.json`)，包含以下：

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "baseUrl": "./",
    "module": "commonjs"
  },
  "angularCompilerOptions": {
    "entryModule": "app/app.module#AppServerModule"
  }
}
```

此文件将配置角的 AoT 编译器对于调用模块看起来 `app.server.module`。通过创建一个新文件添加这 `ClientApp/src/app/app.server.module.ts` (与现有一起 `app.module.ts`) 包含以下：

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-loader';
import { AppComponent } from './app.component';
import { AppModule } from './app.module';

@NgModule({
  imports: [AppModule, ServerModule, ModuleMapLoaderModule],
  bootstrap: [AppComponent]
})
export class AppServerModule { }
```

此模块继承自客户端 `app.module` 并定义哪些额外角度模块 SSR 中均可用。

回想一下，新 `ssr` 中的条目 `.angular-cli.json` 引用一个入口点文件称为 `main.server.ts`。你尚未尚未添加该文件中，并现在是时候，若要这样做。创建一个新文件 `ClientApp/src/main.server.ts` (与现有一起 `main.ts`)，包含以下：

```
import 'zone.js/dist/zone-node';
import 'reflect-metadata';
import { renderModule, renderModuleFactory } from '@angular/platform-server';
import { APP_BASE_HREF } from '@angular/common';
import { enableProdMode } from '@angular/core';
import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
import { createServerRenderer } from 'aspnet-prerendering';
export { AppServerModule } from './app/app.server.module';

enableProdMode();

export default createServerRenderer(params => {
  const { AppServerModule, AppServerModuleNgFactory, LAZY_MODULE_MAP } = (module as any).exports;

  const options = {
    document: params.data.originalHtml,
    url: params.url,
    extraProviders: [
      provideModuleMap(LAZY_MODULE_MAP),
      { provide: APP_BASE_HREF, useValue: params.baseUrl },
      { provide: 'BASE_URL', useValue: params.origin + params.baseUrl }
    ]
  };

  const renderPromise = AppServerModuleNgFactory
    ? /* AoT */ renderModuleFactory(AppServerModuleNgFactory, options)
    : /* dev */ renderModule(AppServerModule, options);

  return renderPromise.then(html => ({ html }));
});
```

此文件的代码是什么 ASP.NET Core 执行每个请求运行时 `UseSpaPrerendering` 添加到的中间件启动类。处理接收 `params` 从 .NET 代码（如所请求的 URL）以及角度 SSR API 调用以获取生成的 HTML。

严格来讲，这不足以在开发模式下启用 SSR。若要使一个最终的更改，以便你的应用程序能否正常工作时发布至关重要。在您的应用程序的 main 中 .csproj 文件中，将 `BuildServerSideRenderer` 属性值设置为 `true`：

```
<!-- Set this to true if you enable server-side prerendering -->
<BuildServerSideRenderer>true</BuildServerSideRenderer>
```

这会将配置生成过程运行 `build:ssr` 在发布过程并将 SSR 文件部署到服务器。如果你不启用此功能，SSR 将在生产环境中失败。

你的应用程序运行时在开发或生产模式下，角度代码预先将呈现为 HTML 服务器上。客户端代码执行正常。

将数据从 .NET 代码传递到 TypeScript 代码

在 SSR，你可能需要将每个请求数据从 ASP.NET Core 应用程序传递到应用程序角度。例如，你无法传递 cookie 信息或内容从数据库读取。若要执行此操作，编辑你启动类。中的回调 `UseSpaPrerendering`，设置的值 `options.SupplyData` 如下所示：

```
options.SupplyData = (context, data) =>
{
    // Creates a new value called isHttpsRequest that's passed to TypeScript code
    data["isHttpsRequest"] = context.Request.IsHttps;
};
```

`SupplyData` 回调允许传递任意、每个请求、JSON 序列化数据（如字符串、布尔值或数字）。你 `main.server.ts` 代码接收为 `params.data`。例如，上面的代码示例将传递一个布尔值作为 `params.data.isHttpsRequest` 到 `createServerRenderer` 回调。你可以将传递给你的应用以支持角任何方式的其他部分。有关示例，请参阅如何 `main.server.ts` 传递 `BASE_URL` 到其构造函数声明为接收它的任何组件的值。

SSR 的缺点

并非所有应用都受益于 SSR。主要优点是任何可察觉的性能。达到你的应用程序，通过慢速网络连接或慢速的移动设备上的访问者看到初始 UI 快，即使它未采用一段时间才能提取或分析 JavaScript 捆绑包。但是，许多 SPA 主要用通过快速的计算机上的快速、内部公司网络应用程序几乎立刻出现的位置。

同时，没有启用 SSR 明显的缺点。它向你的开发过程的复杂性。你的代码必须在两个不同环境中运行：客户端和服务器端（在 Node.js 环境中从 ASP.NET Core 调用）。下面是一些需要牢记的事项：

- SSR 要求 Node.js 安装生产服务器上。这将自动对于某些部署方案，例如 Azure 应用程序服务，但对于其他操作系统，如 Azure Service Fabric 这种情况。
- 启用 `BuildServerSideRenderer` 生成标志原因你 `node_modules` 目录发布。此文件夹包含 20000 多文件，这提高了部署时间。
- 若要在 Node.js 环境中运行你的代码，它不能依赖于是否存在特定浏览器的 JavaScript API 例如 `window` 或 `localStorage`。如果你的代码（或你引用某些第三方库）尝试使用这些 API，你将获得 SSR 期间出错。例如，不使用 jQuery 因为它引用了在多个位置的特定浏览器的 API。若要防止出现错误，您必须避免 SSR 或避免特定浏览器的 API 或库。可以将任何对此类 API 的调用包装在检查以确保它们不在 SSR 过程中调用。例如，在 JavaScript 或 TypeScript 代码中使用如下所示检查：

```
if (typeof window !== 'undefined') {
    // Call browser-specific APIs here
}
```

使用 ASP.NET Core 响应项目模板

2018/4/10 • 4 min to read • [Edit Online](#)

注意

本文档不有关响应项目模板包括在 ASP.NET 核心 2.0。它是有关与其则可手动更新较新响应模板。默认情况下，该模板包含在 ASP.NET 核心 2.1。

已更新的响应项目模板提供了方便起点 ASP.NET Core 应用使用响应和[创建响应应用\(CRA\)](#) 约定，可以实现的丰富的客户端用户界面 (UI)。

该模板是等效于创建充当 API 的后端，将 ASP.NET Core 项目和标准 CRA 做出反应项目操作作为 UI 中，但承载在能够生成和发布作为一个单元的单个应用程序项目中的便利。

创建新的应用程序

如果使用 ASP.NET 核心 2.0，请确保你已[安装更新的响应项目模板](#)。如果你有 ASP.NET 核心 2.1，则不需要安装它。

从命令提示符下使用命令创建新项目 `dotnet new react` 空的目录中。例如，以下命令创建的应用程序中我新应用目录并切换到该目录：

```
dotnet new react -o my-new-app  
cd my-new-app
```

从 Visual Studio 或.NET Core CLI 运行应用程序：

- [Visual Studio](#)
- [.NET Core CLI](#)

打开生成*.csproj*文件中，并从那里运行正常的应用程序。

生成过程还原首次运行，可能需要几分钟的 npm 依赖关系。后续的生成处于快得多。

项目模板创建一个 ASP.NET Core 应用和响应应用程序。ASP.NET Core 应用旨在用于数据访问、授权和其他服务器端问题。响应应用程序中，驻留在*ClientApp*子目录，旨在用于所有 UI 问题。

添加页面、映像、样式、模块、等。

*ClientApp*目录是标准的 CRA 做出响应的应用程序。请参阅官方[CRA 文档](#)有关详细信息。

略有差异通过此模板创建的响应应用程序和项目之间创建由 CRA 然后重试。但是，应用程序的功能保持不变。由模板创建此应用程序包含[Bootstrap](#)-基于布局和一个基本的路由示例。

安装 npm 包

若要安装第三方 npm 包，使用命令提示符处，*ClientApp*子目录。例如：

```
cd ClientApp  
npm install --save <package_name>
```

发布和部署

在开发中，应用将在优化为开发人员方便起见模式下运行。例如，JavaScript 捆绑包包括源地图（以便在调试时，你可以看到原始源代码）。该应用监视 JavaScript、HTML 和 CSS 文件在磁盘上的更改自动重新编译并重新加载当它发现更改这些文件。

在生产中，提供您针对性能进行了优化的应用程序的版本。这被配置为自动发生这种情况。发布时，生成配置会发出缩减，transpiled 生成的客户端代码。与不同的生成、生产生成不需要在服务器上安装的 Node.js。

您可以使用标准[ASP.NET Core 托管和部署方法](#)。

独立运行 CRA 服务器

将项目配置为在开发模式下启动 ASP.NET Core 应用时在后台启动自己 CRA development server 的实例。这很方便，因为它意味着无需手动运行单独的服务器。

没有此默认设置的缺陷。每次修改你的 C# 代码和应用程序需要重新启动，将 ASP.NET 核心 CRA 服务器重新启动。几秒钟后需要重新启动。如果您正在进行频繁的 C# 代码编辑，并且不希望等待 CRA 服务器重新启动，运行 CRA 服务器外部，独立于 ASP.NET 核心进程。若要这样做：

1. 在命令提示符下，切换到*ClientApp*子目录，并启动 CRA 开发服务器：

```
cd ClientApp  
npm start
```

2. 修改 ASP.NET Core 应用程序而不是启动其自身的一个使用外部 CRA 服务器实例。在你启动类中，替换

```
spa.UseReactDevelopmentServer("http://localhost:3000");
```

```
spa.UseProxyToSpaDevelopmentServer("http://localhost:3000");
```

当你启动 ASP.NET Core 应用程序时，它不会启动 CRA 服务器。将改为使用手动启动实例。这使它能够启动并重新启动速度更快。它不再正在等待响应应用程序，以重新生成每次。

使用 ASP.NET Core 响应与回顾的项目模板

2018/4/10 • 1 min to read • [Edit Online](#)

注意

本文档不有关响应与回顾的项目模板包括在 ASP.NET 核心 2.0。它是有关与其则可手动更新较新的响应与回顾模板。默认情况下，该模板包含在 ASP.NET 核心 2.1。

已更新的响应与 Redux 项目模板提供了方便的起始点，为使用 ASP.NET Core 应用做出响应，Redux，和[创建响应应用\(CRA\)](#)约定，可以实现的丰富的客户端用户界面 (UI)。

除了项目创建命令的响应与回顾模板有关的所有信息都是响应模板相同。若要创建此项目类型，请运行 `dotnet new reactredux` 而不是 `dotnet new react`。有关响应基于这两个模板共同的功能的详细信息，请参阅[做出响应模板文档](#)。

要开始使用 SignalR 在 ASP.NET Core 上

2018/5/17 • 6 min to read • [Edit Online](#)

作者: [Rachel Appel](#)

本教程教生成实时应用程序使用 ASP.NET Core SignalR 的基础知识。

The image contains two side-by-side screenshots of a web browser displaying a SignalR Chat application. Both screenshots show a dark-themed interface with a header bar containing the title 'SignalRChat' and a three-line menu icon.

Screenshot 1 (Left):

- Name: Bill Gates
- Message: Hello, computers!
- Send button
- Log:
 - 3/15/2018 11:12:10 PM **Bill Gates**: Hello, computers!
 - 3/15/2018 11:12:29 PM **Satya Nadella**: Hello, cloud computers!

© 2018 - SignalRChat

Screenshot 2 (Right):

- Name: Satya Nadella
- Message: Hello, cloud computers!
- Send button
- Log:
 - 3/15/2018 11:12:10 PM **Bill Gates**: Hello, computers!
 - 3/15/2018 11:12:29 PM **Satya Nadella**: Hello, cloud computers!

© 2018 - SignalRChat

本教程演示了下列 SignalR 开发任务:

- 在 ASP.NET 核心 web 应用上创建 SignalR。
- 创建一个 SignalR 集线器，以将内容推送到客户端。
- 修改 `Startup` 类并将应用配置。

[查看或下载示例代码\(如何下载\)](#)

系统必备

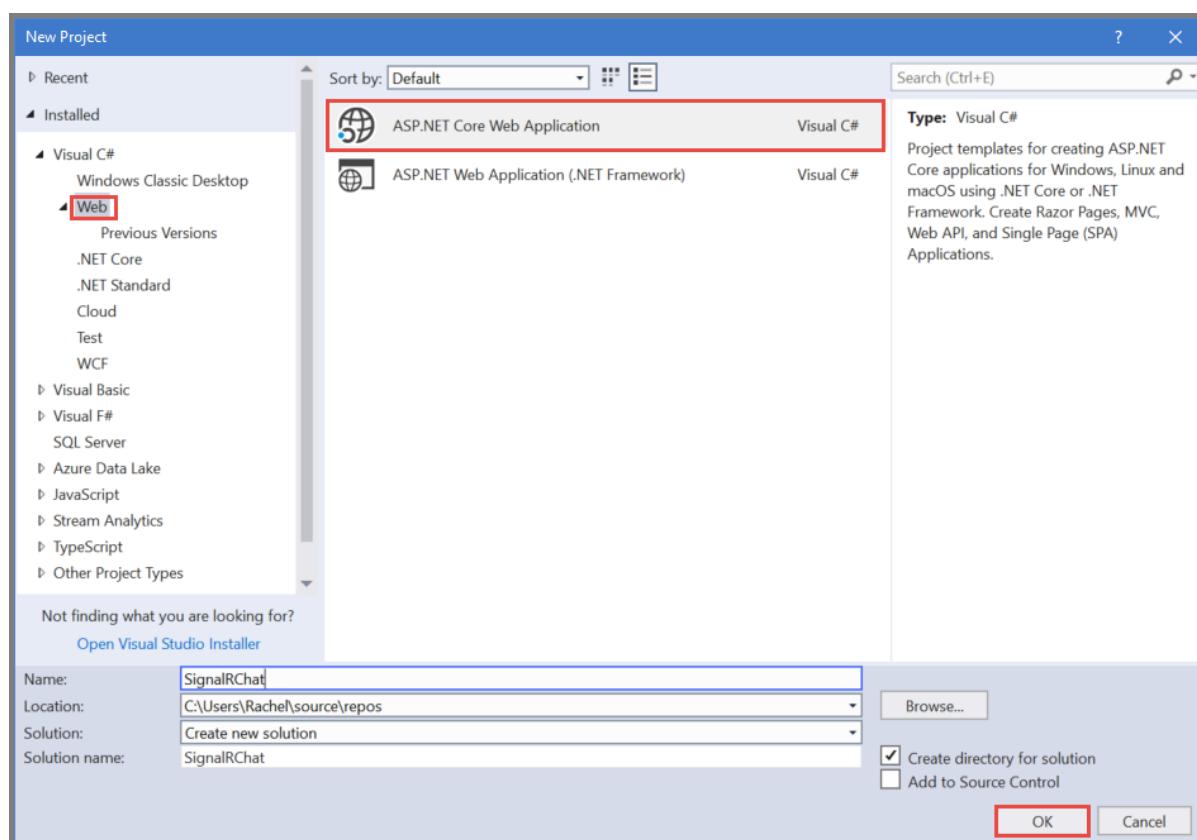
安装以下软件：

- [Visual Studio](#)
- [Visual Studio Code](#)
- [.NET 核心 2.1.0 RC 1 SDK 或更高版本](#)
- [Visual Studio 2017 15.7 或使用更高版本 ASP.NET 和 web 开发工作负载](#)
- [npm](#)

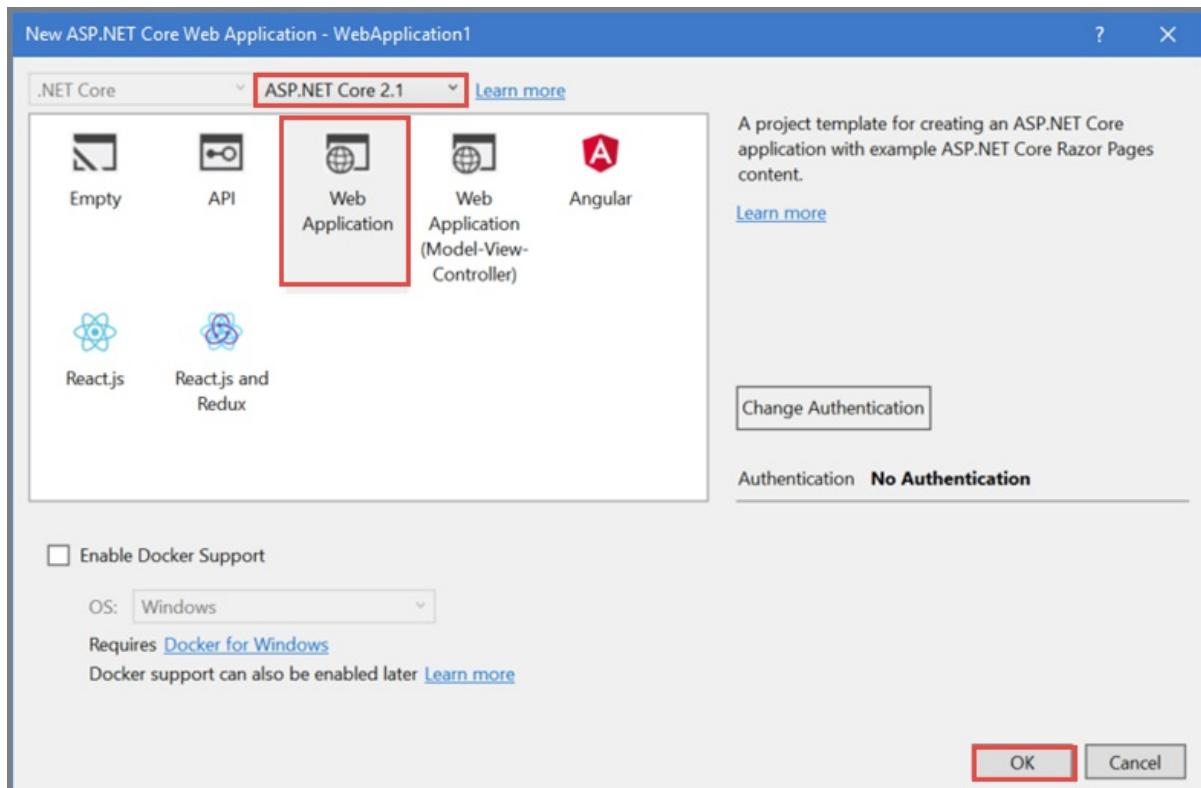
创建 ASP.NET Core 项目承载 SignalR 客户端和服务器

- [Visual Studio](#)
- [Visual Studio Code](#)

1. 使用文件 > 新项目菜单选项，然后选择**ASP.NET 核心 Web 应用程序**。将项目*SignalRChat*。



2. 选择**Web 应用程序**创建使用 Razor 页项目。然后选择**确定**。请确保**ASP.NET 核心 2.1**尽管 SignalR 运行在较旧版本的.NET framework 选择器，从选择。



Visual Studio 包含 `Microsoft.AspNetCore.SignalR` 作为的一部分包含其服务器库包其**ASP.NET 核心 Web 应用程序**模板。但是，适用于 SignalR 的 JavaScript 客户端库必须安装使用 `npm`。

3. 在中运行以下命令程序包管理器控制台窗口，请从项目根：

```
npm init -y  
npm install @aspnet/signalr
```

4. 复制 `signalrjs` 文件从 `node_modules\@aspnet\signalr\dist\browser*` 到 `lib*` 项目文件夹中的。

创建 SignalR Hub

允许客户端和服务器相互调用方法的高级管道作为服务的类，则集线器。

- [Visual Studio](#)
- [Visual Studio Code](#)

1. 将类添加到项目中，通过选择 **文件 > 新建 > 文件** 并选择 **Visual C# 类**。
2. 继承自 `Microsoft.AspNetCore.SignalR.Hub`。`Hub` 类包含属性和管理连接和组，以及发送和接收数据的事件。
3. 创建 `SendMessage` 将消息发送到所有连接的聊天客户端的方法。请注意它将返回 **任务**，这是因为 SignalR 是异步的。更好地缩放异步代码。

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

配置项目以使用 SignalR

必须配置 SignalR 服务器，这样就知道要传递给 SignalR 的请求。

1. 若要配置 SignalR 项目，请修改项目的 `Startup.ConfigureServices` 方法。

```
services.AddSignalR
```

 作为的一部分添加 SignalR 中间件管道。

2. 配置路由到你使用的中心 `UseSignalR`。

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });

            services.AddMvc();

            //services.AddCors(options => options.AddPolicy("CorsPolicy",
            //builder =>
            //{
            //    builder.AllowAnyMethod().AllowAnyHeader()
            //    .WithOrigins("http://localhost:55830")
            //    .AllowCredentials();
            //}));

            services.AddSignalR();
        }

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

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            //app.UseCors("CorsPolicy");
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chathub");
            });
            app.UseMvc();
        }
    }
}
```

创建 SignalR 客户端代码

1. 替换中的内容 `Pages\Index.cshtml` 替换为以下代码：

```
@page
<div class="container">
    <div class="row">&nbsp;</div>
    <div class="row">
        <div class="col-6">&nbsp;</div>
        <div class="col-6">
            User.....<input type="text" id="userInput" />
            <br />
            Message...<input type="text" id="messageInput" />
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6">&nbsp;</div>
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/lib/signalr.js"></script>
<script src="~/js/chat.js"></script>
/*
    If you need a version of the chat script that is compatible with IE 11, replace the above <script>
    tag that imports chat.js with this one
    <script src="~/js/es5-chat.js"></script>
*/

```

前面的 HTML 显示名称和消息字段和提交按钮。请注意在底部的脚本引用：至 SignalR 的引用和 `chat.js`。

2. 添加一个名为的 JavaScript 文件 `chat.js` 到 `wwwroot\js` 文件夹。向新文件添加以下代码：

```
// The following sample code uses modern ECMAScript 6 features
// that aren't supported in Internet Explorer 11.
// To convert the sample for environments that do not support ECMAScript 6,
// such as Internet Explorer 11, use a transpiler such as
// Babel at http://babeljs.io/.
//
// See Es5-chat.js for a Babel transpiled version of the following code:

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .build();

connection.on("ReceiveMessage", (user, message) => {
    const encodedMsg = user + " says " + message;
    const li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

document.getElementById("sendButton").addEventListener("click", event => {
    const user = document.getElementById("userInput").value;
    const message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(err => console.error(err.toString()));
    event.preventDefault();
});

connection.start().catch(err => console.error(err.toString()));
```

运行应用

- [Visual Studio](#)
- [Visual Studio Code](#)

1. 选择调试 > 启动而不调试启动浏览器并加载网站本地。从地址栏复制 URL。
2. 打开另一个浏览器实例（任何浏览器），然后在地址栏中粘贴该 URL。
3. 选择任一浏览器，输入名称和消息，然后单击发送按钮。名称和消息会显示在两个页面上立即。

SignalRChat

Name

Message

3/15/2018 11:12:10 PM **Bill Gates**: Hello, computers!
3/15/2018 11:12:29 PM **Satya Nadella**: Hello, cloud computers!

© 2018 - SignalRChat

SignalRChat

Name

Message

3/15/2018 11:12:10 PM **Bill Gates**: Hello, computers!
3/15/2018 11:12:29 PM **Satya Nadella**: Hello, cloud computers!

© 2018 - SignalRChat

相关资源

[ASP.NET 核心 SignalR 简介](#)

ASP.NET 核心使用 SignalR 中的中心

2018/5/18 • 3 min to read • [Edit Online](#)

通过[Rachel Appel](#)和[Kevin 怪兽](#)

[查看或下载的示例代码 \(如何下载\)](#)

什么是 SignalR hub

SignalR 中心 API，你可以从服务器连接的客户端上调用方法。在服务器代码中，你可以定义客户端调用的方法。在客户端代码中，你可以定义从服务器调用的方法。SignalR 负责在后台，使实时客户端到服务器和服务器客户端通信的所有内容。

配置 SignalR 集线器

SignalR 中间件需要某些服务，通过调用配置 `services.AddSignalR()`。

```
services.AddSignalR();
```

在 SignalR 功能添加到 ASP.NET 核心应用程序时，通过调用设置 SignalR 路由 `app.UseSignalR` 中 `Startup.Configure` 方法。

```
app.UseSignalR(route =>
{
    route.MapHub<ChatHub>("/chathub");
});
```

创建并使用中心

通过声明一个类继承自创建中心 `Hub`，并向其中添加公共方法。客户端可以调用方法定义为 `public`。

```

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }

    public Task SendMessageToCaller(string message)
    {
        return Clients.Caller.SendAsync("ReceiveMessage", message);
    }

    public Task SendMessageToGroups(string message)
    {
        List<string> groups = new List<string>() { "SignalR Users" };
        return Clients.Groups(groups).SendAsync("ReceiveMessage", message);
    }

    public override async Task OnConnectedAsync()
    {
        await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
        await base.OnConnectedAsync();
    }

    public override async Task OnDisconnectedAsync(Exception exception)
    {
        await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR Users");
        await base.OnDisconnectedAsync(exception);
    }
}

```

你可以指定返回类型和参数，包括复杂类型并~~对~~数组，就像在任何 C# 方法。SignalR 处理的序列化和反序列化的复杂对象和数组中参数和返回值。

客户端对象

每个实例 `Hub` 类有一个名为 `Clients` 包含服务器和客户端之间通信的以下成员：

属性	描述
<code>All</code>	在所有连接的客户端上调用方法
<code>Caller</code>	调用 hub 方法在客户端上调用方法
<code>Others</code>	除调用的方法的客户端的所有连接客户端上调用方法

此外，`Hub.Clients` 包含以下方法：

方法	描述
<code>AllExcept</code>	在指定的连接除外的所有连接的客户端上调用方法
<code>Client</code>	在特定连接的客户端上调用方法
<code>Clients</code>	在特定连接的客户端上调用方法
<code>Group</code>	将消息发送到指定的组中的所有连接

方法	描述
GroupExcept	将消息发送到指定的组，除非指定连接中的所有连接
Groups	将一条消息发送到多个组的连接
OthersInGroup	将一条消息发送到一组的连接，不包括客户端调用 hub 方法
User	将消息发送到与特定用户关联的所有连接
Users	将消息发送到与指定的用户相关联的所有连接

每个属性或方法上表中的返回一个包含对象 `SendAsync` 方法。 `SendAsync` 方法允许你提供的名称和要调用的客户端方法的参数。

将消息发送到客户端

若要使对特定客户端的调用，使用的属性 `Clients` 对象。在下面的示例中，`SendMessageToCaller` 方法演示如何将消息发送到调用 hub 方法的连接。`SendMessageToGroups` 方法将消息发送到存储中的组 `List` 名为 `groups`。

```
public Task SendMessageToCaller(string message)
{
    return Clients.Caller.SendAsync("ReceiveMessage", message);
}

public Task SendMessageToGroups(string message)
{
    List<string> groups = new List<string>() { "SignalR Users" };
    return Clients.Groups(groups).SendAsync("ReceiveMessage", message);
}
```

处理事件的连接

SignalR 中心 API 提供 `OnConnectedAsync` 和 `OnDisconnectedAsync` 虚拟方法，以管理和跟踪连接。重写 `OnConnectedAsync` 虚拟方法，客户端连接到集线器，如将其添加到组时，执行操作。

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnConnectedAsync();
}

public override async Task OnDisconnectedAsync(Exception exception)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnDisconnectedAsync(exception);
}
```

处理错误

在中心方法中引发的异常会发送到调用的方法的客户端。JavaScript 客户端上 `invoke` 方法返回 JavaScript 承诺。当客户端收到错误处理程序附加到承诺使用 `catch`，其调用和作为 JavaScript 传递 `Error` 对象。

```
connection.invoke("SendMessage", user, message).catch(err => console.error(err));
```

相关资源

- [ASP.NET 核心 SignalR 简介](#)
- [JavaScript 客户端](#)
- [发布到 Azure](#)

使用 ASP.NET Core 的移动开发

2018/3/19 • 1 min to read • [Edit Online](#)

- [为本机移动应用创建后端服务](#)

使用 ASP.NET Core 为本机移动应用创建后端服务

2018/5/14 • 9 min to read • [Edit Online](#)

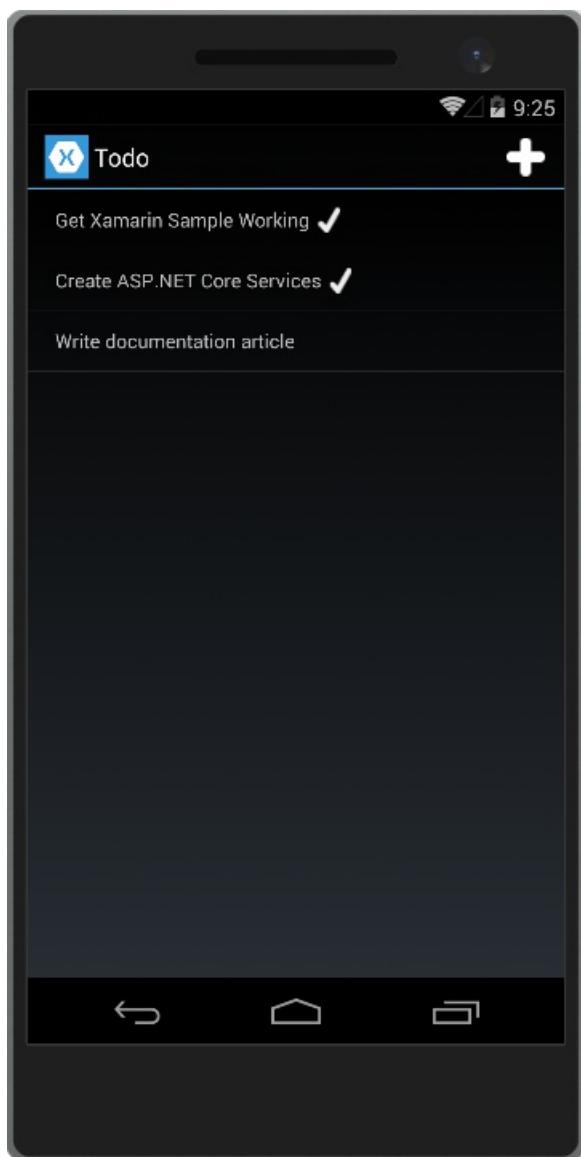
作者: [Steve Smith](#)

移动应用可与 ASP.NET Core 后端服务轻松通信。

[查看或下载后端服务代码示例](#)

本机移动应用示例

本教程演示如何创建使用 ASP.NET Core MVC 支持本机移动应用的后端服务。它使用 [Xamarin Forms ToDoRest 应用](#) 作为其本机客户端，其中包括 Android、iOS、Windows Universal 和 Window Phone 设备的单独本机客户端。你可以遵循链接中的教程来创建本机应用程序（并安装需要的免费 Xamarin 工具），以及下载 Xamarin 示例解决方案。Xamarin 示例包含一个 ASP.NET Web API 2 服务项目，使用本文中的 ASP.NET Core 应用替换（客户端无需进行任何更改）。

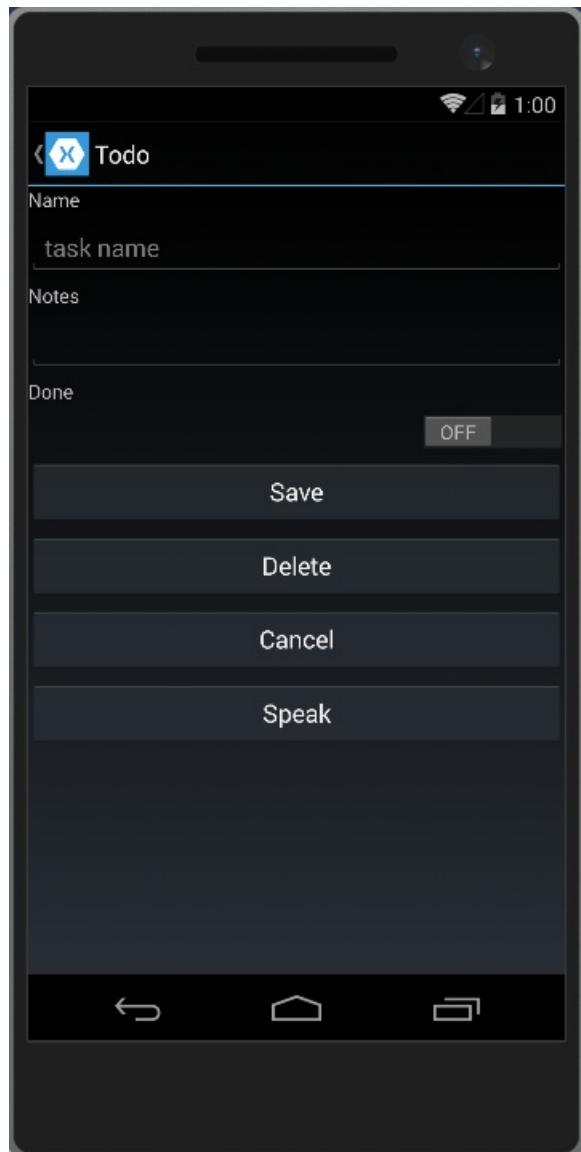


功能

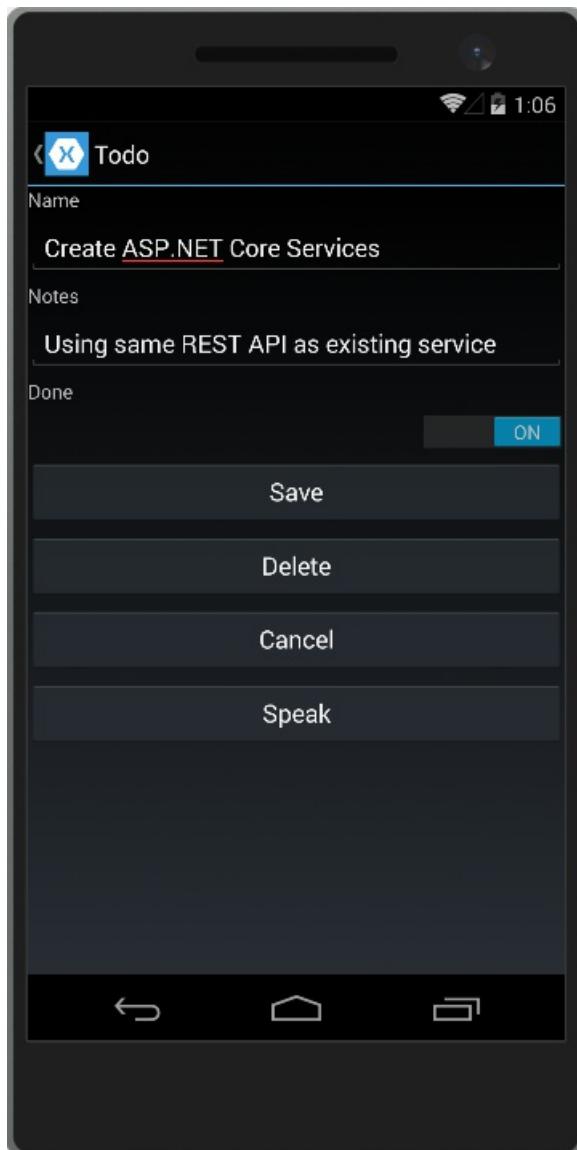
ToDoRest 应用支持列出、添加、删除和更新待办事项。每个项都有一个 ID、Name(名称)、Notes(说明)以及一个指示该项是否已完成的属性 Done。

待办事项的主视图如上所示，列出每个项的名称，并使用复选标记指示它是否已完成。

点击  图标打开“添加项”对话框：



点击主列表屏幕上的项将打开一个编辑对话框，在其中可以修改项的名称、说明以及是否完成，或删除项目：



此示例默认配置为使用托管在 developer.xamarin.com 上的后端服务，允许只读操作。若要使用在你计算机上运行的下一节创建的 ASP.NET Core 应用对其进行测试，你需要更新应用程序的 `RestUrl` 常量。导航到 `ToDoREST` 项目，然后打开 `Constants.cs` 文件。使用包含计算机 IP 的 URL 地址替换 `RestUrl`（不是 `localhost` 或 `127.0.0.1`，因为此地址用于从设备模拟器中，而不是从你的计算机中访问）。请包括端口号（`5000`）。为了测试你的服务能否在设备上正常运行，请确保没有活动的防火墙阻止访问此端口。

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems/{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

创建 ASP.NET Core 项目

在 Visual Studio 中创建一个新的 ASP.NET Core Web 应用程序。选择 Web API 模板和 No Authentication（无身份验证）。将项目命名为 `ToDoApi`。

Select a template:

ASP.NET Core Templates

Empty



Web API



Web Application

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET MVC Views and Controllers.

[Learn more](#)[Change Authentication](#)Authentication: **No Authentication**

Microsoft Azure

 Host in the cloud

App Service ▾

OK

Cancel

对于向端口 5000 进行的请求，应用程序均需作出响应。更新 Program.cs，使其包含 `.UseUrls("http://*:5000")`，以便实现以下操作：

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

注意

请确保直接运行应用程序，而不是在 IIS Express 后运行，因为在默认情况下，后者会忽略非本地请求。从命令提示符处运行 `dotnet run`，或从 Visual Studio 工具栏中的“调试目标”下拉列表中选择应用程序名称配置文件。

添加一个模型类来表示待办事项。使用 `[Required]` 属性标记必需字段：

```
using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

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

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

        public bool Done { get; set; }
    }
}
```

API 方法需要通过某种方式处理数据。使用原始 Xamarin 示例所用的 `IToDoRepository` 接口：

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

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}
```

在此示例中，该实现仅使用一个专用项集合：

```
using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
        }
    }
}
```

```

public ToDoItem Find(string id)
{
    return _todoList.FirstOrDefault(item => item.ID == id);
}

public void Insert(ToDoItem item)
{
    _todoList.Add(item);
}

public void Update(ToDoItem item)
{
    var todoItem = this.Find(item.ID);
    var index = _todoList.IndexOf(todoItem);
    _todoList.RemoveAt(index);
    _todoList.Insert(index, item);
}

public void Delete(string id)
{
    _todoList.Remove(this.Find(id));
}

private void InitializeData()
{
    _todoList = new List<ToDoItem>();

    var todoItem1 = new ToDoItem
    {
        ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
        Name = "Learn app development",
        Notes = "Attend Xamarin University",
        Done = true
    };

    var todoItem2 = new ToDoItem
    {
        ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
        Name = "Develop apps",
        Notes = "Use Xamarin Studio/Visual Studio",
        Done = false
    };

    var todoItem3 = new ToDoItem
    {
        ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
        Name = "Publish apps",
        Notes = "All app stores",
        Done = false,
    };

    _todoList.Add(todoItem1);
    _todoList.Add(todoItem2);
    _todoList.Add(todoItem3);
}
}

```

在 *Startup.cs* 中配置该实现:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository, ToDoRepository>();
}
```

现可创建 `ToDoItemsController`。

提示

有关创建 Web API 的详细信息, 请参阅[使用 ASP.NET Core MVC 和 Visual Studio 生成首个 Web API](#)。

创建控制器

在项目中添加新控制器 `ToDoItemsController`。它应继承 `Microsoft.AspNetCore.Mvc.Controller`。添加 `Route` 属性以指示控制器将处理路径以 `api/todoitems` 开始的请求。路由中的 `[controller]` 标记会被控制器的名称代替(省略 `Controller` 后缀), 这对全局路由特别有用。详细了解 [路由](#)。

控制器需要 `IToDoRepository` 才能正常运行;通过控制器的构造函数请求该类型的实例。在运行时, 此实例将使用框架对 [依赖关系注入](#) 的支持来提供。

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly IToDoRepository _ToDoRepository;

        public ToDoItemsController(IToDoRepository ToDoRepository)
        {
            _ToDoRepository = ToDoRepository;
        }
    }
}
```

此 API 支持四个不同的 HTTP 谓词来执行对数据源的 CRUD(创建、读取、更新、删除)操作。最简单的是读取操作, 它对应于 HTTP GET 请求。

读取项目

要请求项列表, 可对 `List` 方法使用 GET 请求。`[HttpGet]` 方法的 `List` 属性指示此操作应仅处理 GET 请求。此操作的路由是在控制器上指定的路由。你不一定必须将操作名称用作路由的一部分。你只需确保每个操作都有唯一的和明确的路由。路由属性可以分别应用在控制器和方法级别, 以此生成特定的路由。

```
[HttpGet]
public IActionResult List()
{
    return Ok(_ToDoRepository.All);
}
```

`List` 方法返回 200 OK 响应代码和所有 ToDo 项, 并序列化为 JSON。

你可以使用多种工具测试新的 API 方法, 如 Postman, 如此处所示:

The screenshot shows the Postman application interface. At the top, there are tabs for Runner, Import, Builder (which is selected), and Team Library. On the right side of the header, there are icons for a gear (Sync), a bell (Notifications), and a dropdown menu. Below the header, the URL `http://192.168.1.207:5000/a` is entered in the address bar, along with a plus sign icon for adding new requests. To the right of the URL, it says "No environment" with a dropdown arrow and a "Sync Off" button. Below the address bar, there are buttons for "Send" (blue) and "Save" (grey). Underneath these buttons, there are tabs for Authorization, Headers (which is selected), Body, Pre-request Script, Tests, and a "Generate Code" button. In the Headers section, there is a table with columns "key" and "value". Below this, there are buttons for "Bulk Edit" and "Presets". The main content area shows the response details. At the top of the response panel, it says "Body", "Cookies", "Headers (4)", and "Tests". To the right, it shows "Status: 200 OK" and "Time: 69 ms". Below this, there are tabs for "Pretty" (selected), "Raw", "Preview", and "JSON" (with a dropdown arrow). The "Pretty" tab displays the JSON response in a readable, numbered format:

```
1 [ [ 2 { 3   "id": "6bb8a868-dba1-4f1a-93b7-24ebce87e243", 4     "name": "Learn app development", 5     "notes": "Attend Xamarin University", 6     "done": true 7 }, 8   { 9     "id": "b94afb54-a1cb-4313-8af3-b7511551b33b", 10    "name": "Develop apps", 11    "notes": "Use Xamarin Studio/Visual Studio", 12    "done": false 13 }, 14   { 15     "id": "ecfa6f80-3671-4911-aabe-63cc442c1ecf", 16     "name": "Publish apps", 17     "notes": "All app stores", 18     "done": false 19   } ]
```

创建项目

按照约定, 创建新数据项映射到 HTTP POST 谓词。`Create` 方法具有应用于该对象的 `[HttpPost]` 属性, 并接受 `ToDoItem` 实例。由于 `item` 参数将在 POST 的正文中传递, 因此该参数用 `[FromBody]` 属性修饰。

在该方法中, 会检查项的有效性和之前是否存在于数据存储, 并且如果没有任何问题, 则使用存储库添加。检查 `ModelState.IsValid` 将执行 模型验证, 应该在每个接受用户输入的 API 方法中执行此步骤。

```
[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _ToDoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TodoItemIDInUse.ToString());
        }
        _ToDoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

示例中使用一个枚举，后者包含传递到移动客户端的错误代码：

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

使用 Postman 测试添加新项，选择 POST 谓词并在请求正文中以 JSON 格式提供新对象。你还应添加一个请求标头指定 Content-Type 为 application/json。

Builder

No environment

POST http://192.168.1.207:5000/api/todoitems

Params Send Save

Authorization Headers (2) Body (Pre-request Script Tests Generate Code)

form-data x-www-form-urlencoded raw binary JSON (application/json)

```

1 {
2   "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
3   "Name": "A Test Item",
4   "Notes": "asdf",
5   "Done": false
6 }

```

Body Cookies Headers (4) Tests Status: 200 OK Time: 227 ms

Pretty Raw Preview JSON

```

1 {
2   "id": "6bb8b868-dba1-4f1a-93b7-24ebce87243",
3   "name": "A Test Item",
4   "notes": "asdf",
5   "done": false
6 }

```

该方法返回在响应中新建的项。

更新项目

通过使用 HTTP PUT 请求来修改记录。除了此更改之外，Edit 方法几乎与 Create 完全相同。请注意，如果未找到记录，则 Edit 操作将返回 NotFound (404) 响应。

```

[HttpPut]
public IActionResult Edit([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _ToDoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _ToDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}

```

若要使用 Postman 进行测试，将谓词更改为 PUT。在请求正文中指定要更新的对象数据。

The screenshot shows the Postman application interface. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), 'Team Library', and various status indicators. The main workspace shows a 'PUT' request to 'http://192.168.1.207:5000/api/todolist'. The 'Body' tab is active, showing raw JSON data:

```
1 {  
2     "ID": "6bb8b868-dba1-4f1a-93b7-24ebce87243",  
3     "Name": "An UPDATED Test Item",  
4     "Notes": "Some updated notes",  
5     "Done": true  
6 }
```

Below the body, the 'Body' tab is selected, showing the response status as '204 No Content' and time as '91 ms'. The response body is empty.

为了与预先存在的 API 保持一致，此方法在成功时返回 `NoContent` (204) 响应。

删除项目

删除记录可以通过向服务发出 DELETE 请求并传递要删除项的 ID 来完成。与更新一样，请求的项不存在时会收到 `NotFound` 响应。请求成功会得到 `NoContent` (204) 响应。

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

请注意，在测试删除功能时，请求正文中不需要任何内容。

The screenshot shows the Postman application interface. At the top, there's a navigation bar with icons for Runner, Import, Builder (which is highlighted in orange), Team Library, and various status indicators like SYNC OFF and a bell icon. Below the navigation bar, the URL `http://192.168.1.207:5000/a` is entered, and the environment dropdown shows "No environment".

The main area contains a request configuration panel. It shows a **DELETE** method selected, a target URL of `http://192.168.1.207:5000/api/todoitems/6bb8b8`, and a "Params" section. On the right, there are "Send" and "Save" buttons. A "Bulk Edit" button is also present.

Below the request configuration, there are tabs for Authorization, Headers (2), Body (which is currently selected), Pre-request Script, Tests, and Generate Code. Under the Body tab, the content type is set to `JSON (application/json)`. There are four radio button options: form-data, x-www-form-urlencoded, raw, and binary, with "raw" being selected.

The "Body" section itself is empty, indicated by a large gray box with the number "1" in the top-left corner.

At the bottom of the interface, there's a summary bar showing "Status: 204 No Content" and "Time: 91 ms". Below this, there are tabs for Body, Cookies, and Headers (2). The Body tab is selected again, showing a "Pretty" button, a "Raw" button, a "Preview" button, and a "HTML" dropdown set to "HTML". To the right of these buttons are icons for copy and search.

常见的 Web API 约定

开发应用程序的后端服务时，你将想要使用一组一致的约定或策略来处理横切关注点。例如，在上面所示服务中，针对不存在的特定记录的请求会收到 `NotFound` 响应，而不是 `BadRequest` 响应。同样，对于此服务，传递模型绑定类型的命令始终检查 `ModelState.IsValid` 并为无效的模型类型返回 `BadRequest`。

一旦为 API 指定通用策略，一般可以将其封装在 `Filter(筛选器)`。详细了解 [如何封装 ASP.NET Core MVC 应用程序中的通用 API 策略](#)。

托管和部署 ASP.NET Core

2018/5/17 • 4 min to read • [Edit Online](#)

一般而言，向托管环境部署 ASP.NET Core 应用需执行以下操作：

- 将应用发布到托管服务器上的文件夹。
- 设置进程管理器，该管理器在收到请求时启动应用，并在应用发生故障或服务器重启后重新启动应用。
- 设置将请求转发到应用的反向代理。

发布到文件夹

[dotnet 发布](#) CLI 命令编译应用代码并复制所需的文件，以便在“发布”文件夹中运行应用。从 Visual Studio 进行部署时，在文件被复制到部署目标前，系统会自动执行 [dotnet publish](#) 步骤。

文件夹内容

“发布”文件夹包含应用的“.exe”和“.dll”文件、应用依赖项以及 .NET 运行时（根据需要）。

.NET Core 应用可以作为“独立”应用或“依赖于框架”的应用进行发布。如果应用是独立应用，则包含 .NET 运行时的“.dll”文件会包括在“发布”文件夹内。如果应用依赖于框架，.NET 运行时文件就不包含在内，因为应用包含对服务器上安装的 .NET 版本的引用。默认部署模型是依赖于框架的模型。有关详细信息，请参阅 [.NET Core 应用程序部署](#)。

除了“.exe”和“.dll”文件以外，ASP.NET Core 应用的“发布”文件夹通常包含配置文件、静态资产和 MVC 视图。有关详细信息，请参阅 [目录结构](#)。

设置进程管理器

ASP.NET Core 应用是一个控制台应用，在服务器启动时必须启动该应用，并且在出现故障后必须重新启动它。若要自动启动和重新启动，需要使用进程管理器。用于 ASP.NET Core 的最常见进程管理器是：

- Linux
 - [Nginx](#)
 - [Apache](#)
- Windows
 - [IIS](#)
 - [Windows 服务](#)

设置反向代理

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果该应用使用的是 [Kestrel](#) Web 服务器，则可以将 [Nginx](#)、[Apache](#) 或 [IIS](#) 用作反向代理服务器。反向代理服务器接收到来自 Internet 的 HTTP 请求，并在进行一些初步处理后将这些请求转发到 Kestrel。有关详细信息，请参阅 [何时结合使用 Kestrel 和反向代理](#)。

代理服务器和负载均衡器方案

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。如不提供附加配置，应用可能无法访问方案（HTTP/HTTPS）和发出请求的远程 IP 地址。有关详细信息，请参阅 [配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

使用 Visual Studio 和 MSBuild 自动执行部署

除将输出从 `dotnet publish` 复制到服务器外，部署通常还需要其他任务。例如，可能需要从“发布”文件夹获取或排除额外文件。Visual Studio 使用 MSBuild 进行 Web 部署，用户可以自定义 MSBuild 以便在部署期间执行其他任务。有关详细信息，请参阅 [Visual Studio 中的发布配置文件](#) 和 [使用 MSBuild 和 Team Foundation Build 一书](#)。

通过[发布 Web 功能或内置 Git 支持](#)，可以将应用从 Visual Studio 直接部署到 Azure 应用服务。Visual Studio Team Services 支持[对 Azure App Service 进行持续部署](#)。

发布到 Azure

有关如何使用 Visual Studio 将应用发布到 Azure 的说明，请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。还可以从[命令行](#)将该应用发布到 Azure。

其他资源

有关将 Docker 用作托管环境的信息，请参阅[在 Docker 中托管 ASP.NET Core 应用](#)。

在 Azure 应用服务上托管 ASP.NET Core

2018/5/14 • 7 min to read • [Edit Online](#)

Azure 应用服务是一个用于托管 Web 应用(包括 ASP.NET Core)的 Microsoft 云计算平台服务。

有用资源

Azure 应用文档、教程、示例、操作指南和其他资源由 Azure [Web 应用文档](#)提供。两个有关托管 ASP.NET Core 应用的重要教程为：

[快速入门:在 Azure 中创建 ASP.NET Core Web 应用](#)

使用 Visual Studio 创建 ASP.NET Core Web 应用，并将其部署到 Windows 上的 Azure 应用服务。

[快速入门:在 Linux 上的应用服务中创建 .NET Core Web 应用](#)

使用命令行创建 ASP.NET Core Web 应用，并将其部署到 Linux 上的 Azure 应用服务。

ASP.NET Core 文档中提供以下文章：

[使用 Visual Studio 发布到 Azure](#)

了解如何使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure 应用服务。

[使用 CLI 工具发布到 Azure](#)

了解如何使用 Git 命令行客户端将 ASP.NET Core 应用发布到 Azure 应用服务。

[使用 Visual Studio 和 Git 持续部署到 Azure](#)

了解如何使用 Visual Studio 创建 ASP.NET Core Web 应用并使用 Git 将它部署到 Azure 应用服务以实现持续部署。

[使用 VSTS 持续部署到 Azure](#)

为 ASP.NET Core 应用设置 CI 生成，然后创建针对 Azure 应用服务的持续部署发布。

[Azure Web 应用沙盒](#)

探索 Azure Apps 平台强制实施的 Azure 应用服务运行时执行限制。

应用程序配置

在 ASP.NET Core 2.0 和更高版本中，[Microsoft.AspNetCore.All](#) 元包中的三个包为部署到 Azure 应用服务的应用提供自动日志记录功能：

- [Microsoft.AspNetCore.AzureAppServices.HostingStartup](#) 使用 [IHostingStartup](#) 提供 ASP.NET Core 与 Azure 应用服务的启动集成。添加的日志记录功能由 [Microsoft.AspNetCore.AzureAppServicesIntegration](#) 包提供。
- [Microsoft.AspNetCore.AzureAppServicesIntegration](#) 执行 [AddAzureWebAppDiagnostics](#)，在 [Microsoft.Extensions.Logging.AzureAppServices](#) 包中添加 Azure 应用服务诊断日志记录提供程序。
- [Microsoft.Extensions.Logging.AzureAppServices](#) 提供记录器实现，支持 Azure 应用服务诊断日志和日志流式处理功能。

代理服务器和负载均衡器方案

配置转发头中间件的 IIS 集成中间件和 ASP.NET Core 模块将配置为转发方案 (HTTP/HTTPS) 和发出请求的远程 IP 地址。对于托管在其他代理服务器和负载均衡器后方的应用，可能需要附加配置。有关详细信息，请参阅[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

监视和日志记录

有关监视、日志记录和故障排除的信息，请参阅以下文章：

[如何：在 Azure 应用服务中监视应用](#)

了解如何查看应用和应用服务计划的配额和指标。

[在 Azure 应用服务中为应用启用诊断日志记录](#)

了解如何启用和访问 HTTP 状态代码、失败请求和 Web 服务器活动的诊断日志记录。

[ASP.NET Core 中的错误处理简介](#)

了解在 ASP.NET Core 应用中处理错误的常见方法。

[对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)

了解如何使用 ASP.NET Core 应用诊断 Azure 应用服务部署问题。

[Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考](#)

使用故障排除建议查看 Azure 应用服务/IIS 托管的应用的常见部署配置错误。

数据保护密钥环和部署槽位

[数据保护密钥](#)保存在 %HOME%\ASP.NET\DataProtection-Keys 文件夹中。此文件夹由网络存储提供支持，并跨托管应用的所有计算机同步。密钥不是静态保护的。此文件夹向单个部署槽位中应用的所有实例提供密钥环。各部署槽位（例如过渡槽和生成槽）不共享密钥环。

在部署槽位之间交换时，使用数据保护的任意系统都无法使用之前槽位中的密钥环来解密存储的数据。ASP.NET Cookie 中间件使用数据保护来保护其 cookie。这导致用户注销使用标准 ASP.NET Cookie 中间件的应用。对于独立于槽位的密钥环解决方案，请使用外部密钥环提供程序，例如：

- Azure Blob 存储
- Azure Key Vault
- SQL 存储
- Redis 缓存

有关详细信息，请参阅[密钥存储提供程序](#)。

将 ASP.NET Core 预览版部署到 Azure 应用服务

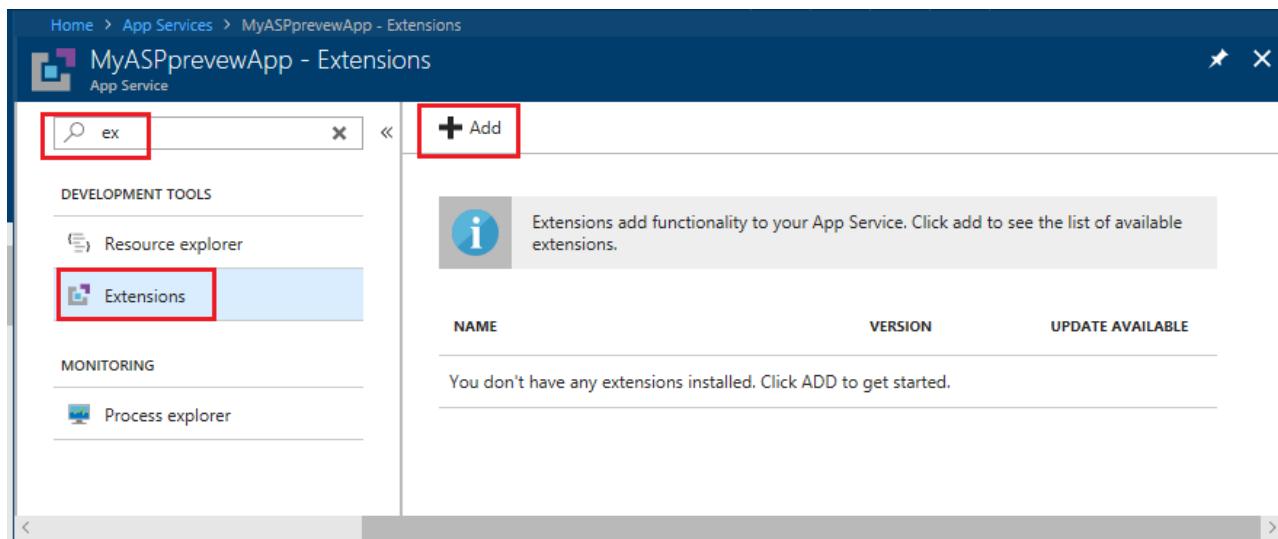
可通过以下方法将 ASP.NET Core 预览应用部署到 Azure 应用服务：

- [安装预览站点扩展](#)
- [部署自包含应用](#)
- [对用于容器的 Web 应用使用 Docker](#)

如果使用预览站点扩展时遇到问题，请在 [GitHub](#) 上打开相应的问题。

安装预览站点扩展

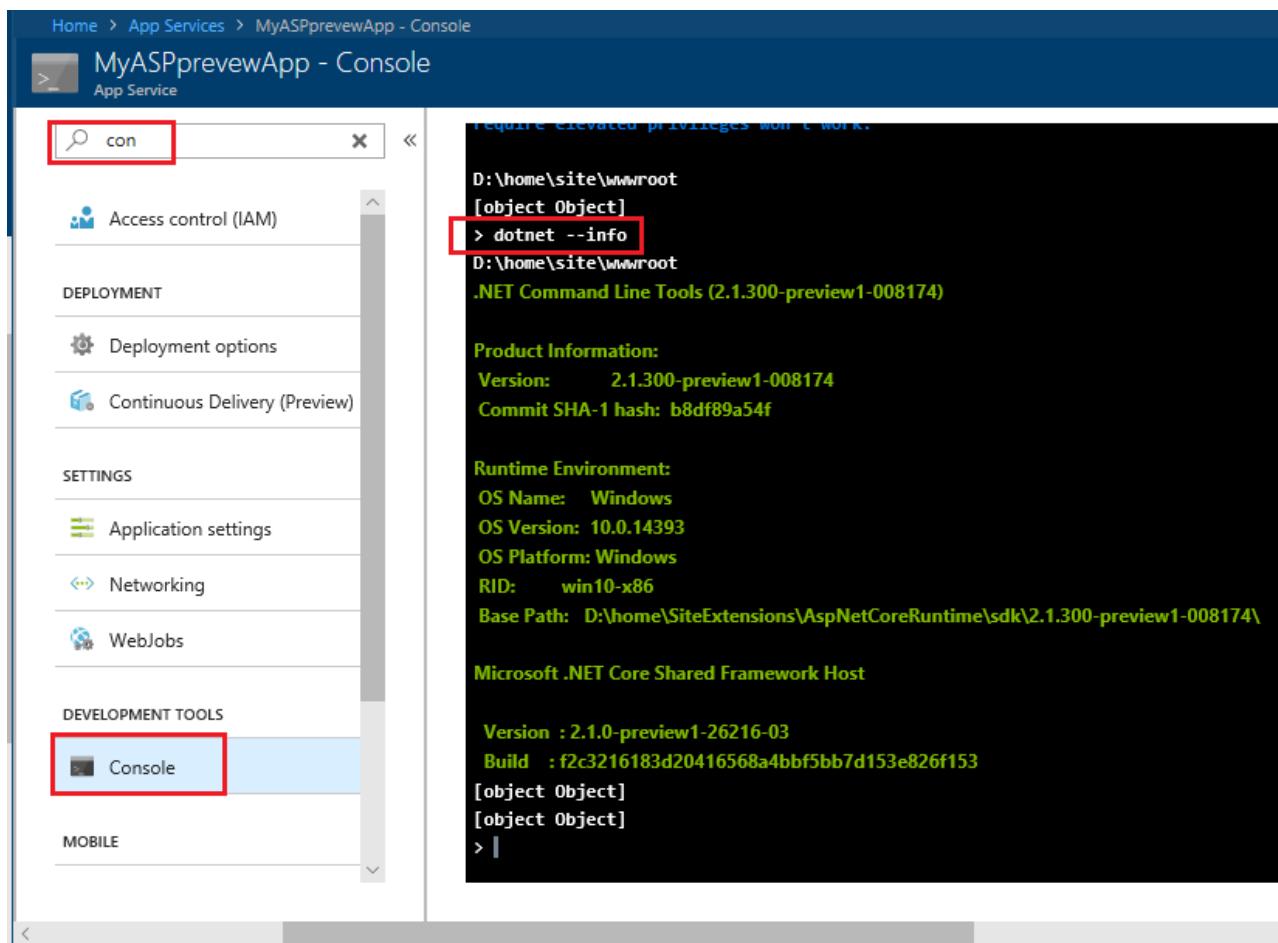
- 从 Azure 门户导航到“应用服务”边栏选项卡。
- 在搜索框中输入“ex”。
- 选择“扩展”。
- 选择“添加”。



- 选择“ASP.NET Core 2.1 (x86) 运行时”或“ASP.NET Core 2.1 (x64)”运行时。
- 选择“确定”。再次选择“确定”。

添加操作完成时，即表示已安装最新的 .NET Core 2.1 预览。通过在控制台中运行 `dotnet --info` 来验证安装。从“应用服务”边栏选项卡：

- 在搜索框中输入“con”。
- 选择“控制台”。
- 在控制台中输入 `dotnet --info`。



前面的图像在写入时是最新的。你可能会看到不同的版本。

`dotnet --info` 显示已安装该预览的站点扩展的路径。它显示应用从该站点扩展运行，而不是从默认的 ProgramFiles 位置运行。如果看到 ProgramFiles，请重启该站点并运行 `dotnet --info`。

通过 ARM 模板使用预览站点扩展

如果使用 ARM 模板创建和部署应用，则可使用 `siteextensions` 资源类型将站点扩展添加到 Web 应用。例如：

```
{  
    "type": "siteextensions",  
    "name": "AspNetCoreRuntime",  
    "apiVersion": "2015-04-01",  
    "location": "[resourceGroup().location]",  
    "properties": {  
        "version": "[parameters('aspnetcoreVersion')]"  
    },  
    "dependsOn": [  
        "[resourceId('Microsoft.Web/Sites', parameters('siteName'))]"  
    ]  
}
```

部署自包含应用

可以部署在部署中携带部署预览运行时的[自包含应用](#)。部署自包含应用时：

- 不需要准备站点。
- 相较于依赖框架的部署(使用服务器上的共享运行时和主机)的发布方式，必须以不同的方式发布此类应用。

对所有 ASP.NET Core 应用而言，自包含应用都是一个不错的选择。

对用于容器的 Web 应用使用 Docker

[Docker 中心](#)包含最新的 2.1 预览 Docker 映像。这些映像可以用作基础映像。按常规方法使用映像并部署到用于容器的 Web 应用。

其他资源

- [Web 应用概述\(5 分钟概述视频\)](#)
- [Azure 应用服务:托管 .NET 应用的最佳位置\(55 分钟概述视频\)](#)
- [Azure Friday: Azure 应用服务诊断和故障排除体验\(12 分钟视频\)](#)
- [Azure 应用服务诊断概述](#)

Windows Server 上的 Azure 应用服务使用 [Internet Information Services \(IIS\)](#)。以下是基础 IIS 技术的相关主题：

- [使用 IIS 在 Windows 上托管 ASP.NET Core](#)
- [ASP.NET Core 模块简介](#)
- [ASP.NET Core 模块配置参考](#)
- [IIS Modules 与 ASP.NET Core](#)
- [Microsoft TechNet 库:Windows Server](#)

使用 Visual Studio 将 ASP.NET Core 应用发布到 Azure

2018/5/14 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)、[Cesar Blum Silveira](#) 和 [Rachel Appel](#)

重要事项

要注意, 对于使用 **ASP.NET 核心 2.1 预览版本**

请参阅[到 Azure App Service 部署 ASP.NET Core 预览版](#)。

如果你在 macOS 上工作, 请参阅[从 Visual Studio for Mac 发布到 Azure](#)。

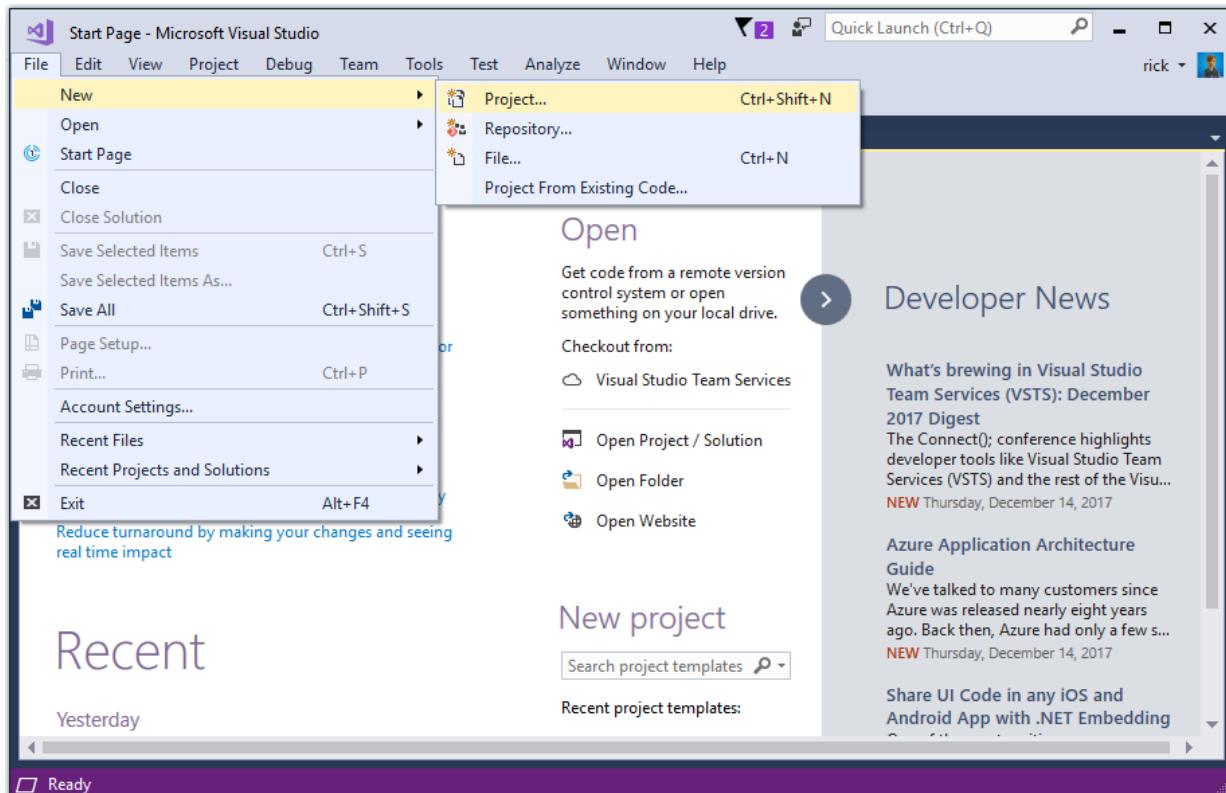
要解决应用服务部署问题, 请参阅[对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)。

设置

- 如果没有[免费 Azure 帐户](#), 请开设一个。

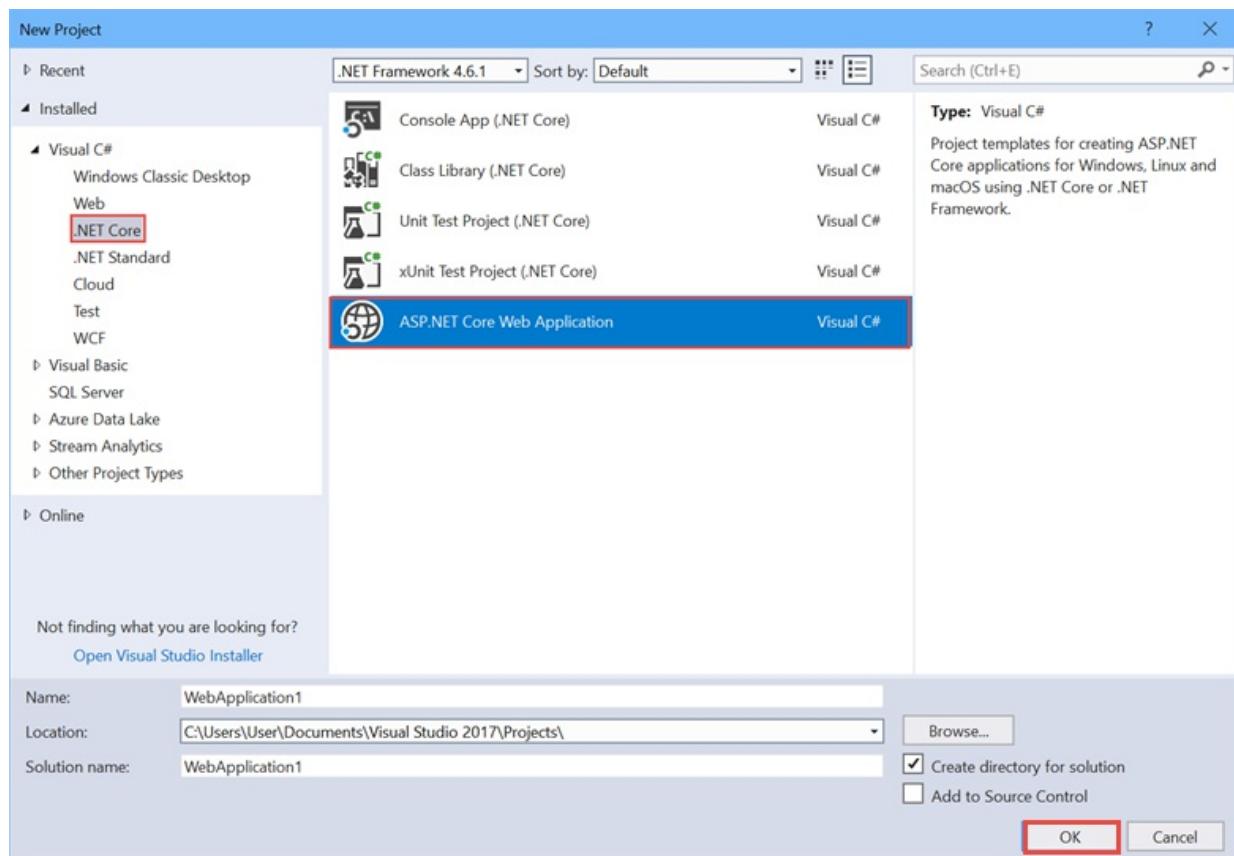
创建 Web 应用

在 Visual Studio 起始页中, 选择“文件”>“新建”>“项目...”



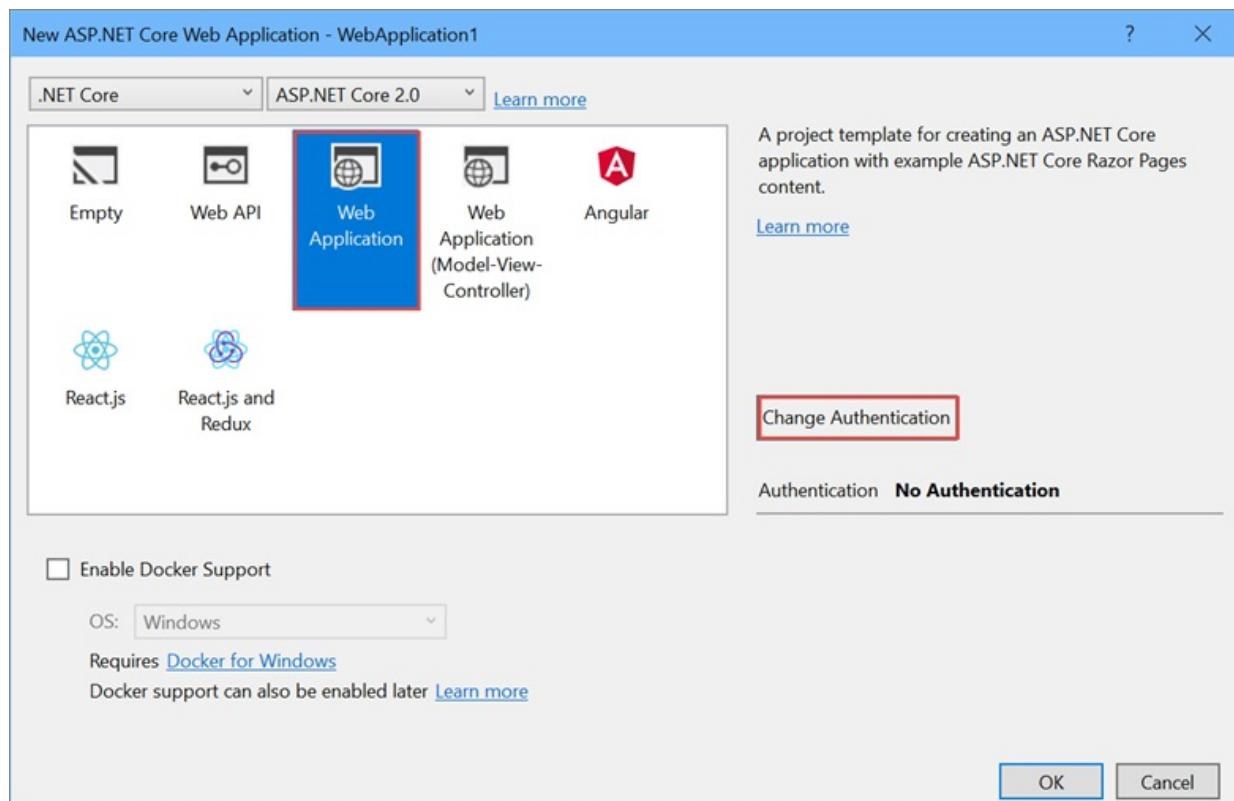
完成“新建项目”对话框:

- 在左侧窗格中, 选择“.NET Core”。
- 在中间窗格中, 选择“ASP.NET Core Web 应用程序”。
- 选择“确定”。



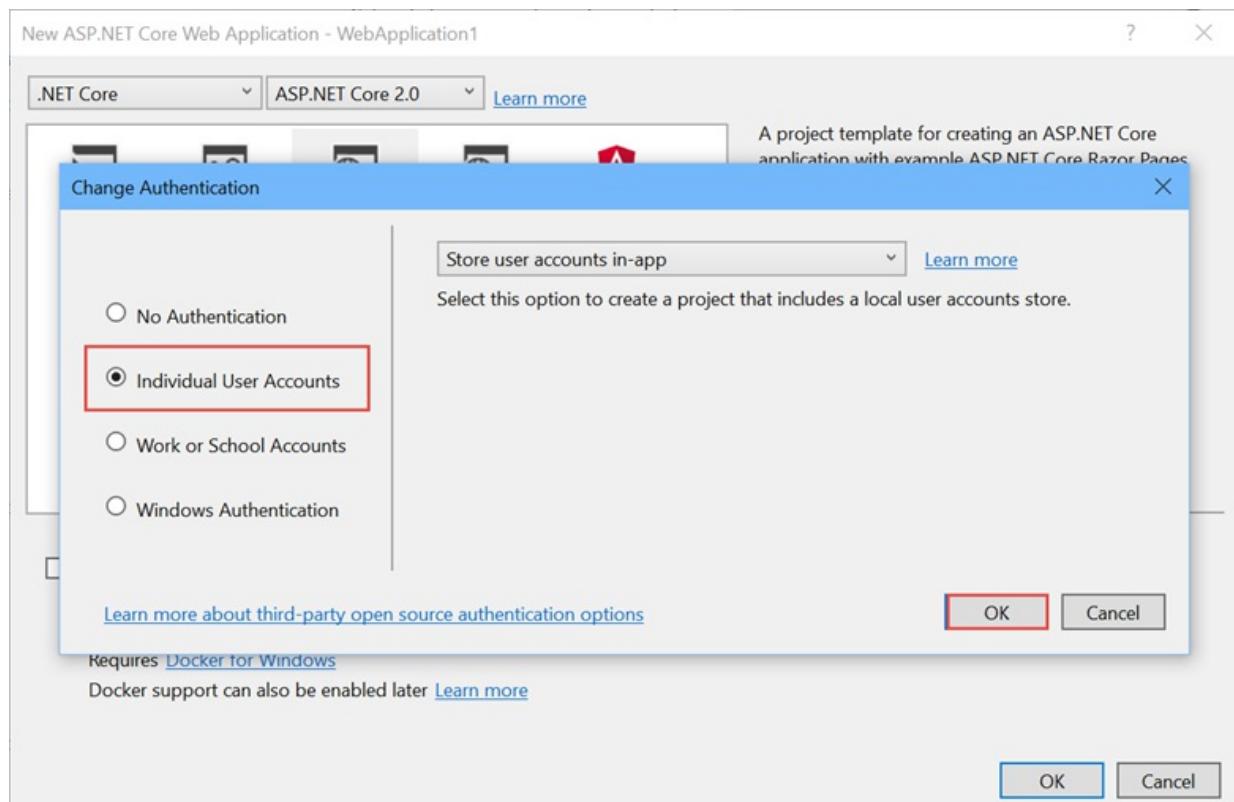
在“新建 ASP.NET Core Web 应用程序”对话框中：

- 选择“Web 应用程序”。
- 选择“更改身份验证”。



“更改身份验证”对话框随即出现。

- 选择“个人用户帐户”。
- 选择“确定”返回到“新建 ASP.NET Core Web 应用程序”，然后再次选择“确定”。



Visual Studio 随即创建解决方案。

运行应用

- 按 Ctrl+F5 运行项目。
- 测试“关于”和“联系”链接。

To see favorites here, select then , and drag to the Favorites Bar folder. Or import from another browser. Import favorites

WebApplication1 Home About Contact Register Log in

ASP.NET Core | Windows | Linux | OSX

Learn how to build ASP.NET apps that can run anywhere.

Application uses

- Sample pages using ASP.NET Core Razor Pages
- Bower for managing client-side libraries
- Theming using Bootstrap

How to

- Working with Razor Pages.
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet.
- Add client packages using Bower.
- Target development, staging or production environment.

Overview

- Conceptual overview of what is ASP.NET Core
- Fundamentals of ASP.NET Core such as Startup and middleware.
- Working with Data
- Security
- Client side development
- Develop on different platforms
- Read more on the documentation site

Run & Deploy

- Run your app
- Run tools such as EF migrations and more
- Publish to Microsoft Azure App Service

注册用户

- 选择“注册”并注册新用户。可使用虚构电子邮件地址。提交时，页面上会显示以下错误：

“内部服务器错误: 处理请求时, 数据库操作失败。SQL 异常: 无法打开数据库。可通过向应用程序数据库上下文应用现有迁移解决此问题。”

- 选择“应用迁移”, 并在页面更新后刷新页面。

A database operation failed while processing the request.

SqlException: Cannot open database "aspnet-WebApplication1-3caf68c5-2764-4579-a54b-15b98365379b" requested by the login. The login failed. Login failed for user 'D\user'.

Applying existing migrations for ApplicationDbContext may resolve this issue

There are migrations for ApplicationDbContext that have not been applied to the database

- 000000000000_CreatedIdentitySchema

[Apply Migrations](#)

In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:

PM> Update-Database

Alternatively, you can apply pending migrations from a command prompt at your project directory:

> dotnet ef database update

应用将显示用于注册新用户的电子邮件和一个“注销”链接。

Home page - WebApplic

Secure | https://localhost:44323

Guest

WebApplication1 Home About Contact

Hello user@example.com! Log out

Microsoft Azure | Learn how Microsoft's Azure cloud platform allows you to build, deploy, and scale web apps.

Learn More

Application uses

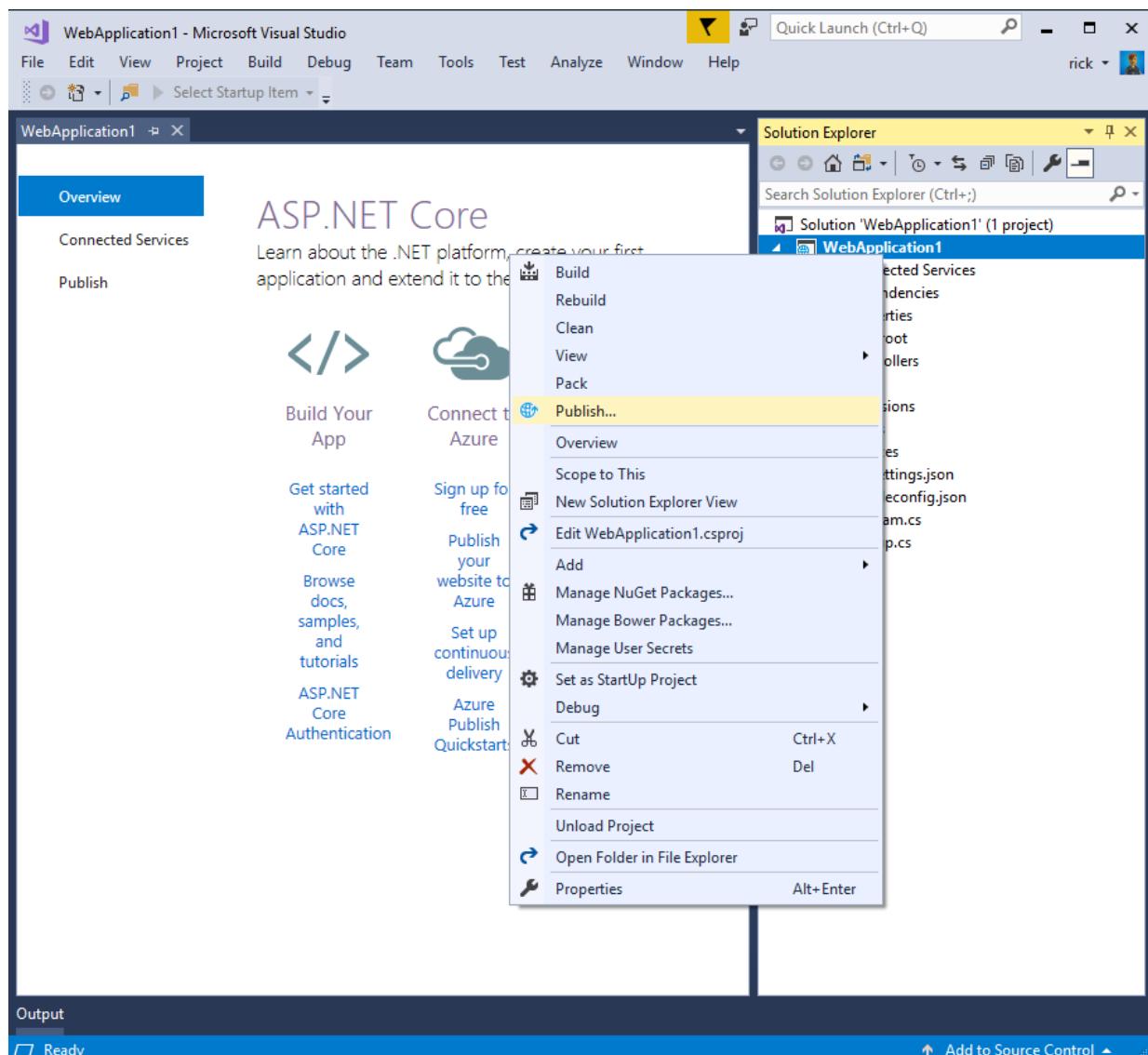
- Sample pages using ASP.NET Core Razor Pages
- Theming using Bootstrap

How to

- Working with Razor Pages.
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet

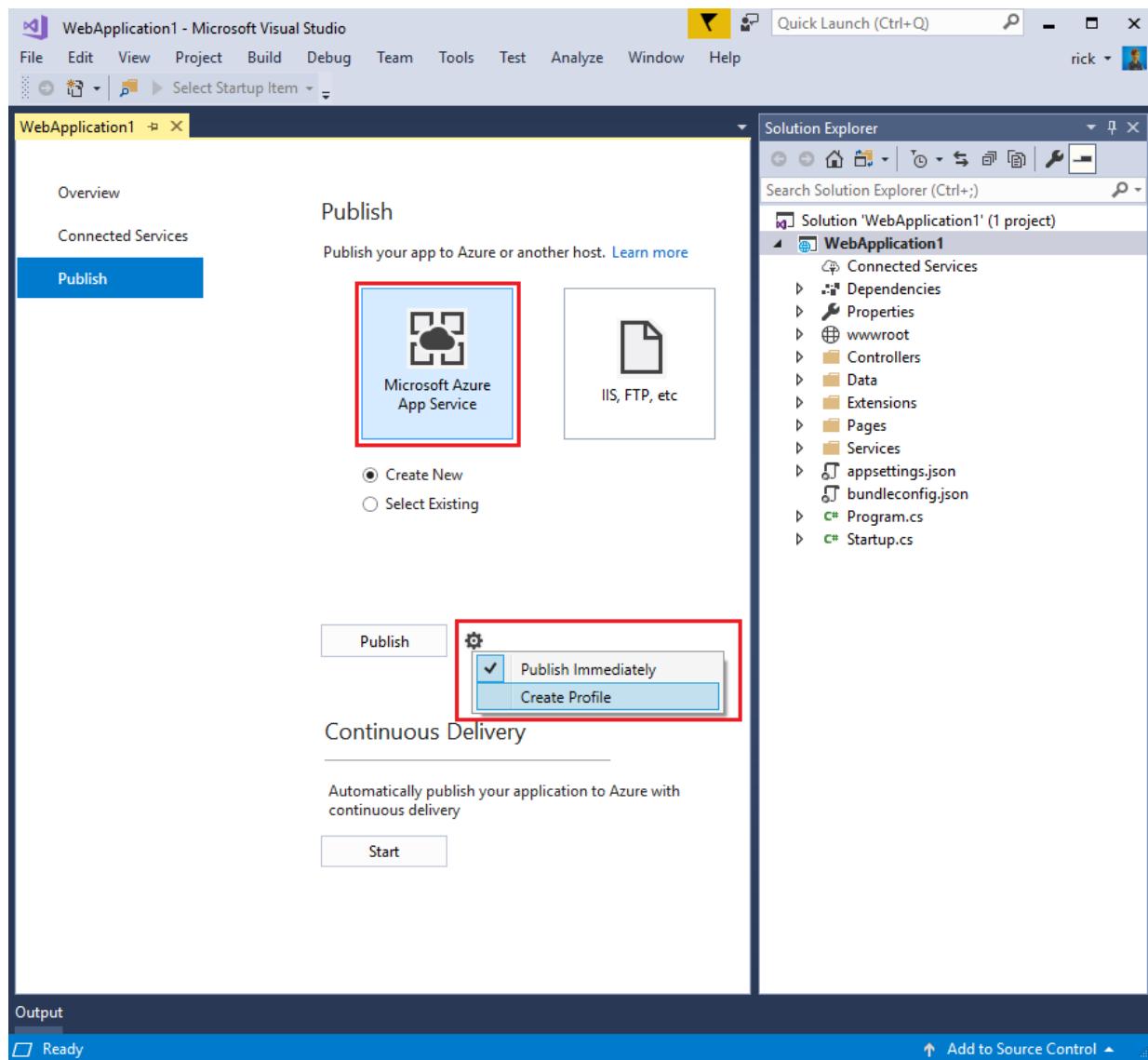
将应用部署到 Azure

在解决方案资源管理器中右键单击该项目，然后选择“发布...”。



在“发布”对话框中：

- 选择“Microsoft Azure 应用服务”。
- 选择齿轮图标，然后选择“创建配置文件”。
- 选择“创建配置文件”。



创建 Azure 资源

"创建应用服务"对话框随即显示：

- 输入订阅。
- “应用名称”、“资源组”和“应用服务计划”输入字段已填充。可以保留这些名称，也可以进行更改。

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account
a.ricka00@gmail.com

Hosting (i) **Services**

App Name Change Type ▾
WebApplication120171215025005

Subscription
MSDN

Resource Group
WebApplication120171215025005ResourceGroup* New...

App Service Plan
WebApplication120171215025005Plan* New...

Clicking the Create button will create the following Azure resources
[Explore additional Azure services](#)

App Service - WebApplication120171215025005
App Service Plan - WebApplication120171215025005Plan

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... **Create** **Cancel**

- 选择“服务”选项卡以创建新的数据库。
- 选择绿色的 + 图标以创建新的 SQL 数据库

Create App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft account
a.ricka00@gmail.com

Hosting (i) **Services**

Select any additional Azure resources your app will need

Show: Recommended

Resource Type

SQL Database
Scalable and managed database service for modern business-class apps +

Resources you've selected and configured

Resource Type		
 WebApplication120171215025005Plan App Service Plan	⚙	✖

If you have removed your spending limit or you are using Pay as You Go, there may be monetary impact if you provision additional resources.
[Learn More](#)

Export... **Create** **Cancel**

- 在“配置 SQL 数据库”对话框中选择“新建...”以创建新的数据库。



Configure SQL Database

Create a SQL Database in your subscription for storing data used by your application.

SQL Server

Administrator Username

Administrator Password

Database Name

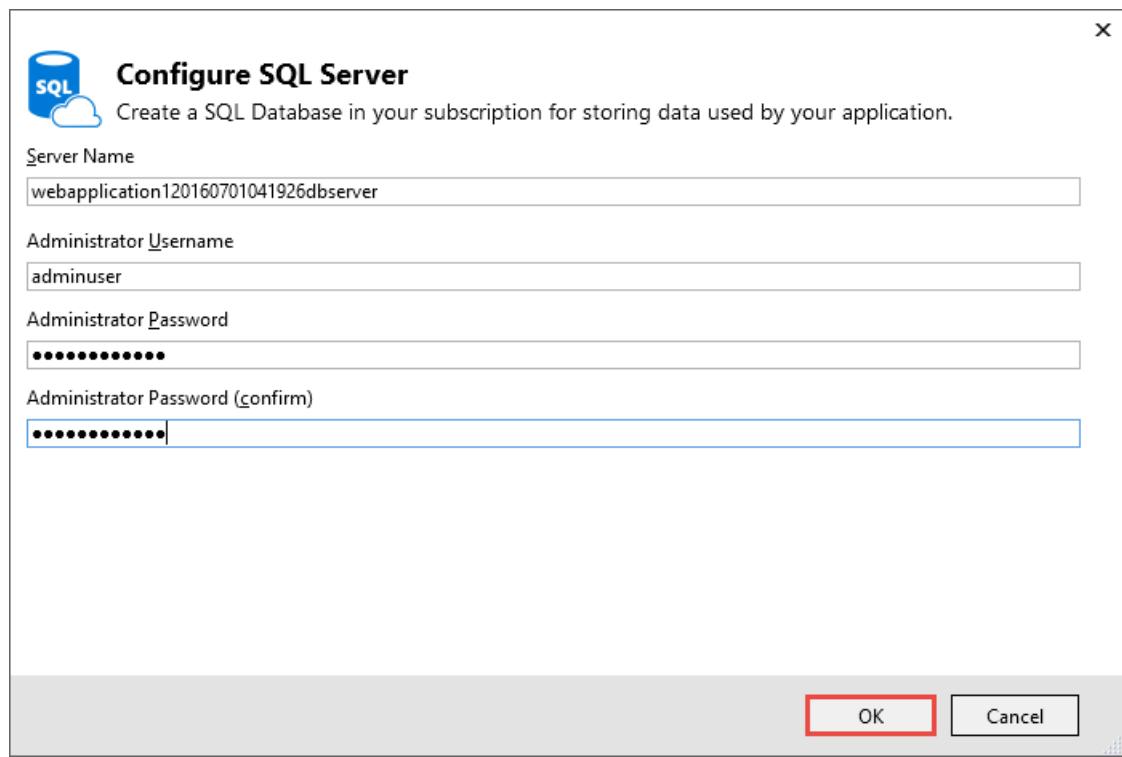
Connection String Name

“配置 SQL Server”对话框随即出现。

- 输入管理员用户名和密码，然后选择“确定”。可保留默认的“服务器名称”。

注意

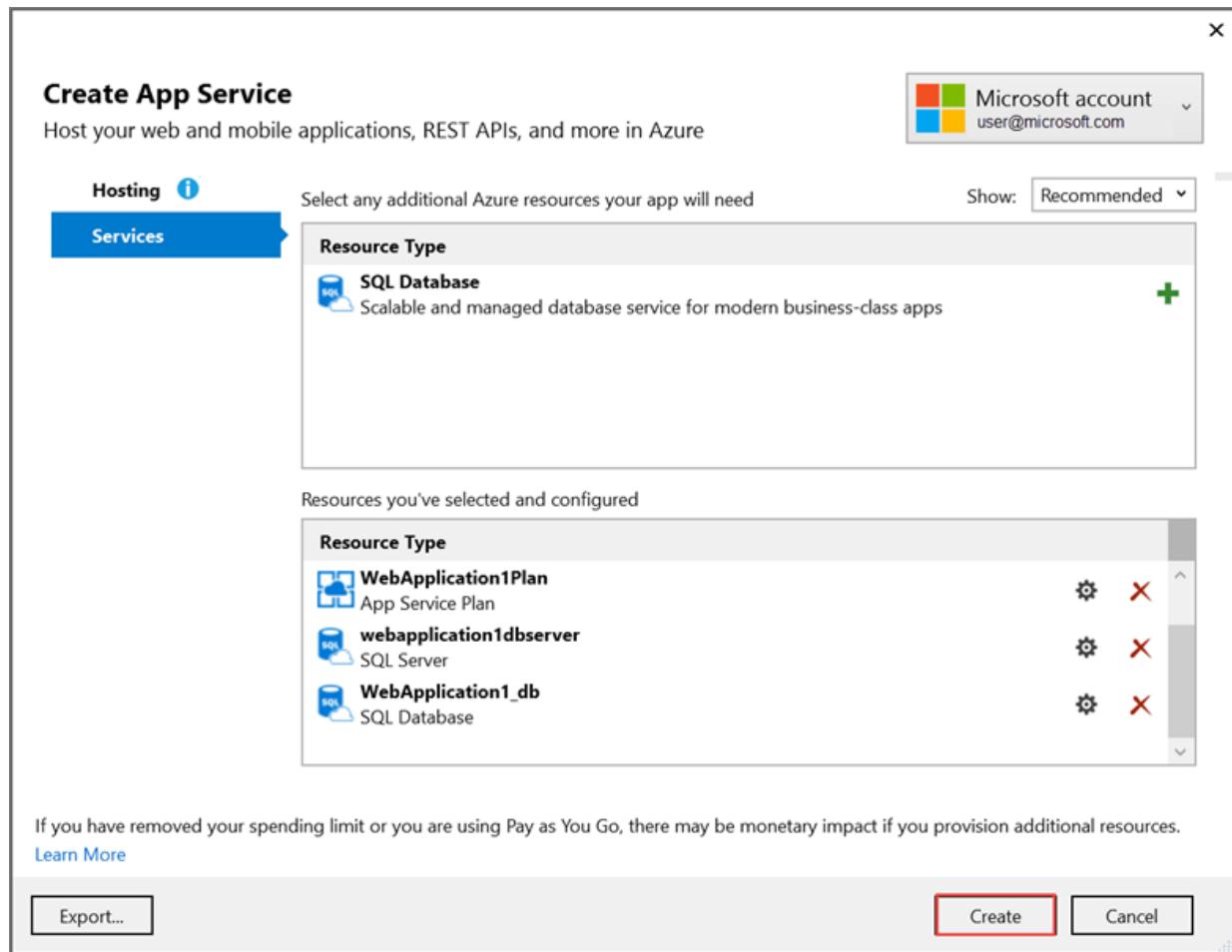
不可使用“admin”作为管理员用户名。



- 选择“确定”。

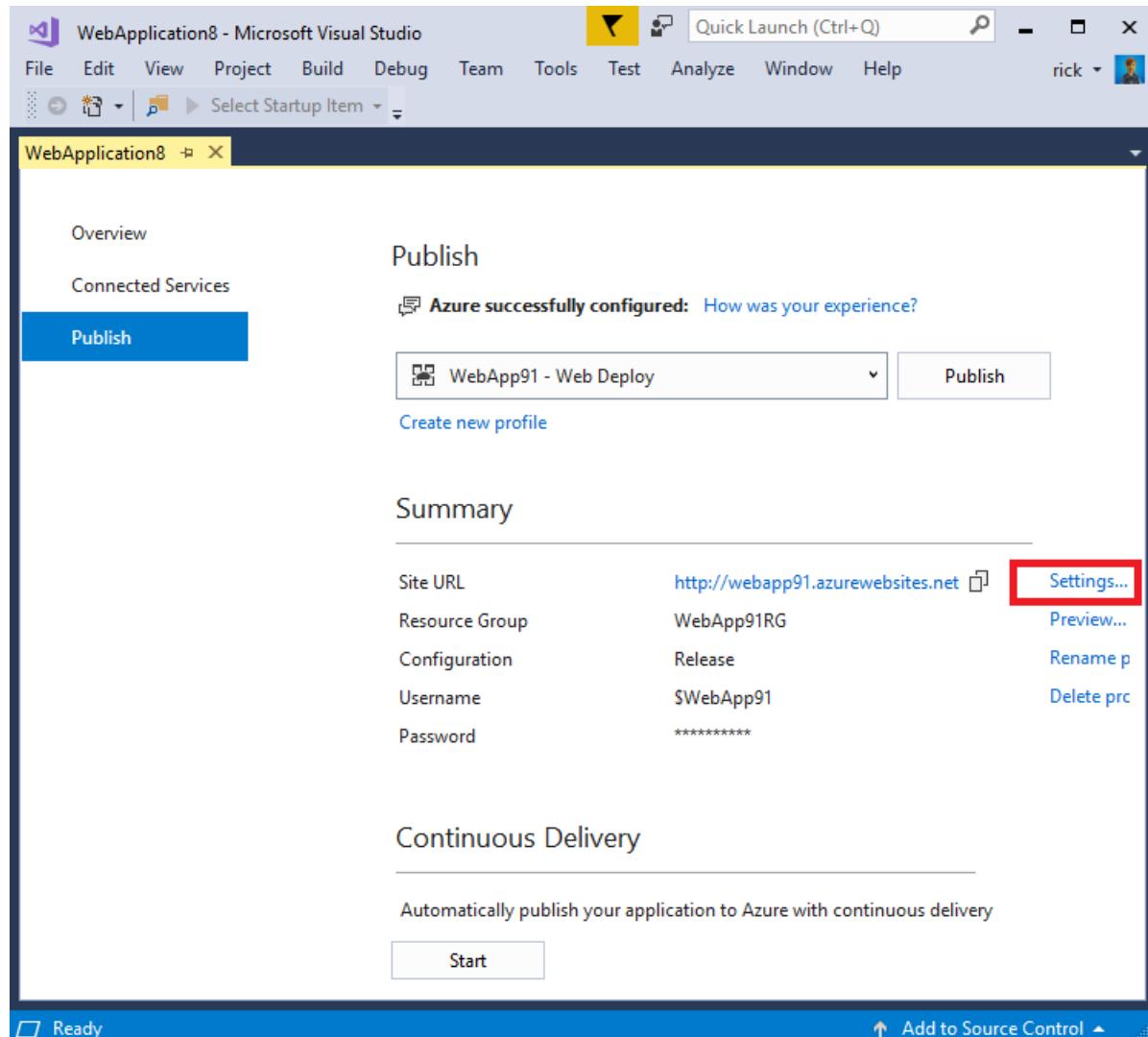
Visual Studio 将返回到“创建应用服务”对话框。

- 选择“创建应用服务”对话框上的“创建”。



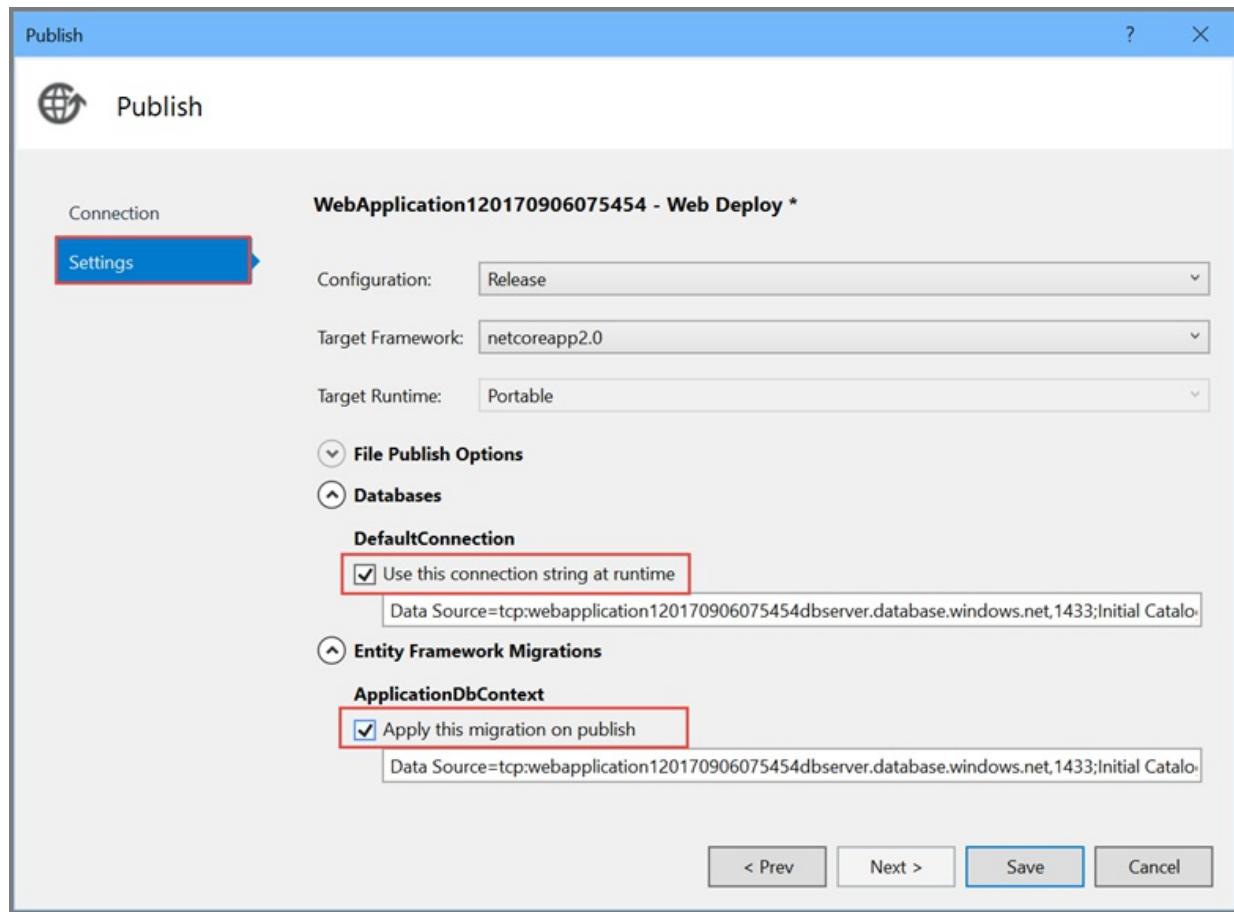
Visual Studio 在 Azure 上创建 Web 应用和 SQL Server。此步骤可能需要几分钟。有关创建的资源的信息，请参阅[其他资源](#)。

部署完成时，选择“设置”：



在“发布”对话框的“设置”页面上：

- 展开“数据库”并选中“在运行时使用此连接字符串”。
- 展开“Entity Framework 迁移”并选中“在发布时应用此迁移”。
- 选择“保存”。Visual Studio 将返回到“发布”对话框。



单击“发布”。Visual Studio 将应用发布到 Azure。部署完成时，应用在浏览器中打开。

在 Azure 中测试应用

- 测试“关于”和“联系”链接
- 注册新用户

The screenshot shows a web browser window with the following details:

- Title Bar:** Register - WebApplication1
- Address Bar:** localhost:44308/Account/Register
- Header:** To see favorites here, select then , and drag to the Favorites Bar folder. Or import from another browser. [Import favorites](#)
- Navigation:** Home, About, Contact
- User Options:** Register, Log in
- Main Content:**

Register

Create a new account.

Email: user@microsoft.co

Password:

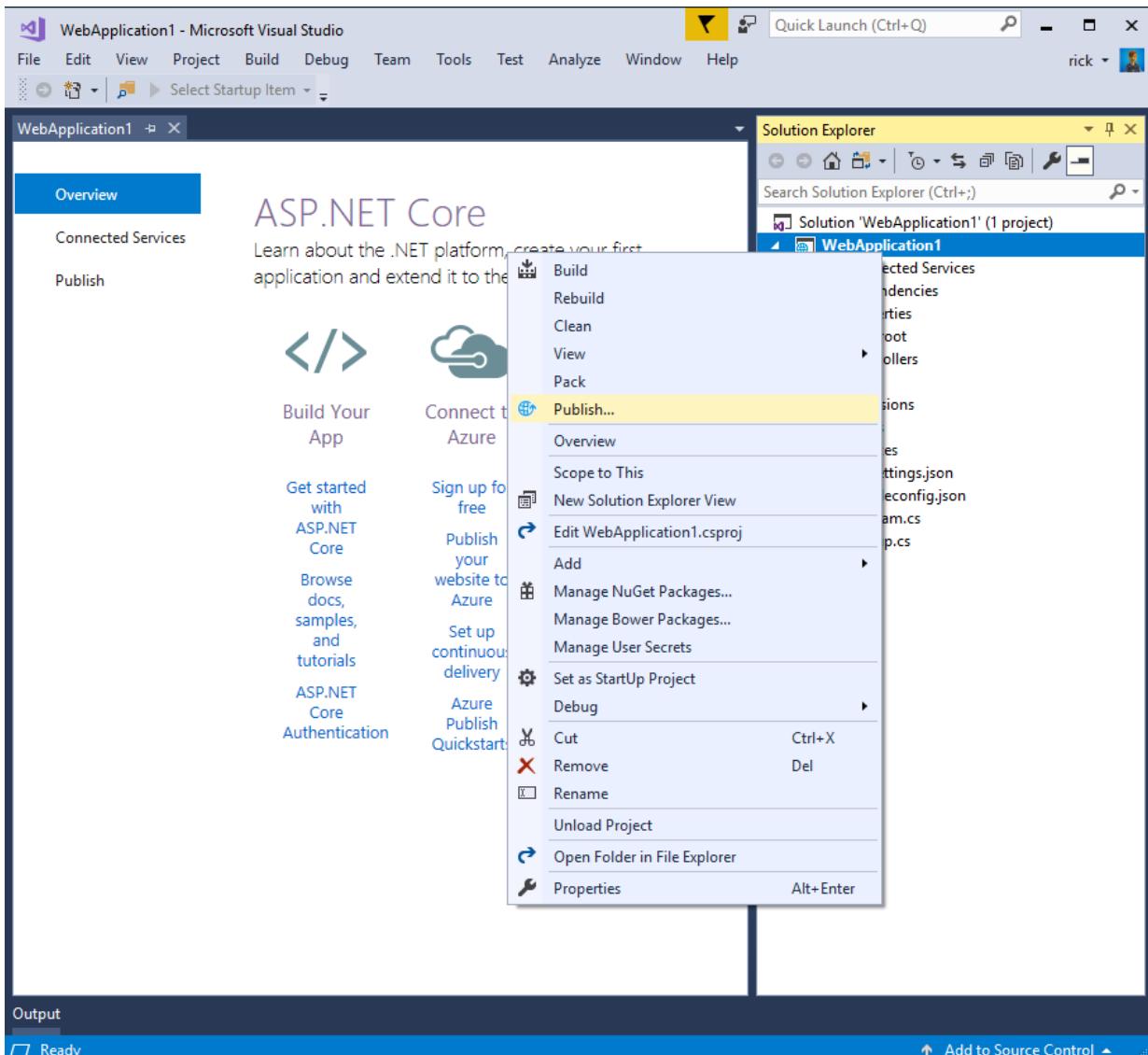
Confirm password:

Buttons:

Page Footer: © 2017 - WebApplication1

更新应用

- 编辑“Pages/About.cshtml”Razor 页面并更改其内容。例如，可以将段落修改为显示“Hello ASP.NET Core!”：`![code-html]About`
- 右键单击项目，然后再次选择“发布...”。



- 应用发布后，验证所做的更改在 Azure 上是否可用。

The screenshot shows a web browser window with the title bar "About - WebApplication1" and the URL "http://webapplication1220170908114215.azurewebsites.net/About". The page content includes a navigation bar with links for Home, About, and Contact, and buttons for Register and Log in. The main content area has a heading "About" and the sub-heading "Your application description page.". Below this, there is a red-bordered text box containing the message "Hello ASP.NET Core!". At the bottom, there is a copyright notice: "© 2017 - WebApplication2".

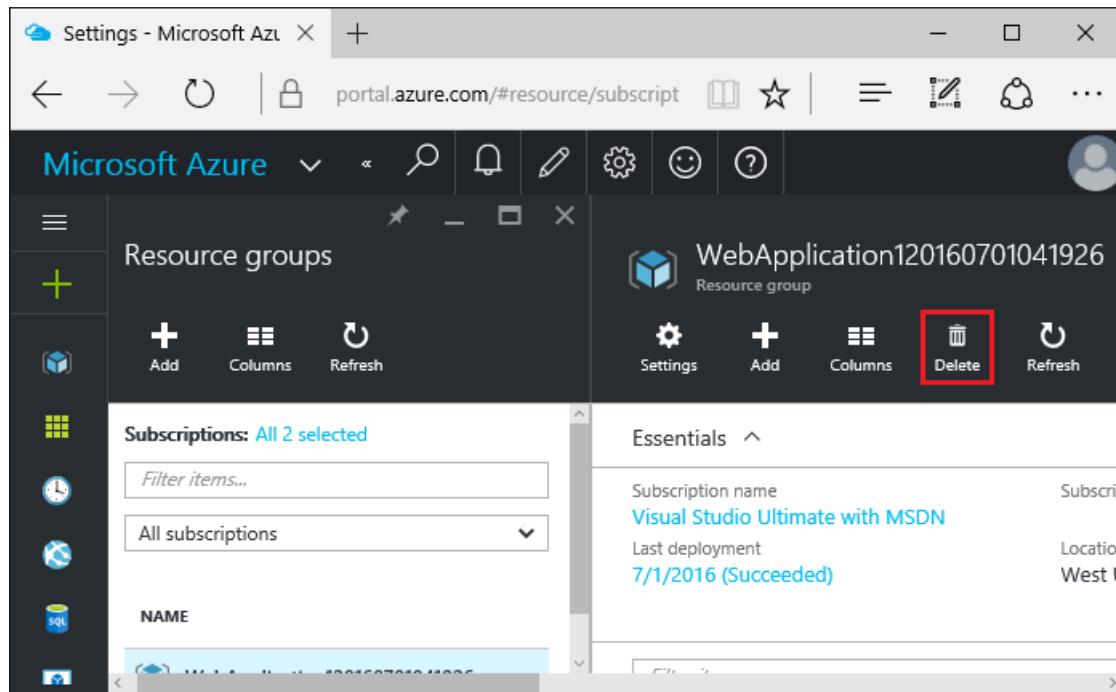
清理

完成应用测试后，转到 [Azure 门户](#) 并删除该应用。

- 选择“资源组”，然后选择所创建的资源组。

The screenshot shows the Microsoft Azure portal interface. The left sidebar menu is visible with items like "Resource groups", "All resources", "Recent", "App Services", "SQL databases", and "Virtual machines (classic)". The "Resource groups" item is highlighted with a red box. The main content area is titled "Resource groups" and shows a list of resource groups. A message at the top states "Subscriptions: All 2 selected". The list includes one item: "WebApplication120160701041926".

- 在“资源组”页面中，选择“删除”。



- 输入资源组的名称并选择“删除”。现已从 Azure 中删除了本教程中创建的应用和其他所有资源。

后续步骤

- 使用 Visual Studio 和 Git 持续部署到 Azure

其他资源

- [Azure 应用服务](#)
- [Azure 资源组](#)
- [Azure SQL 数据库](#)
- [对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)

使用命令行工具将 ASP.NET Core 应用发布到 Azure

2018/5/14 • 4 min to read • [Edit Online](#)

作者: Cam Soper

重要事项

要注意, 对于使用 **ASP.NET 核心 2.1 预览版本**

请参阅到 [Azure App Service 部署 ASP.NET Core 预览版](#)。

本教程将为你介绍如何使用命令行工具生成 ASP.NET Core 应用程序并将其部署到 Microsoft Azure App Service。完成后, 你将拥有一个在 ASP.NET MVC Core 中构建并作为 Azure App Service Web App 托管的 Web 应用程序。本教程是使用 Windows 命令行工具编写的, 但也可以应用于 macOS 和 Linux 环境。

在本教程中, 你将了解:

- 如何使用 Azure CLI 创建 Azure App Service 网站
- 如何使用 Git 命令行工具将 ASP.NET Core 应用程序部署到 Azure App Service

系统必备

若要完成本教程, 你需要:

- [Microsoft Azure 订阅](#)
- [.NET Core SDK 2.0 or later](#)
- [Git 命令行客户端](#)

创建 Web 应用程序

为 Web 应用程序创建新目录, 创建新的 ASP.NET Core MVC 应用程序, 然后在本地运行该网站。

- [Windows](#)
- [其他](#)

```
REM Create a new ASP.NET Core MVC application
dotnet new razor -o MyApplication

REM Change to the new directory that was just created
cd MyApplication

REM Run the application
dotnet run
```

```
C:\ Command Prompt - dotnet run

C:\>dotnet new razor -o MyApplication
The template "ASP.NET Core Web App" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/template-3pn for details.

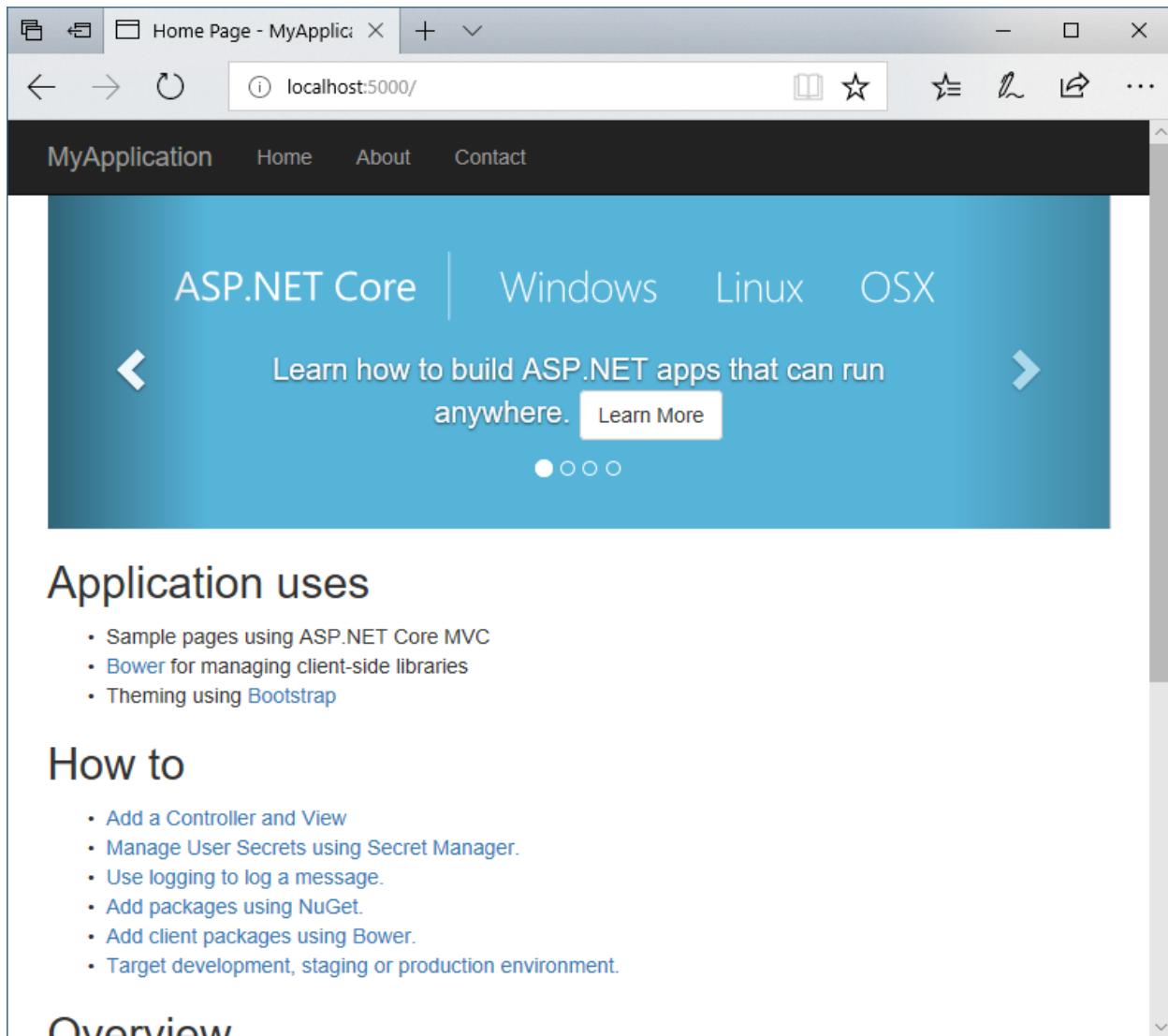
Processing post-creation actions...
Running 'dotnet restore' on MyApplication\MyApplication.csproj...
Restoring packages for C:\MyApplication\MyApplication.csproj...
Restore completed in 42.37 ms for C:\MyApplication\MyApplication.csproj.
Generating MSBuild file C:\MyApplication\obj\MyApplication.csproj.nuget.g.props.
Generating MSBuild file C:\MyApplication\obj\MyApplication.csproj.nuget.g.targets.
Restore completed in 1.5 sec for C:\MyApplication\MyApplication.csproj.

Restore succeeded.

C:\>cd MyApplication

C:\MyApplication>dotnet run
Hosting environment: Production
Content root path: C:\MyApplication
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

通过浏览器到 <http://localhost:5000> 测试应用程序。



创建 Azure App Service 实例

使用 [Azure Cloud Shell](#), 创建资源组、App Service 计划和 App Service Web 应用。

```
# Generate a unique Web App name
let randomNum=$RANDOM*$RANDOM
webappname=tutorialApp$randomNum

# Create the DotNetAzureTutorial resource group
az group create --name DotNetAzureTutorial --location EastUS

# Create an App Service plan.
az appservice plan create --name $webappname --resource-group DotNetAzureTutorial --sku FREE

# Create the Web App
az webapp create --name $webappname --resource-group DotNetAzureTutorial --plan $webappname
```

在部署之前，使用以下命令设置帐户级部署凭据：

```
az webapp deployment user set --user-name <desired user name> --password <desired password>
```

需要使用部署 URL 来使用 Git 部署应用程序。检索类似如下的 URL。

```
az webapp deployment source config-local-git -n $webappname -g DotNetAzureTutorial --query [url] -o tsv
```

请注意以 `.git` 结尾的已显示 URL。它在下一步中使用。

使用 Git 部署应用程序

你已准备好使用 Git 从本地计算机部署。

注意

忽略 Git 中任何关于行尾的警告是安全的。

- [Windows](#)
- [其他](#)

```
REM Initialize the local Git repository
git init

REM Add the contents of the working directory to the repo
git add --all

REM Commit the changes to the local repo
git commit -a -m "Initial commit"

REM Add the URL as a Git remote repository
git remote add azure <THE GIT URL YOU NOTED EARLIER>

REM Push the local repository to the remote
git push azure master
```

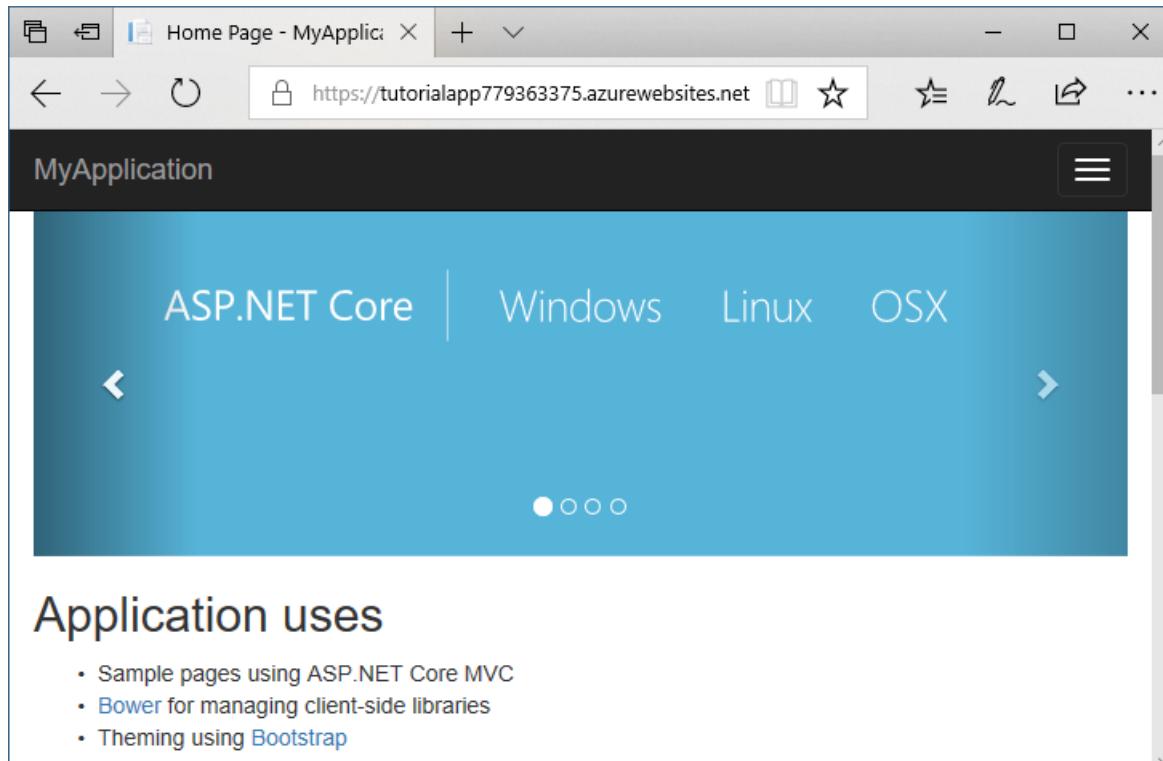
Git 提示前面设置的部署凭据。身份验证后，将应用程序推送到远程位置，然后生成并部署。

```
C:\ Command Prompt
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.css.map'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.min.css'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\css\bootstrap.min.css.map'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicon-halflings-regular.eot'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicon-halflings-regular.svg'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicon-halflings-regular.ttf'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicon-halflings-regular.woff'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\fonts\glyphicon-halflings-regular.woff2'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\bootstrap.js'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\bootstrap.min.js'
remote: Copying file: 'wwwroot\lib\bootstrap\dist\js\npm.js'
remote: Copying file: 'wwwroot\lib\jquery\.bower.json'
remote: Copying file: 'wwwroot\lib\jquery\LICENSE.txt'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.js'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.min.js'
remote: Copying file: 'wwwroot\lib\jquery\dist\jquery.min.map'
remote: Copying file: 'wwwroot\lib\jquery-validation\.bower.json'
remote: Copying file: 'wwwroot\lib\jquery-validation\LICENSE.md'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\additional-methods.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\additional-methods.min.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\jquery.validate.js'
remote: Copying file: 'wwwroot\lib\jquery-validation\dist\jquery.validate.min.js'
remote: Omitting next output lines...
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Deployment successful.
To https://tutorialapp779363375.scm.azurewebsites.net/tutorialApp779363375.git
 * [new branch]      master -> master
C:\MyApplication>
```

测试应用程序

通过浏览到 <https://<web app name>.azurewebsites.net> 测试应用程序。若要在 Cloud Shell(或 Azure CLI)中显示地址, 请使用以下方法:

```
az webapp show -n $webappname -g DotNetAzureTutorial --query defaultHostName -o tsv
```



清理

完成测试应用并检查代码和资源后, 通过删除资源组来删除 Web 应用并制定计划。

```
az group delete -n DotNetAzureTutorial
```

后续步骤

在本教程中，你将了解：

- 如何使用 Azure CLI 创建 Azure App Service 网站
- 如何使用 Git 命令行工具将 ASP.NET Core 应用程序部署到 Azure App Service

接下来，了解如何使用命令行来部署使用 CosmosDB 的现有 Web 应用。

[通过 .NET Core 从命令行部署到 Azure](#)

使用 Visual Studio 的 Azure 和 ASP.NET Core 使用 Git 进行持续部署

2018/4/10 • 8 min to read • [Edit Online](#)

作者: Erik Reitan

重要事项

要注意, 对于使用 **ASP.NET 核心 2.1 预览版本**

请参阅[到 Azure App Service 部署 ASP.NET Core 预览版。](#)

本教程演示如何创建使用 Visual Studio 的 ASP.NET 核心 web 应用并将其从 Visual Studio 部署到 Azure App Service, 使用连续部署。

另请参阅 [Use VSTS to Build and Publish to an Azure Web App with Continuous Deployment](#)(使用 VSTS 生成并通过持续部署发布到 Azure Web 应用), 其中演示如何使用 Visual Studio Team Services 为 [Azure App Service](#) 配置持续交付(CD)工作流。在 Team Services 的 azure 持续交付简化了可靠的部署管道的设置以发布托管在 Azure App Service 中的应用程序的更新。可以从 Azure 门户中生成、运行测试, 将部署到过渡槽中, 以及然后部署到生产环境中配置管道。

注意

若要完成本教程, Microsoft Azure 帐户是必需的。若要获取帐户, [激活 MSDN 订户权益](#)或[注册一个免费试用版](#)。

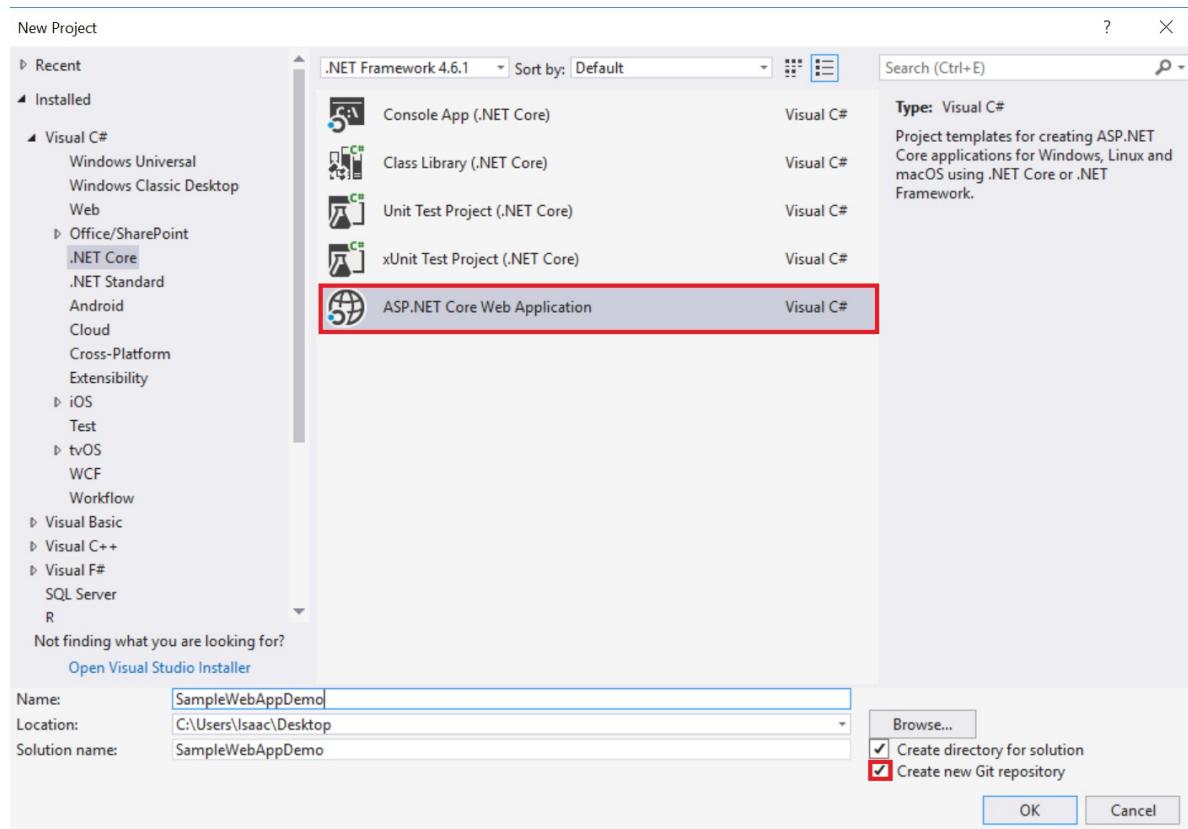
系统必备

本教程假定安装以下软件:

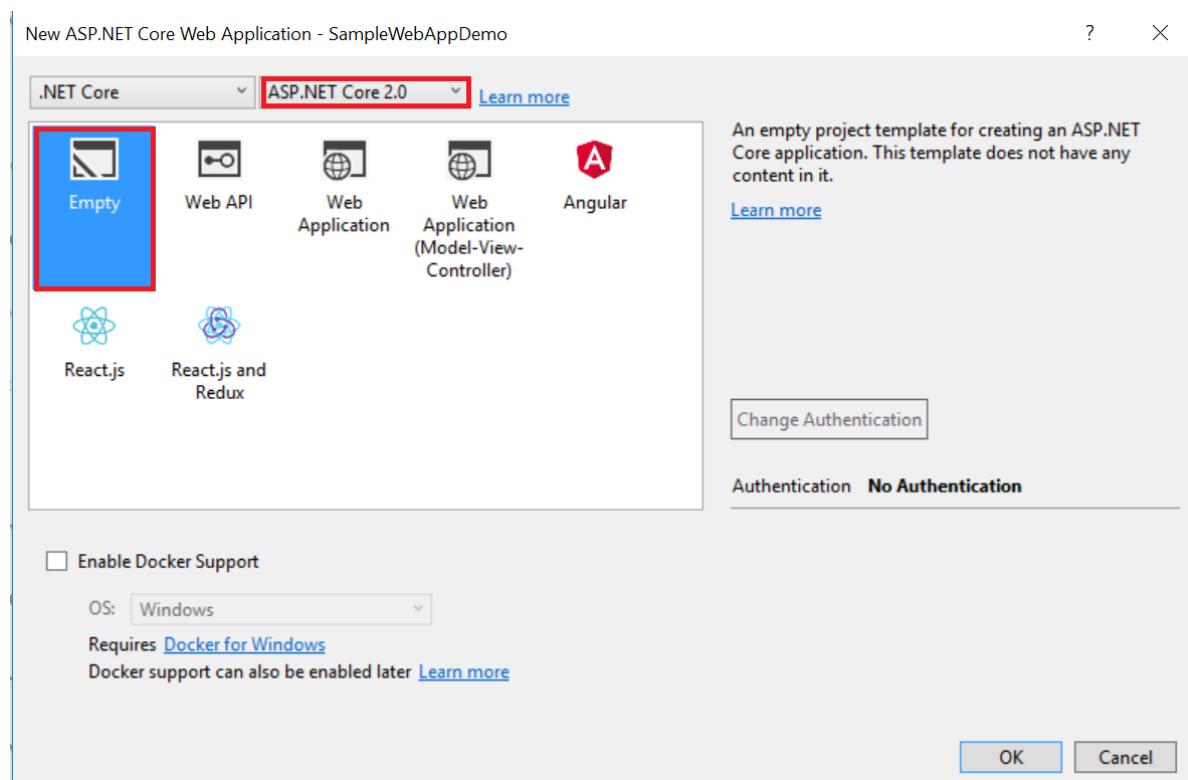
- [Visual Studio](#)
- [.NET Core SDK 2.0 or later](#)
- 用于 Windows 的 [Git](#)

创建 ASP.NET Core Web 应用

1. 启动 Visual Studio。
2. 从“文件”菜单中选择“新建” > “项目”。
3. 选择**ASP.NET Core Web 应用程序**项目模板。它出现在“已安装” > “模板” > “Visual C#” > “.NET Core”下。将项目命名为 `SampleWebAppDemo`。选择**创建新 Git 存储库**选项, 然后单击**确定**。



4. 在“新建 ASP.NET Core 项目”对话框中，选择 ASP.NET Core 的“空”模板，然后单击“确定”。



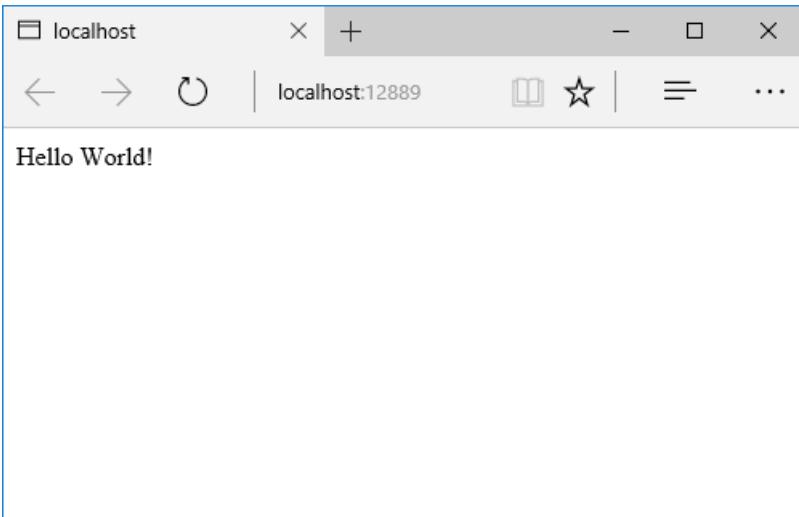
注意

.NET 核心的最新版本为 2.0。

本地运行 Web 应用

1. Visual Studio 完成创建应用后，请选择“调试” > “启动调试”以运行该应用。作为替代方法，按 F5。

可能需要一些时间对 Visual Studio 和新应用进行初始化。完成后，浏览器将显示正在运行的应用。

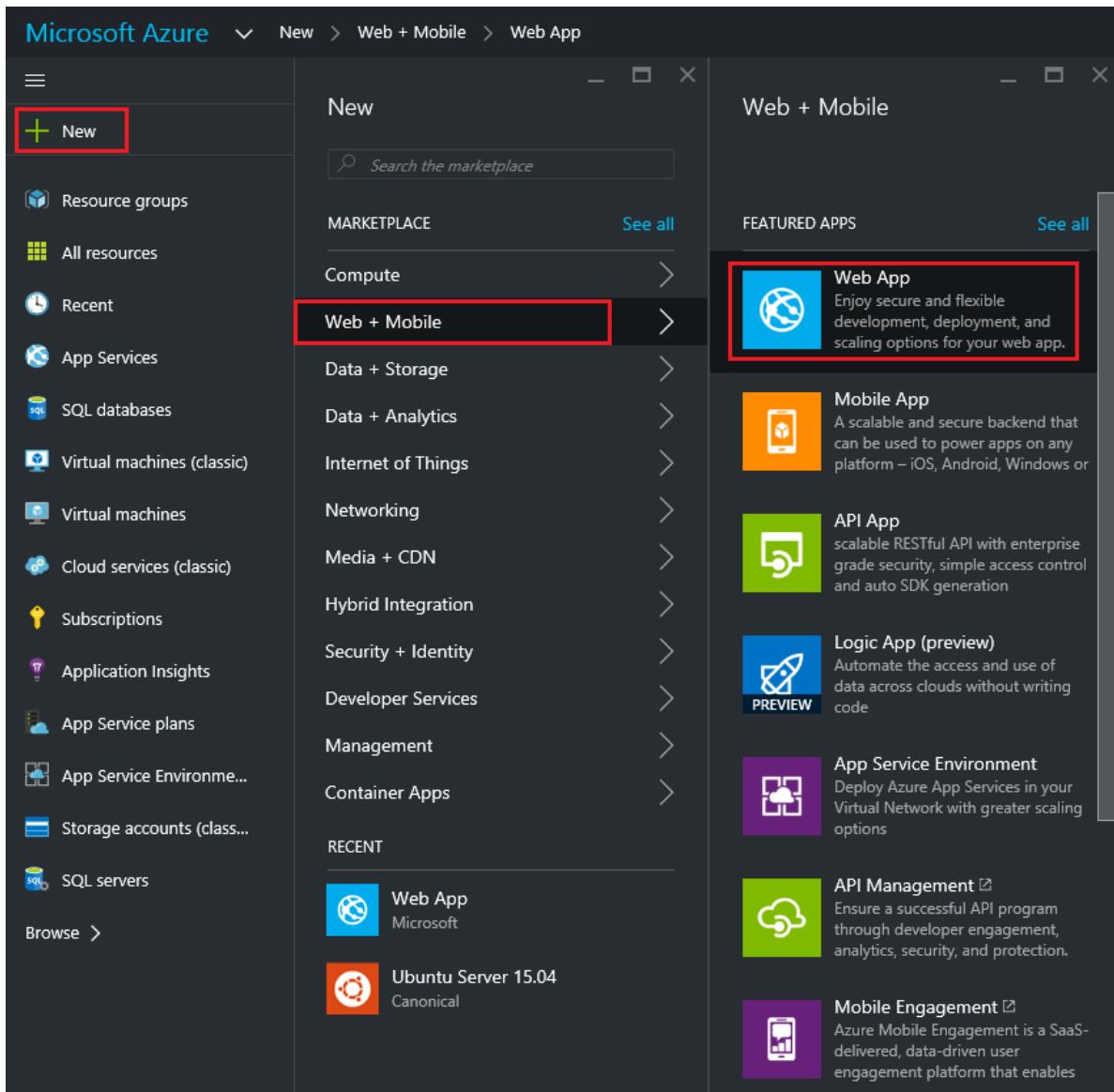


- 之后查看正在运行的 Web 应用，关闭浏览器，然后在 Visual Studio 以停止应用的工具栏中选择“停止调试”图标。

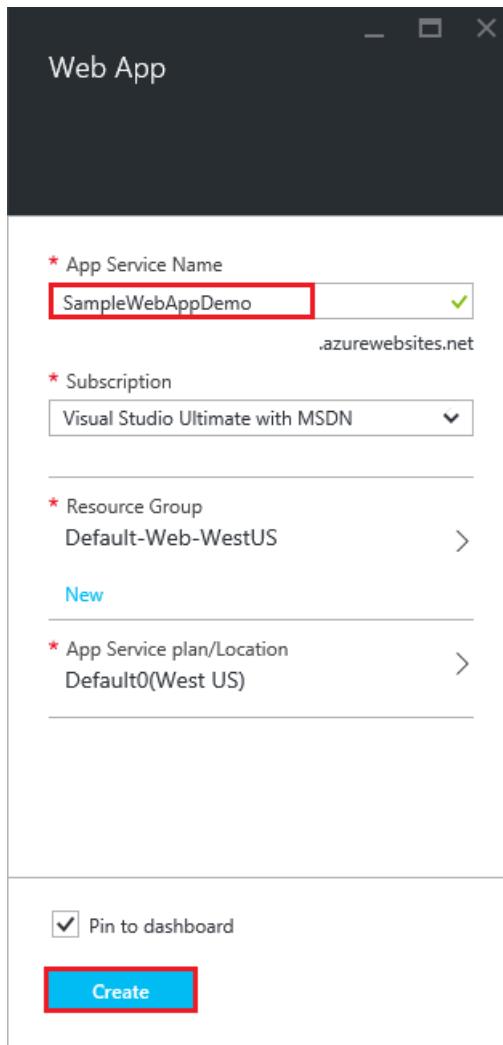
在 Azure 门户中创建 Web 应用

以下步骤在 Azure 门户中创建 web 应用：

- 登录到[Azure 门户](#)。
- 选择[新建](#)在左上角的门户界面。
- 选择**Web + 移动 > Web 应用**。



4. 在“Web 应用”边栏选项卡中，输入“应用服务名称”的唯一值。



注意

App Service 名称名称必须唯一。门户执行此规则时提供的名称。如果提供不同的值，将替代的每个匹配项的值**SampleWebAppDemo**在本教程。

另外，在“Web 应用”边栏选项卡中，选择现有“应用服务计划/位置”或新建一个。如果创建新的计划，请选择定价层、位置和其他选项。App Service 计划的详细信息，请参阅[Azure App Service 计划深入概述](#)。

5. 选择“创建”。Azure 将设置并启动 web 应用。

Microsoft Azure SampleWebAppDemo01

SampleWebAppDemo01 Web app

Resource group: Default-Web-WestUS

Status: Running

Location: West US

Subscription name: Visual Studio Ultimate with MSDN

Subscription id:

URL: <http://samplewebapppdemo01.azurewebsites...>

App Service plan/pricing tier: Default0 (Free)

FTP/Deployment username: SampleWebAppDemo01\erikre01

FTP hostname: ftp://waws-prod-bay-005.ftp.azurewebsites...

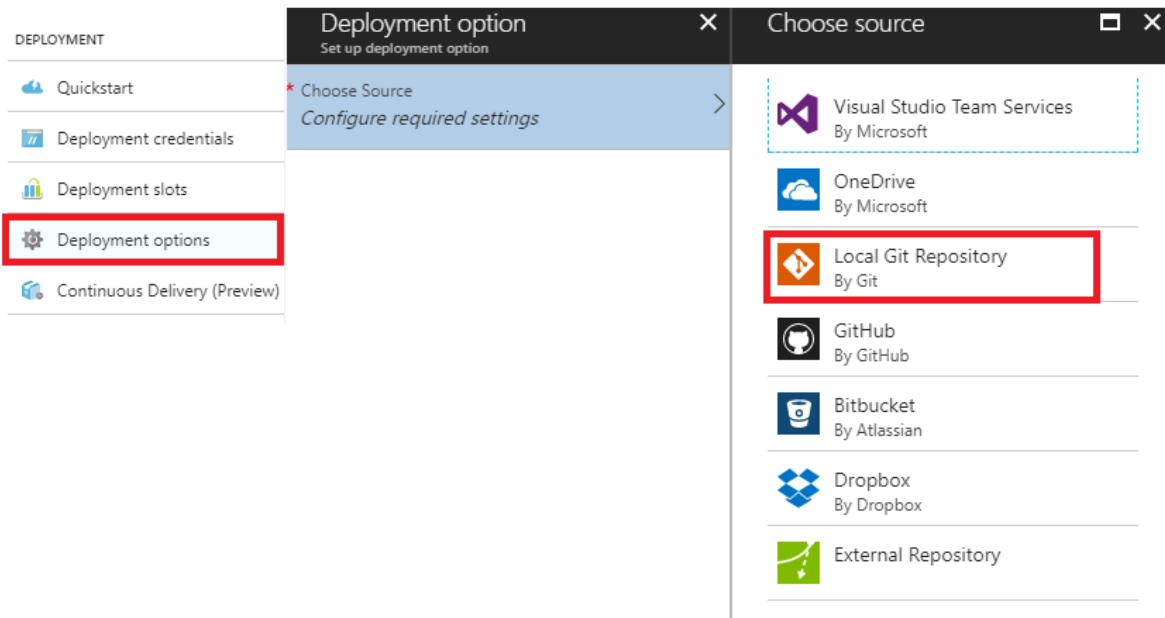
FTPS hostname: https://waws-prod-bay-005.ftp.azurewebsites...

All settings →

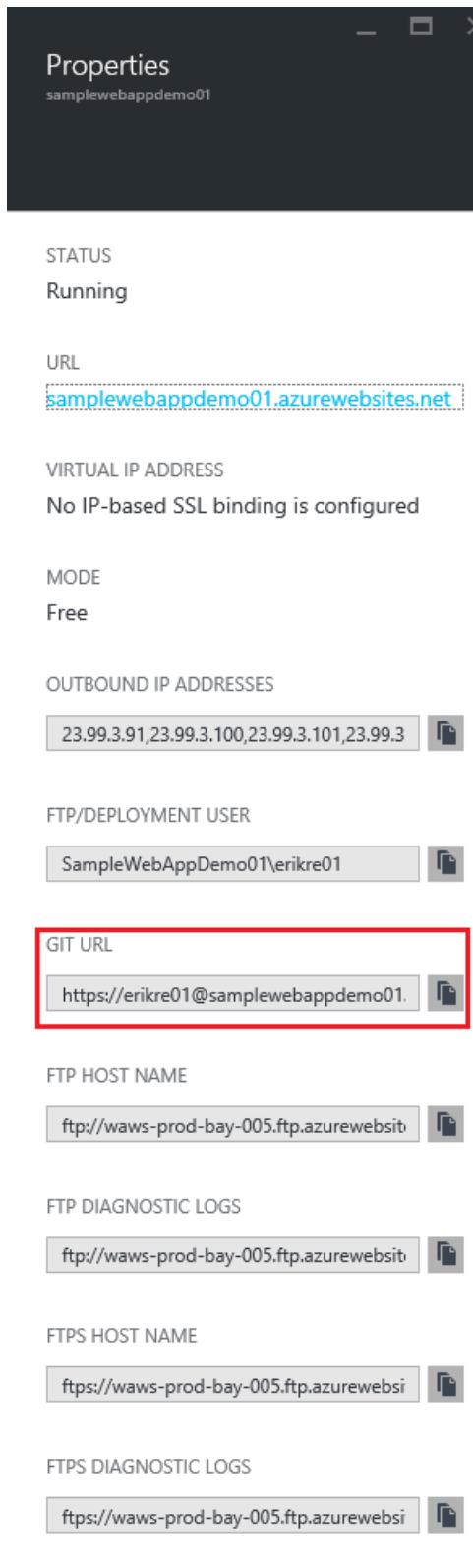
为新 Web 应用启用 Git 发布

Git 是一个分布式的版本控制系统，可用来部署 Azure App Service web 应用。Web 应用程序代码存储在本地 Git 存储库，并且该代码通过将推送到远程存储库部署到 Azure。

1. 登录到[Azure 门户](#)。
2. 选择[应用程序服务](#)以查看与 Azure 订阅关联的应用程序服务的列表。
3. 选择在本教程的上一节中创建的 web 应用。
4. 在“部署”边栏选项卡中，选择“部署选项” > “选择源” > “本地 Git 存储库”。



5. 选择“确定”。
6. 如果以前尚未设置部署凭据用于发布 web 应用或其他 App Service 应用，它们现在设置：
 - 选择设置 > 部署凭据。设置部署凭据显示边栏选项卡。
 - 创建用户名和密码。设置 Git 时，请保存以供将来使用的密码。
 - 选择“保存”。
7. 在**Web 应用**边栏选项卡，选择设置 > 属性。要将部署到远程 Git 存储库的 URL 将显示在**GIT URL**。
8. 复制“GIT URL”的值，稍后将在本教程中用到。

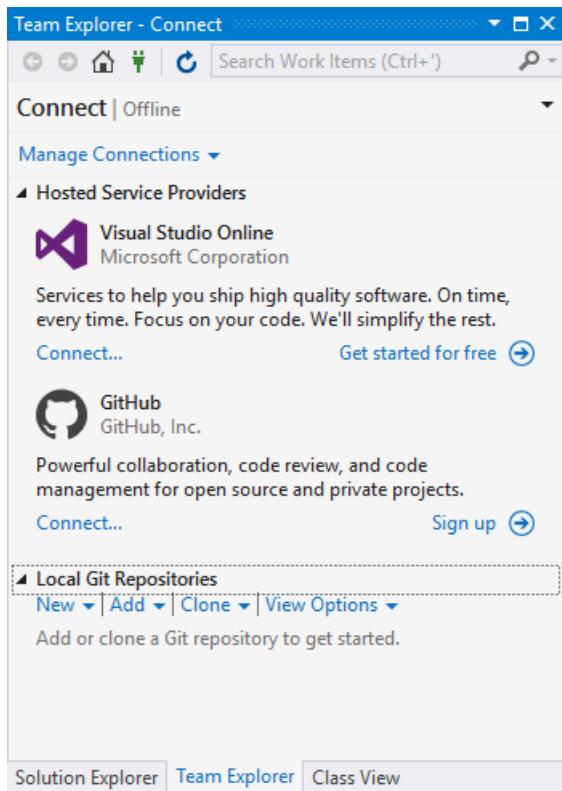


将 web 应用发布到 Azure App Service

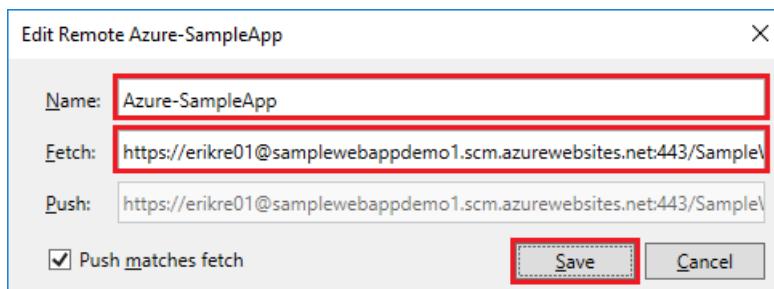
在本部分中，创建使用 Visual Studio 和推送从该存储库到 Azure 来部署 web 应用的本地 Git 存储库。涉及到的步骤如下：

- 添加使用 GIT URL 值，因此可以将本地存储库部署到 Azure 的远程存储库设置。
- 提交项目进行的更改。
- 在 Azure 上，项目进行的更改从本地存储库推送到远程存储库。

1. 在“解决方案资源管理器”中右键单击“解决方案 ‘SampleWebAppDemo’”并选择“提交”。团队资源管理器显示。



2. 在“团队资源管理器”中，选择“主页”(主页图标)>“设置”>“存储库设置”。
3. 在远程网站部分存储库设置，选择添加。添加远程对话框随即显示。
4. 将远程的“名称”设置为“Azure-SampleApp”。
5. 将的值设置提取到**Git URL**，在本教程前面部分中从 Azure 复制。请注意，此 URL 是以 .git 结尾的。

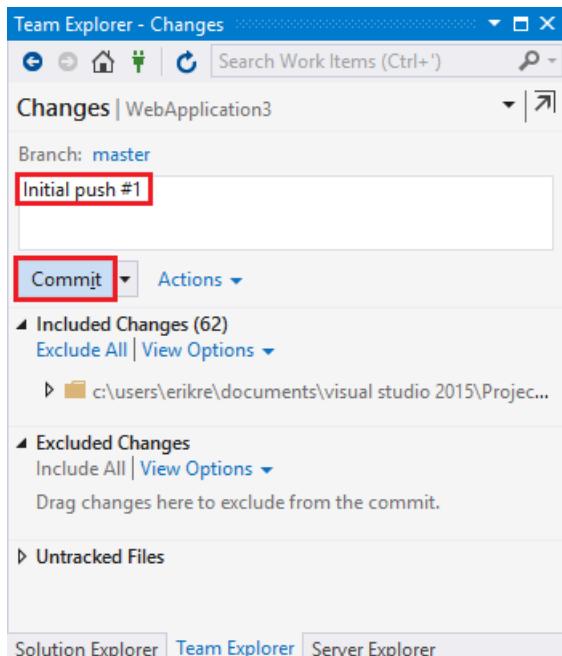


注意

作为替代方法，指定从远程存储库命令窗口通过打开命令窗口、将更改为项目目录，然后输入命令。示例：

```
git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp.git
```

6. 选择“主页”(主页图标)>“设置”>“全局设置”。确认设置的名称和电子邮件地址。选择**更新必要情况**。
7. 选择“主页”>“更改”以返回到“更改”视图。
8. 输入提交消息，如**初始推送 #1**和选择**提交**。此操作将创建**提交本地**。



注意

作为替代方法中的更改提交命令窗口通过打开命令窗口、将更改为项目目录，然后输入 git 命令。示例：

```
git add .  
git commit -am "Initial Push #1"
```

9. 选择“主页”>“同步”>“操作”>“打开命令提示符”。命令提示符打开的项目目录。

10. 在命令窗口中输入以下命令：

```
git push -u Azure-SampleApp master
```

11. 输入 Azure 部署凭据之前在 Azure 中创建的密码。

此命令启动将本地项目文件推送到 Azure 的过程。上面的命令的输出由部署成功消息结尾。

```
remote: Finished successfully.  
remote: Running post deployment command(s)...  
remote: Deployment successful.  
To https://username@samplewebappdemo01.scm.azurewebsites.net:443/SampleWebAppDemo01.git  
 * [new branch]      master -> master  
Branch master set up to track remote branch master from Azure-SampleApp.
```

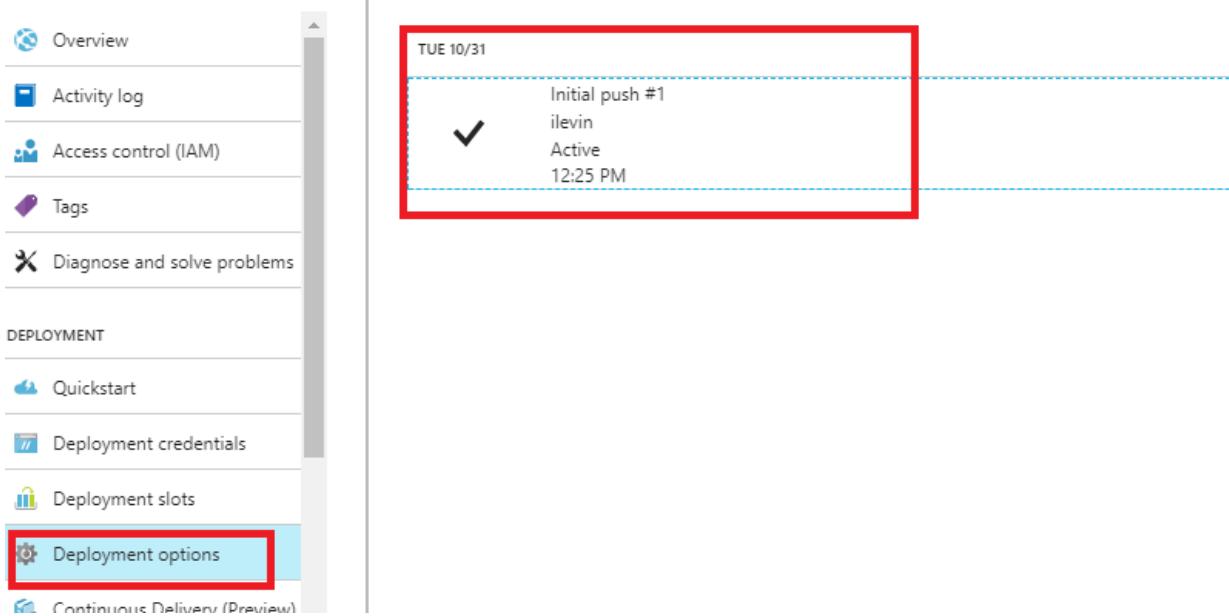
注意

如果需要在项目上的协作，请考虑在推送到 GitHub 之前将推送到 Azure。

验证活动的部署

验证与本地环境到 Azure web 应用程序传输成功。

在 Azure 门户，选择 web 应用。选择部署 > 部署选项。



在 Azure 中运行此应用

现在, web 应用部署到 Azure, 运行该应用。

这可以通过两种方式实现:

- 在 Azure 门户中, 找到 web 应用的 web 应用边栏选项卡。选择浏览在默认浏览器中查看应用。
- 打开浏览器并输入 web 应用的 URL。示例: `http://SampleWebAppDemo.azurewebsites.net`

更新 web 应用程序并将重新发布

对本地代码进行更改后, 重新发布:

1. 在 Visual Studio 的“解决方案资源管理器”中, 打开 Startup.cs 文件。
2. 在 `Configure` 方法中, 修改 `Response.WriteAsync` 方法, 使它显示以下内容:

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```
3. 保存对 `Startup.cs`。
4. 在“解决方案资源管理器”中右键单击“解决方案 ‘SampleWebAppDemo’”并选择“提交”。团队资源管理器显示。
5. 输入提交消息, 如 `Update #2`。
6. 按“提交”按钮以提交项目更改。
7. 选择“主页” > “同步” > “操作” > “推送”。

注意

作为替代方法, 将从更改推送命令窗口通过打开命令窗口、将更改为项目目录, 然后输入 git 命令。示例:

```
git push -u Azure-SampleApp master
```

在 Azure 中查看更新的 Web 应用

通过选择查看更新的 web 应用浏览从 web 应用边栏选项卡在 Azure 门户或通过打开浏览器并输入 web 应用的

URL。示例：<http://SampleWebAppDemo.azurewebsites.net>

其他资源

- [VSTS 用于生成并发布到 Azure Web 应用程序使用连续部署](#)
- [项目 Kudu](#)

解决在 Azure App Service 上的 ASP.NET 核心

2018/4/10 • 11 min to read • [Edit Online](#)

作者:Luke Latham

重要事项

要注意，对于使用 **ASP.NET 核心 2.1 预览版本**

请参阅到 [Azure App Service 部署 ASP.NET Core 预览版](#)。

本文说明了如何诊断 ASP.NET Core 应用使用 Azure App Service 的诊断工具的启动问题。有关其他故障排除建议，请参阅[Azure App Service 诊断概述](#)和[如何: 在 Azure App Service 中监视应用](#)Azure 文档中。

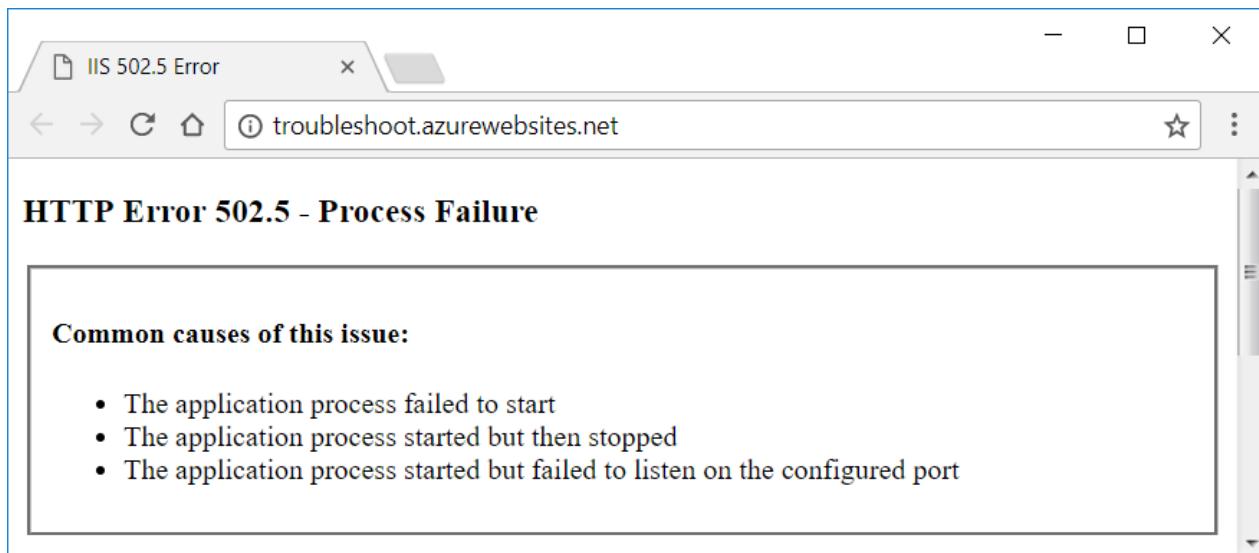
应用程序启动错误

502.5 进程故障

工作进程将失败。不启动应用程序。

[ASP.NET 核心模块](#)尝试启动工作进程，但它无法启动。经常检查应用程序事件日志可帮助解决这种类型的问题。访问日志中进行了说明[应用程序事件日志](#)部分。

502.5 进程失败配置错误的应用程序会导致工作进程失败，返回错误页：



500 内部服务器错误

启动该应用程序，但某个错误阻止在完成请求的服务器。

启动过程中或创建响应时，应用程序的代码中出现此错误。响应可能包含任何内容，或响应可能显示为500 内部服务器错误浏览器中。应用程序事件日志通常表明应用程序正常启动。从服务器的角度来看，这是正确的。应用程序未启动，但它无法生成有效的响应。[Kudu 控制台中运行此应用或启用 ASP.NET 核心模块 stdout 日志](#)以解决该问题。

连接重置

如果发送标头后，将发生错误，则服务器尝试发送太晚**500 内部服务器错误**发生错误时。响应的复杂对象的序列化期间发生错误时经常会出现这种情况。此类型的错误显示为连接重置客户端上的错误。[应用程序日志记录](#)可以帮助解决这些类型的错误。

默认启动限制

ASP.NET 核心模块配置了默认值 `startupTimeLimit` 的 120 秒。时保留为默认值，应用可能需要最多两分钟，以启动之前模块记录进程失败。有关模块的配置的信息，请参阅 [aspNetCore 元素的特性的](#)。

应用程序启动错误疑难解答

应用程序事件日志

若要访问应用程序事件日志，请使用 **诊断并解决问题** 在 Azure 门户中的边栏选项卡：

1. 在 Azure 门户中，打开应用的边栏选项卡中 **应用程序服务** 边栏选项卡。
2. 选择 **诊断并解决问题** 边栏选项卡。
3. 下选择问题类别，选择 **Web 应用程序** 向下按钮。
4. 下建议解决方案，打开的窗格打开 **应用程序事件日志**。选择 **打开应用程序事件日志** 按钮。
5. 检查提供的最新错误 *IIS AspNetCoreModule* 中源列。

除了使用 **诊断并解决问题** 边栏选项卡是检查应用程序事件日志文件直接使用 [Kudu](#)：

1. 选择高级工具中的边栏选项卡 **开发工具区域**。选择转→按钮。Kudu 控制台打开新浏览器选项卡或窗口中。
2. 使用的页上，顶部导航栏打开调试控制台和选择 **CMD**。
3. 打开 **LogFiles** 文件夹。
4. 选择铅笔图标旁边 *eventlog.xml* 文件。
5. 检查日志。滚动到底部的日志，以查看最新的事件。

Kudu 控制台中运行此应用

许多启动错误未生成应用程序事件日志中的有用信息。你可以在运行应用程序 [Kudu](#) 远程执行控制台，以发现错误：

1. 选择高级工具中的边栏选项卡 **开发工具区域**。选择转→按钮。Kudu 控制台打开新浏览器选项卡或窗口中。
2. 使用的页上，顶部导航栏打开调试控制台和选择 **CMD**。
3. 打开的文件夹的路径 **站点 > wwwroot**。
4. 在控制台中，通过执行应用程序的程序集运行该应用。
 - 如果在应用程序处于 **framework 相关部署**，运行具有的应用程序的程序集 `dotnet.exe`。在下面的命令中，应用程序的程序集的名称替换 `<assembly_name>`：`dotnet .\<assembly_name>.dll`
 - 如果在应用程序处于 **独立的部署** 中运行应用程序的可执行文件。在下面的命令中，应用程序的程序集的名称替换 `<assembly_name>`：`<assembly_name>.exe`
5. 控制台应用程序中，显示任何错误，从输出传送到 Kudu 控制台。

ASP.NET 核心模块 `stdout` 日志

ASP.NET 核心模块 `stdout` 日志通常记录找不到应用程序事件日志中的有用的错误消息。用于启用和查看 `stdout` 日志：

1. 导航到 **诊断并解决问题** 在 Azure 门户中的边栏选项卡。
2. 下选择问题类别，选择 **Web 应用程序** 向下按钮。
3. 下建议解决方案 > **启用 Stdout 日志重定向**，选择该按钮 **打开 Kudu 控制台编辑 Web.Config**。
4. 在 Kudu **Diagnostic** 控制台，打开的文件夹的路径 **站点 > wwwroot**。向下滚动到显示 `web.config` 列表底部的文件。
5. 单击铅笔图标旁边 `web.config` 文件。
6. 设置 `stdoutEnabled` 到 `true` 和更改 `stdoutLogFile` 路径：`\?\%home%\LogFiles\stdout`。
7. 选择 **保存** 以保存已更新 `web.config` 文件。
8. 向应用程序发出的请求。
9. 返回到 Azure 门户。选择高级工具中的边栏选项卡 **开发工具区域**。选择转→按钮。Kudu 控制台打开新浏览器

器选项卡或窗口中。

10. 使用的页上，顶部导航栏打开调试控制台和选择**CMD**。
11. 选择**LogFiles**文件夹。
12. 检查**已修改**列并选择铅笔图标以编辑 stdout 日志的最新的修改日期。
13. 在打开日志文件后，将显示错误。

重要提示！禁用完成故障排除时，日志记录 stdout。

1. 在 Kudu **Diagnostic** 控制台，则返回到路径站点 > **wwwroot**以显示**web.config**文件。打开**web.config**再次通过选择铅笔图标的文件。
2. 设置**stdoutLogEnabled**到 `false`。
3. 选择**保存**保存文件。

警告

若要禁用 stdout 日志可能会导致应用程序或服务器失败。日志文件大小或创建的日志文件数没有限制。仅使用日志记录以应用启动问题疑难解答的 stdout。

对于常规日志记录在 ASP.NET Core 应用程序在启动之后，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅[第三方日志记录提供程序](#)。

常见的启动错误

请参阅[ASP.NET 核心常见错误参考](#)。中的参考主题介绍了大部分常见防止应用程序启动的问题。

慢速或悬挂应用程序

当应用程序响应速度很慢或挂起的请求时，请参阅[在 Azure App Service 中进行故障排除慢速的 web 应用程序性能问题](#)用于调试的指导。

远程调试

请参见下面的主题：

- [远程调试的故障排除 Azure App Service 中使用 Visual Studio 中的 web 应用部分](#) (Azure 文档)
- [在 Visual Studio 2017 在 Azure 中的 IIS 上的远程调试 ASP.NET Core](#) (Visual Studio 文档)

Application Insights

[Application Insights](#)提供从托管在 Azure App Service, 包括错误日志记录和报告功能的应用程序的遥测。

Application Insights 只能针对在应用程序启动时应用的日志记录功能变得可用之后发生的错误报告。有关详细信息，请参阅[Application Insights for ASP.NET Core](#)。

监视边栏选项卡

监视边栏选项卡提供疑难解答的体验到本主题前面所述的方法的替代方法。这些边栏选项卡可以用于诊断 500 系列错误。

确认安装了 ASP.NET 核心扩展。如果未安装的扩展，请手动安装它们：

1. 在**开发工具**边栏选项卡部分中，选择**扩展**边栏选项卡。
2. **ASP.NET 核心扩展**应出现在列表中。
3. 如果未安装的扩展，选择**添加**按钮。
4. 选择**ASP.NET 核心扩展**从列表中。

5. 选择**确定**以接受法律条款。
6. 选择**确定上将扩展添加**边栏选项卡。
7. 弹出一条信息性消息表示成功安装的扩展。

如果未启用 stdout 日志记录, 请按照下列步骤:

1. 在 Azure 门户中, 选择高级工具中的边栏选项卡**开发工具**区域。选择转→按钮。Kudu 控制台打开新浏览器选项卡或窗口中。
2. 使用的页上, 顶部导航栏打开调试控制台和选择**CMD**。
3. 打开的文件夹的路径站点 > **wwwroot**向下滚至揭示**web.config**列表底部的文件。
4. 单击铅笔图标旁边**web.config**文件。
5. 设置**stdoutLogEnabled**到 `true` 和更改**stdoutLogFile**路径: `\?\%home%\LogFiles\stdout`。
6. 选择**保存**以保存已更新**web.config**文件。

继续激活诊断日志记录:

1. 在 Azure 门户中, 选择**诊断日志**边栏选项卡。
2. 选择**上切换应用程序日志记录 (文件系统) 和详细错误消息**。选择**保存**边栏选项卡顶部的按钮。
3. 若要包含失败的请求跟踪, 也称为失败请求事件缓冲 (FREB) 日志记录, 选择**上切换失败请求跟踪**。
4. 选择**日志流**边栏选项卡, 其中立即下列出**诊断日志**在门户中的边栏选项卡。
5. 向应用程序发出的请求。
6. 在日志的流数据, 指示错误的原因。

重要提示 ! 请务必禁用完成故障排除时, 日志记录 stdout。请参阅中的说明**ASP.NET 核心模块 stdout 日志部分**。

若要查看失败的请求跟踪日志 (FREB 日志):

1. 导航到**诊断并解决问题**在 Azure 门户中的边栏选项卡。
2. 选择**失败请求跟踪日志从支持工具侧栏区域**。

请参阅失败请求跟踪部分中的 Azure App Service 主题中的 web 应用启用诊断日志记录和 Azure 中的 Web 应用的应用程序性能常见问题: 如何打开失败的请求跟踪? 有关详细信息。

有关详细信息, 请参阅[启用 Azure App Service 中的 web 应用的诊断日志记录](#)。

警告

若要禁用 stdout 日志可能会导致应用程序或服务器失败。日志文件大小或创建的日志文件数没有限制。

对于例程日志记录在 ASP.NET Core 应用程序, 使用限制日志文件大小和旋转日志的日志记录库。有关详细信息, 请参阅[第三方日志记录提供程序](#)。

其他资源

- [ASP.NET Core 中的错误处理简介](#)
- [Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考](#)
- [对 Azure App Service 中使用 Visual Studio 中的 web 应用进行故障排除](#)
- [排除"502 错误网关"和"503 服务不可用" Azure web 应用中的 HTTP 的错误](#)
- [解决在 Azure App Service 的慢速 web 应用程序性能问题](#)
- [在 Azure 中的 Web 应用的应用程序性能常见问题](#)
- [Azure Web 应用沙盒 \(应用程序服务运行时执行限制\)](#)
- [Azure Friday: Azure 应用服务诊断和故障排除体验\(12 分钟视频\)](#)

使用 IIS 在 Windows 上托管 ASP.NET Core

2018/4/28 • 23 min to read • [Edit Online](#)

作者: [Luke Latham](#) 和 [Rick Anderson](#)

支持的操作系统

支持下列操作系统:

- Windows 7 或更高版本
- Windows Server 2008 R2 或更高版本

[HTTP.sys 服务器](#)(以前称为 [WebListener](#))在使用 IIS 的反向代理配置中不起作用。请使用 [Kestrel 服务器](#)。

应用程序配置

启用 `IISIntegration` 组件

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

典型的 `Program.cs` 调用 `CreateDefaultBuilder` 以开始设置主机。`CreateDefaultBuilder` 将 `Kestrel` 配置为 Web 服务器，并通过配置 [ASP.NET Core 模块](#)的基路径和端口来实现 IIS 集成:

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

ASP.NET Core 模块生成分配给后端进程的动态端口。`UseIISIntegration` 方法获取该动态端口，并将 `Kestrel` 配置为侦听 `http://localhost:{dynamicPort}/`。这将替代其他 URL 配置，如对 `UseUrls` 或 [Kestrel 的侦听 API](#) 的调用。因此，使用模块时，不需要调用 `UseUrls` 或 `Kestrel` 的 `Listen` API。如果调用 `UseUrls` 或 `Listen`，则 `Kestrel` 会侦听在没有 IIS 的情况下运行应用时指定的端口。

有关托管的详细信息，请参阅 [ASP.NET Core 中的托管](#)。

IIS 选项

若要配置 IIS 选项，请在 `ConfigureServices` 中加入适用于 `IISOptions` 的服务配置。在下面的示例中，禁用将客户端证书转发到应用以填充 `HttpContext.Connection.ClientCertificate`:

```
services.Configure<IISSettings>(options =>
{
    options.ForwardClientCertificate = false;
});
```

选项	默认	设置
----	----	----

选项	默认	设置
AutomaticAuthentication	true	若为 true, IIS 集成中间件将设置经过 Windows 身份验证进行身份验证的 HttpContext.User。若为 false, 中间件仅提供 HttpContext.User 的标识并在 AuthenticationScheme 显式请求时响应质询。必须在 IIS 中启用 Windows 身份验证使 AutomaticAuthentication 得以运行。有关详细信息, 请参阅 Windows 身份验证主题 。
AuthenticationDisplayName	null	设置在登录页上向用户显示的显示名。
ForwardClientCertificate	true	若为 true, 且存在 MS-ASPNETCORE-CLIENTCERT 请求头, 则填充 HttpContext.Connection.ClientCertificate。

代理服务器和负载均衡器方案

配置转发头中间件的 IIS 集成中间件和 ASP.NET Core 模块将配置为转发方案 (HTTP/HTTPS) 和发出请求的远程 IP 地址。对于托管在其他代理服务器和负载均衡器后方的应用, 可能需要附加配置。有关详细信息, 请参阅 [配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

web.config 文件

web.config 文件配置 [ASP.NET Core 模块](#)。web.config 的创建、转换和发布由 .NET Core Web SDK (`Microsoft.NET.Sdk.Web`) 处理。SDK 设置在项目文件的顶部:

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

如果项目中不存在 web.config 文件, 则会使用正确的 processPath 和参数创建该文件, 以便配置 [ASP.NET Core 模块](#), 并将该文件移动到已发布的输出。

如果项目中存在 web.config 文件, 则会使用正确的 processPath 和参数转换该文件, 以便配置 ASP.NET Core 模块, 并将该文件移动到已发布的输出。转换不会修改文件中的 IIS 配置设置。

web.config 文件可能会提供其他 IIS 配置设置, 以控制活动的 IIS 模块。有关能够处理 ASP.NET Core 应用请求的 IIS 模块的信息, 请参阅 [IIS 模块主题](#)。

要防止 Web SDK 转换 web.config 文件, 请使用项目文件中的 `<IsTransformWebConfigDisabled>` 属性:

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

在禁用 Web SDK 转换文件时, 开发人员应手动设置 processPath 和参数。有关详细信息, 请参阅 [ASP.NET Core 模块配置参考](#)。

web.config 文件位置

ASP.NET Core 应用在 IIS 与 Kestrel 服务器之间的反向代理中托管。为了创建反向代理, web.config 文件必须存在于已部署应用的内容根路径(通常为应用基路径)中。该位置与向 IIS 提供的网站物理路径相同。若要使用 Web 部署发布多个应用, 应用的根路径中需要包含 web.config 文件。

敏感文件存在于应用的物理路径中, 如 `<assembly>.runtimeconfig.json`、`<assembly>.xml`(XML 文档注释)和

<assembly>.deps.json。当存在 web.config 文件且站点正常启动的情况下，若要请求获取这些敏感文件，IIS 不会提供。如果缺少 web.config 文件、命名不正确，或无法配置站点以正常启动，IIS 可能会公开提供敏感文件。

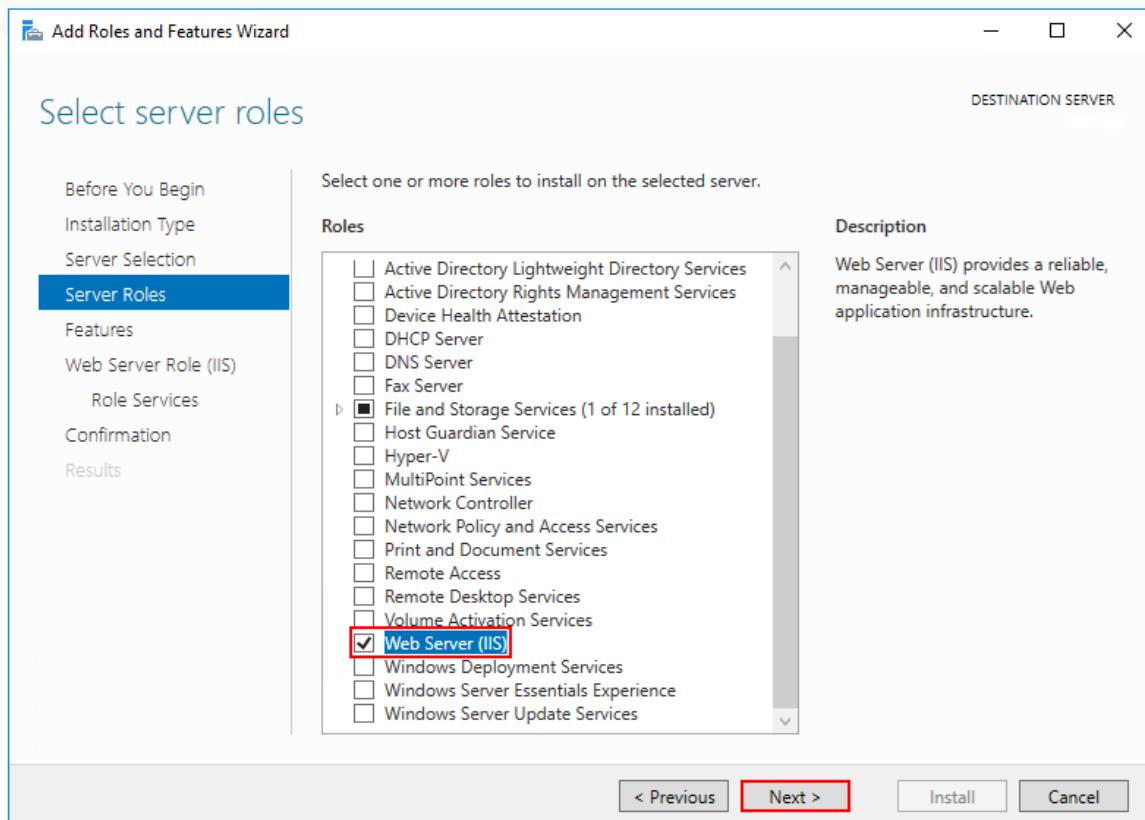
部署中必须始终存在 web.config 文件且正确命名，并可以配置站点以正常启动。请勿从生产部署中删除 web.config 文件。

IIS 配置

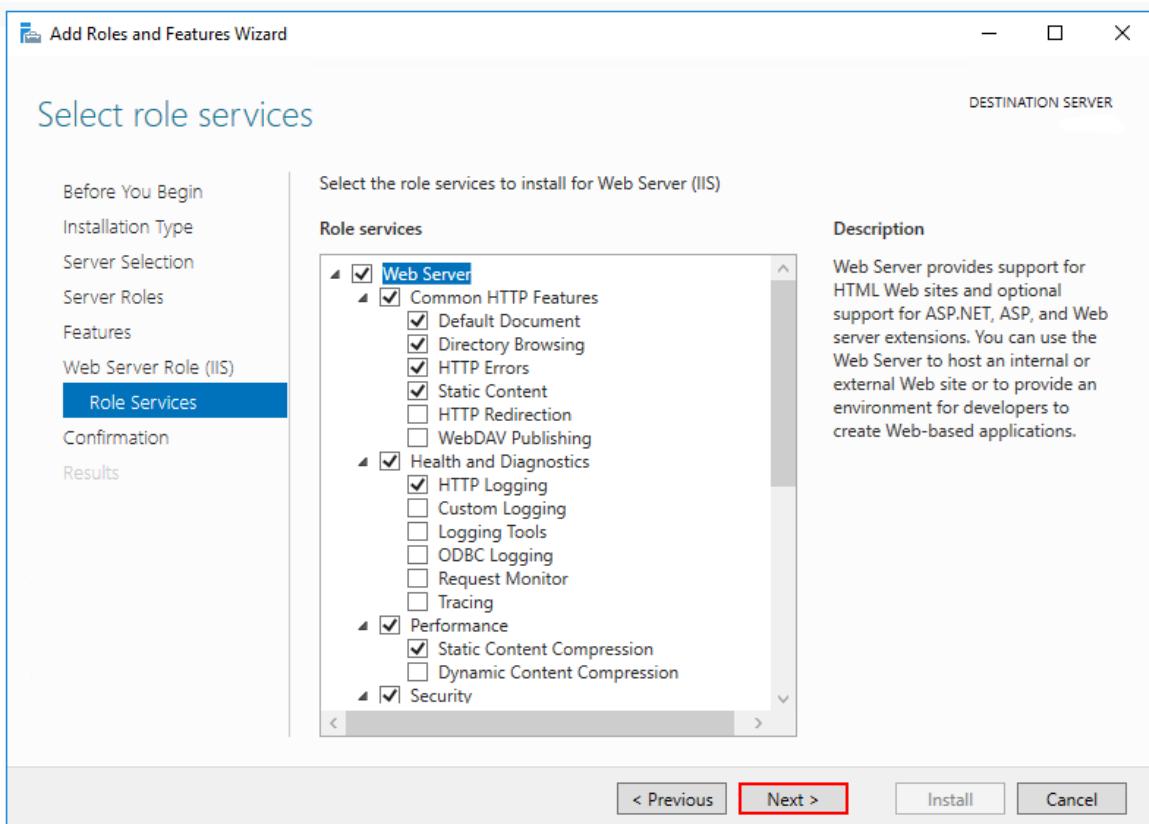
Windows Server 操作系统

启用 Web 服务器 (IIS) 服务器角色并建立角色服务。

- 通过“管理”菜单或“服务器管理器”中的链接使用“添加角色和功能”向导。在“服务器角色”步骤中，选中“Web 服务器(IIS)”框。



- 在“功能”步骤后，为 Web 服务器 (IIS) 加载“角色服务”步骤。选择所需 IIS 角色服务，或接受提供的默认角色服务。



Windows 身份验证(可选)

若要启用 Windows 身份验证, 请展开以下节点：“Web 服务器” > “安全”。选择“Windows 身份验证”功能。有关详细信息, 请参阅 [Windows 身份验证 <windowsAuthentication>](#) 和[配置 Windows 身份验证](#)。

Websocket(可选)

Websocket 支持 ASP.NET Core 1.1 或更高版本。若要启用 Websocket, 请展开以下节点：“Web 服务器” > “应用程序开发”。选择“WebSocket 协议”功能。有关详细信息, 请参阅 [WebSockets](#)。

3. 继续执行“确认”步骤, 安装 Web 服务器角色和服务。安装 Web 服务器 (IIS) 角色后无需重启服务器/IIS。

Windows 桌面操作系统

启用“IIS 管理控制台”和“万维网服务”。

1. 导航到“控制面板” > “程序” > “程序和功能” > “打开或关闭 Windows 功能”(位于屏幕左侧)。
2. 打开“Internet Information Services”节点。打开“Web 管理工具”节点。
3. 选中“IIS 管理控制台”框。
4. 选中“万维网服务”框。
5. 接受“万维网服务”的默认功能, 或自定义 IIS 功能。

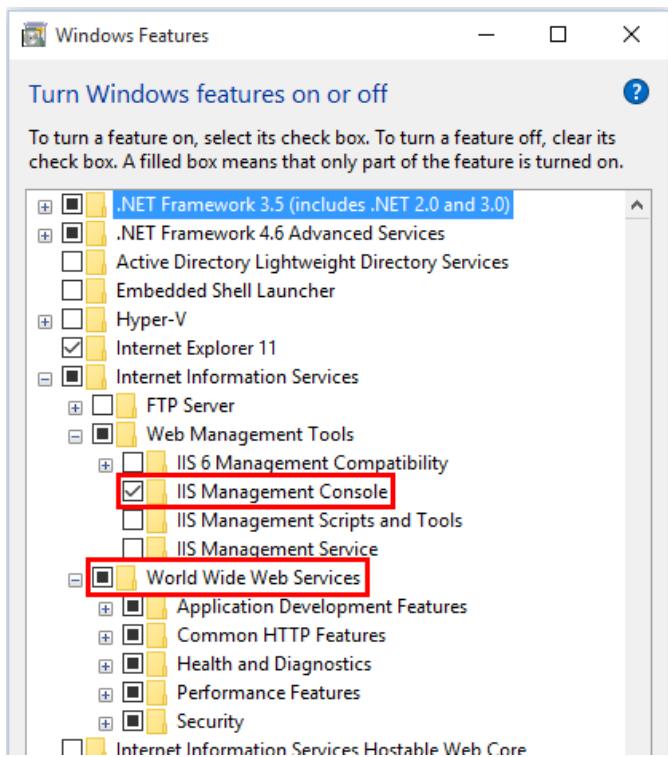
Windows 身份验证(可选)

若要启用 Windows 身份验证, 请展开以下节点：“万维网服务” > “安全”。选择“Windows 身份验证”功能。有关详细信息, 请参阅 [Windows 身份验证 <windowsAuthentication>](#) 和[配置 Windows 身份验证](#)。

Websocket(可选)

Websocket 支持 ASP.NET Core 1.1 或更高版本。若要启用 Websocket, 请展开以下节点：“万维网服务” > “应用程序开发功能”。选择“WebSocket 协议”功能。有关详细信息, 请参阅 [WebSockets](#)。

6. 如果 IIS 安装需要重新启动, 则重新启动系统。



安装 .NET Core 托管捆绑包

1. 在托管系统上安装 .NET Core 托管捆绑包。捆绑包可安装 .NET Core 运行时、.NET Core 库和 [ASP.NET Core 模块](#)。该模块创建 IIS 与 Kestrel 服务器之间的反向代理。如果系统没有 Internet 连接，请先获取并安装 [Microsoft Visual C++ 2015 Redistributable](#)，然后再安装 .NET Core 托管捆绑包。

- 导航到 [.NET 所有下载页](#)。
- 从列表中选择最新的非预览 .NET Core 运行时 (.NET Core > 运行时 > .NET Core 运行时 x.y.z)。除非你想要使用预览软件，否则请避免选择其链接文本中包含“预览”一词的运行时。
- 在 Windows 下的 .NET Core 运行时下载页上，选择“托管捆绑包安装程序”链接以下载 .NET Core 托管捆绑包。

重要提示！ 如果在 IIS 之前安装了托管捆绑包，则必须修复捆绑包安装。在安装 IIS 后再次运行托管捆绑包安装程序。

若要防止安装程序在 x64 操作系统上安装 x86 程序包，请通过管理员命令提示符使用开关 `OPT_NO_X86=1` 来运行安装程序。

2. 重启系统，或从命令提示符处依次执行 `net stop was /y` 和 `net start w3svc`。重新启动 IIS 将选取安装程序对系统 PATH 所作的更改。

注意

有关 IIS 共享配置的信息，请参阅[使用 IIS 共享配置的 ASP.NET Core 模块](#)。

使用 Visual Studio 进行发布时安装 Web 部署

使用 [Web 部署](#)将应用部署到服务器时，请在服务器上安装最新版本的 Web 部署。要安装 Web 部署，请使用 [Web 平台安装程序 \(WebPI\)](#) 或直接从 [Microsoft 下载中心](#) 获取安装程序。建议使用 WebPI。WebPI 为托管提供程序提供独立的安装程序和配置。

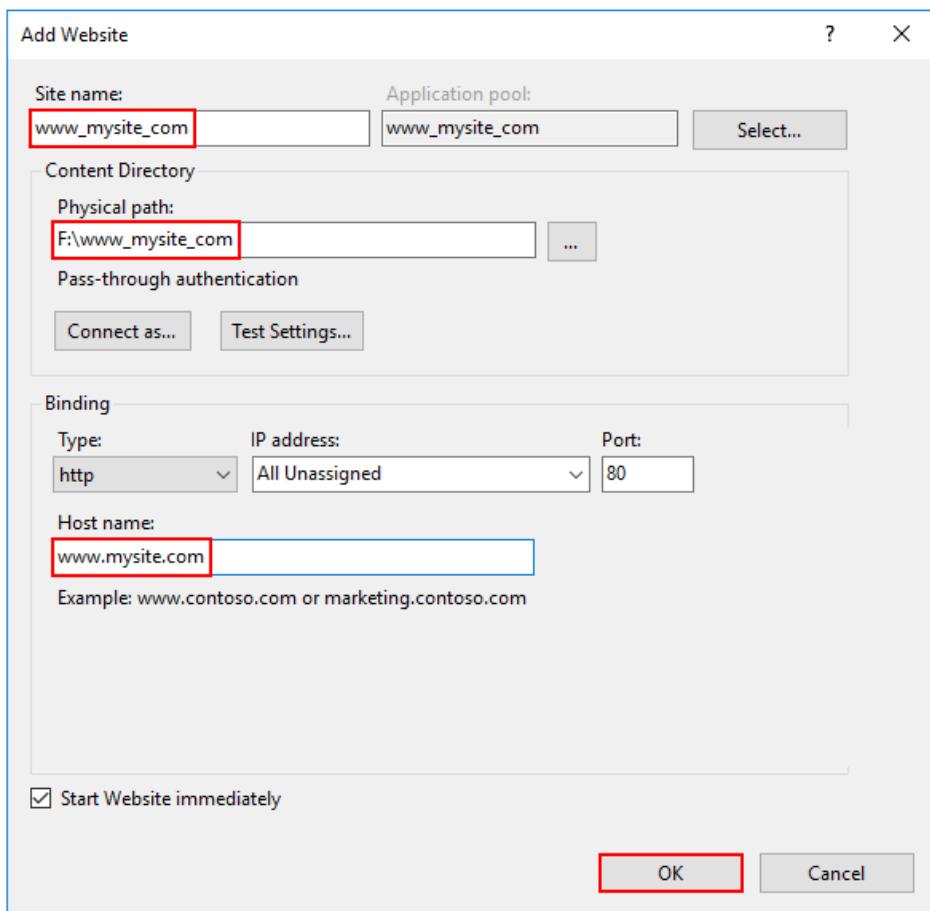
创建 IIS 站点

- 在托管系统上，创建一个文件夹以包含应用已发布的文件夹和文件。[目录结构](#)主题中介绍了应用的部署布局。
- 在新文件夹中创建一个“日志”文件夹，用于在启用 stdout 日志记录时保存 ASP.NET Core 模块 stdout 日志。如果部署应用时有效负载中包含了“日志”文件夹，请跳过此步骤。有关如何启用 MSBuild 以便在本地生成项目时自动创建“日志”文件夹的说明，请参阅[目录结构](#)主题。

重要事项

仅使用 stdout 日志来解决应用启动失败的问题。请勿使用 stdout 日志记录进行常规应用日志记录。日志文件大小或创建的日志文件数没有限制。应用池必须对写入日志的位置具有写入权限。日志位置路径上的所有文件夹都必须存在。有关 stdout 日志的详细信息，请参阅[日志创建和重定向](#)。有关 ASP.NET Core 应用中的日志记录信息，请参阅[日志记录](#)主题。

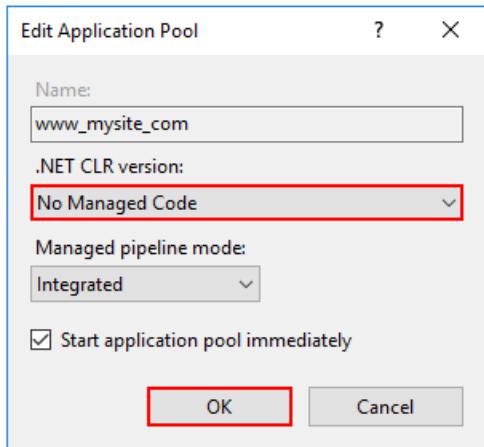
- 在“IIS 管理器”中，打开“连接”面板中的服务器节点。右键单击“站点”文件夹。选择上下文菜单中的“添加网站”。
- 提供网站名称，并将物理路径设置为应用的部署文件夹。提供“绑定”配置，并通过选择“确定”创建网站：



警告

不应使用顶级通配符绑定 (`http://*:80/` 和 `http://+:80`)。顶级通配符绑定可能会为应用带来安全漏洞。此行为同时适用于强通配符和弱通配符。使用显式主机名而不是通配符。如果可控制整个父域(区别于易受攻击的 `*.com`)，则子域通配符绑定(例如，`*.mysub.com`)不具有此安全风险。有关详细信息，请参阅[rfc7230 第 5.4 条](#)。

- 在服务器节点下，选择“应用程序池”。
- 右键单击站点的应用池，然后从上下文菜单中选择“基本设置”。
- 在“编辑应用程序池”窗口中，将“.NET CLR 版本”设置为“无托管代码”：



ASP.NET Core 在单独的进程中运行，并管理运行时。ASP.NET Core 不依赖加载桌面 CLR。将“.NET CLR 版本”设置为“无托管代码”为可选步骤。

8. 确认进程模型标识拥有适当的权限。

如果将应用池的默认标识（“进程模型” > “标识”）从 ApplicationPoolIdentity 更改为另一标识，请验证新标识拥有所需的权限，可访问应用的文件夹、数据库和其他所需资源。例如，应用池需要对文件夹的读取和写入权限，以便应用在其中读取和写入文件。

Windows 身份验证配置（可选）

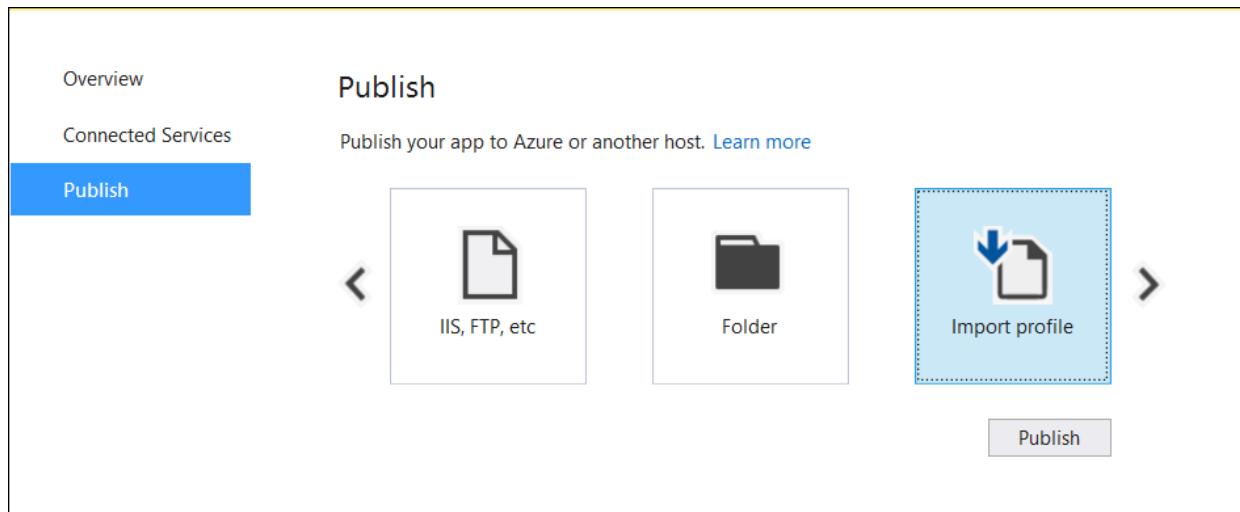
有关详细信息，请参阅[配置 Windows 身份验证](#)。

部署应用

将应用部署到在托管系统上创建的文件夹。建议使用的部署机制是[Web 部署](#)。

在 Visual Studio 内使用 Web 部署

要了解如何创建用于 Web 部署的发布配置文件，请参阅[用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件](#)。如果托管提供程序提供了发布配置文件或支持创建发布配置文件，请下载配置文件并使用 Visual Studio 的“发布”对话框将其导入。



在 Visual Studio 之外使用 Web 部署

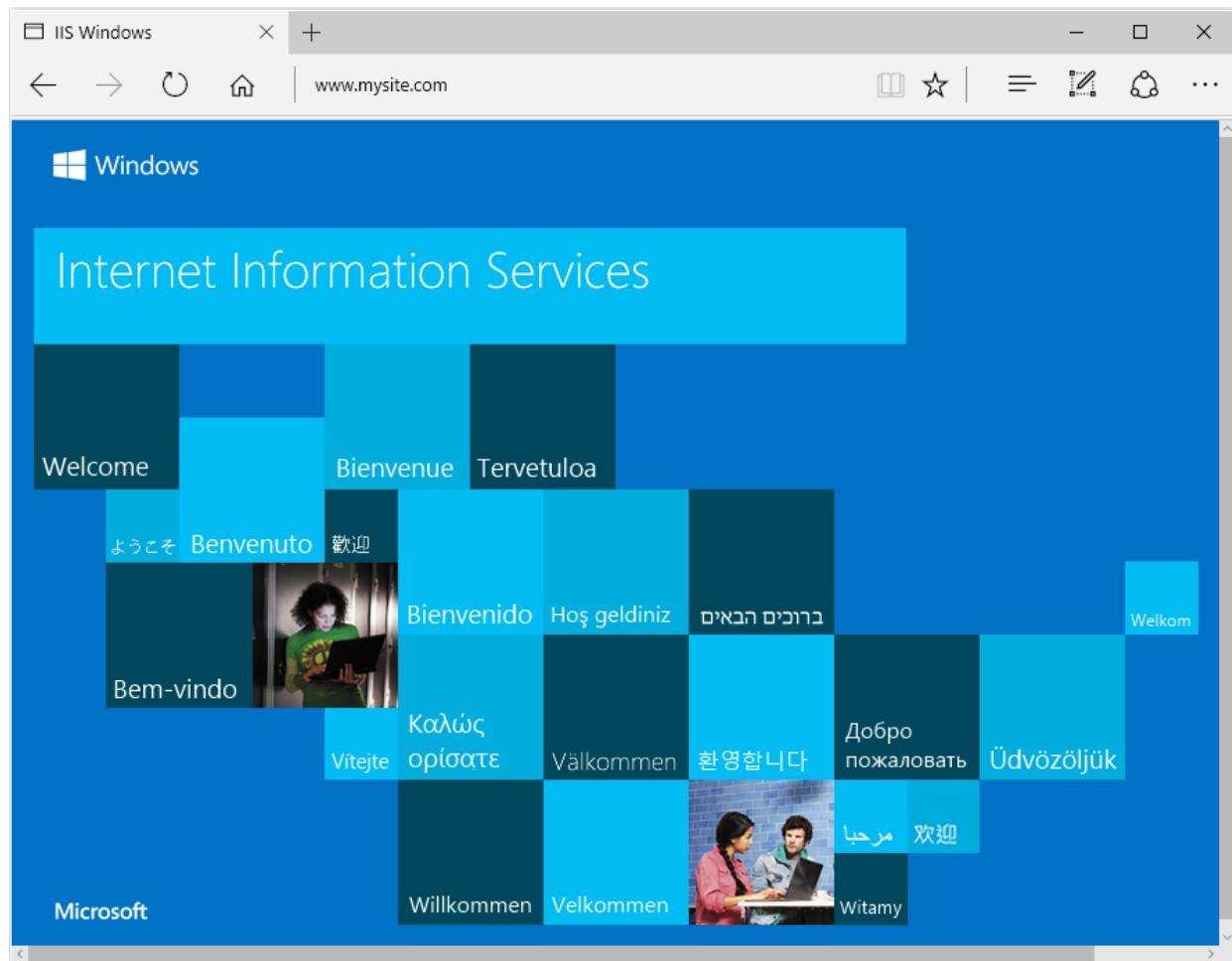
也可以在 Visual Studio 之外从命令行使用[Web 部署](#)。有关详细信息，请参阅[Web Deployment Tool \(Web 部署工具\)](#)。

Web 部署的替代方法

有多种方法可将应用移动到托管系统，例如手动复制、Xcopy、Robocopy 或 PowerShell，可使用其中任何一种方法。

有关将 ASP.NET Core 部署到 IIS 的详细信息, 请参阅[面向 IIS 管理员的部署资源部分](#)。

浏览网站



锁定的部署文件

如果应用正在运行, 部署文件夹中的文件会被锁定。在部署期间, 无法覆盖已锁定的文件。若要在部署中解除已锁定的文件, 请使用以下方法之一停止应用池:

- 使用 Web 部署并在项目文件中引用 `Microsoft.NET.Sdk.Web`。系统会在 Web 应用目录的根目录中放置一个 `app_offline.htm` 文件。该文件存在时, ASP.NET Core 模块会在部署过程中正常关闭该应用并提供 `app_offline.htm` 文件。有关详细信息, 请参阅 [ASP.NET Core 模块配置参考](#)。
- 在服务器上的 IIS 管理器中手动停止应用池。
- 使用 PowerShell 来停止和重新启动应用池(需要使用 PowerShell 5 或更高版本):

```

$webAppPoolName = 'APP_POOL_NAME'

# Stop the AppPool
if((Get-WebAppPoolState $webAppPoolName).Value -ne 'Stopped') {
    Stop-WebAppPool -Name $webAppPoolName
    while((Get-WebAppPoolState $webAppPoolName).Value -ne 'Stopped') {
        Start-Sleep -s 1
    }
    Write-Host `'-AppPool Stopped
}

# Provide script commands here to deploy the app

# Restart the AppPool
if((Get-WebAppPoolState $webAppPoolName).Value -ne 'Started') {
    Start-WebAppPool -Name $webAppPoolName
    while((Get-WebAppPoolState $webAppPoolName).Value -ne 'Started') {
        Start-Sleep -s 1
    }
    Write-Host `'-AppPool Started
}

```

数据保护

[ASP.NET Core 数据保护堆栈](#)由多个 ASP.NET Core 中间件使用，包括用于身份验证的中间件。即使用户代码不调用数据保护 API，也应该使用部署脚本或在用户代码中配置数据保护，以创建持久的加密密钥存储。如果不配置数据保护，则密钥存储在内存中。重启应用时，密钥会被丢弃。

如果密钥环存储于内存中，则在应用重启时：

- 所有基于 cookie 的身份验证令牌都无效。
- 用户需要在下一次请求时再次登录。
- 无法再解密使用密钥环保护的任何数据。这可能包括 [CSRF 令牌](#) 和 [ASP.NET Core MVC TempData cookie](#)。

若要在 IIS 下配置数据保护以持久化密钥环，请使用以下方法之一：

- 创建数据保护注册表项

ASP.NET Core 应用使用的数据保护密钥存储在应用外部的注册表中。要持久保存给定应用的密钥，需为应用池创建注册表项。

对于独立的非 Web 场 IIS 安装，可以对用于 ASP.NET Core 应用的每个应用池使用[数据保护 Provision-AutoGenKeys.ps1 PowerShell 脚本](#)。此脚本在 HKLM 注册表中创建注册表项，仅应用程序的应用池工作进程帐户可对其进行访问。通过计算机范围的密钥使用 DPAPI 对密钥静态加密。

在 web 场方案中，可以将应用配置为使用 UNC 路径存储其数据保护密钥环。默认情况下，数据保护密钥未加密。确保网络共享的文件权限仅限于应用在其下运行的 Windows 帐户。可使用 X509 证书来保护静态密钥。建议使用允许用户上传证书的机制：将证书放置在用户信任的证书存储中，并确保在运行用户应用的所有计算机上都可使用这些证书。有关详细信息，请参阅[配置 ASP.NET Core 数据保护](#)。

- 配置 IIS 应用程序池以加载用户配置文件

此设置位于应用池“高级设置”下的“进程模型”部分。将“加载用户配置文件”设置为 `True`。这会将密钥存储在用户配置文件目录下，并使用 DPAPI 和特定于用户帐户（用于应用池）的密钥进行保护。

- 将文件系统用作密钥环存储

调整应用代码，[将文件系统用作密钥环存储](#)。使用 X509 证书保护密钥环，并确保该证书是受信任的证书。如果它是自签名证书，则将该证书放置于受信任的根存储中。

当在 Web 场中使用 IIS 时：

- 使用所有计算机都可以访问的文件共享。
 - 将 X509 证书部署到每台计算机。[通过代码配置数据保护](#)。
- 设置用于数据保护的计算机范围的策略

数据保护系统对以下操作提供有限支持:为使用数据保护 API 的所有应用设置默认计算机范围的策略。有关详细信息,请参阅[数据保护文档](#)。

子应用程序配置

在根应用下添加的子应用不应将 ASP.NET Core 模块作为处理程序包含在其中。如果在子应用的 web.config 文件中将该模块添加为处理程序,则在尝试浏览子应用时会收到“500.19 内部服务器错误”,即引用错误的配置文件。

以下示例显示 ASP.NET Core 子应用的已发布 web.config 文件:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <aspNetCore processPath="dotnet"
      arguments=".\\<assembly_name>.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\\logs\\stdout" />
  </system.webServer>
</configuration>
```

在 ASP.NET Core 应用下托管非 ASP.NET Core 子应用时,需在子应用的 web.config 文件中显式删除继承的处理程序:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\\<assembly_name>.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\\logs\\stdout" />
  </system.webServer>
</configuration>
```

有关配置 ASP.NET Core 模块的详细信息,请参阅[ASP.NET Core 模块简介](#)和[ASP.NET Core 模块配置参考](#)。

使用 web.config 配置 IIS

对于那些应用于反向代理配置的 IIS 功能, IIS 配置受 web.config 的 `<system.webServer>` 部分影响。如果在服务器级别将 IIS 配置为使用动态压缩,则可通过应用的 web.config 文件中的 `<urlCompression>` 元素禁用它。

有关详细信息,请参阅[<system.webServer> 的配置参考](#)、[ASP.NET Core 模块配置参考](#)和[使用 ASP.NET Core 的 IIS 模块](#)。若要为在独立应用池中运行的各应用设置环境变量(IIS 10.0 或更高版本中支持此操作),请参阅 IIS 参考文档的[环境变量 <environmentVariables>](#)主题中的“AppCmd.exe 命令”部分。

web.config 的配置节

ASP.NET Core 应用不会使用 web.config 中的 ASP.NET 4.x 应用的配置部分进行配置:

- `<system.web>`
- `<appSettings>`

- <connectionStrings>
- <location>

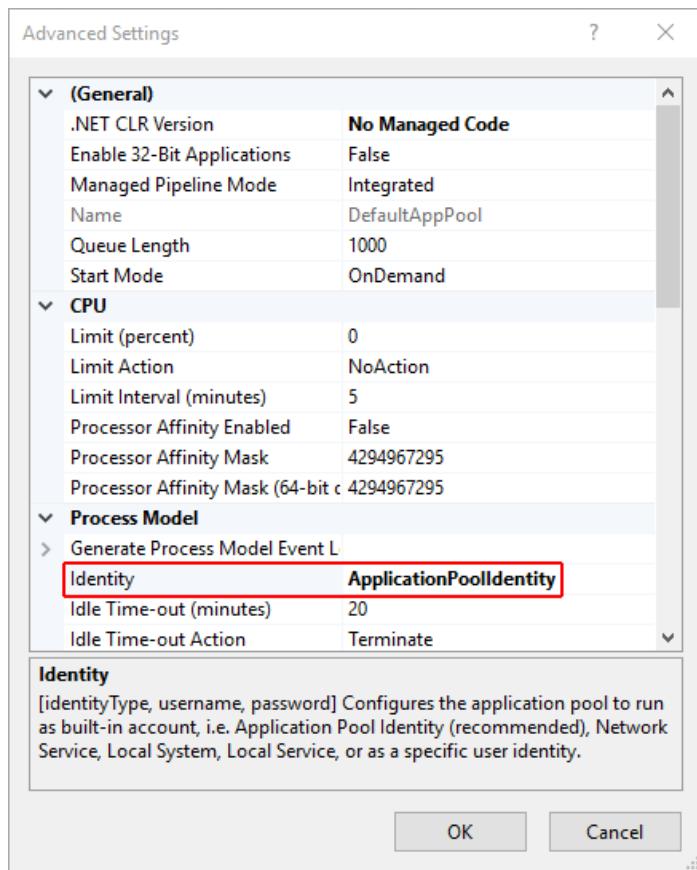
会使用其他的配置提供程序配置 ASP.NET Core 应用。有关详细信息, 请参阅[配置](#)。

应用程序池

在服务器上托管多个网站时, 需在每个应用自己的应用池中运行各应用, 以彼此隔离。IIS“添加网站”对话框默认执行此配置。提供了站点名称时, 该文本会自动传输到“应用程序池”文本框。添加站点时, 会使用该站点名称创建新的应用池。

应用程序池标识

通过应用池标识帐户, 可以在唯一帐户下运行应用, 而无需创建和管理域或本地帐户。在 IIS 8.0 或更高版本上, IIS 管理员工作进程 (WAS) 将使用新应用池的名称创建一个虚拟帐户, 并默认在此帐户下运行应用池的工作进程。在 IIS 管理控制台中, 确保应用池“高级设置”下的“标识”设置为使用 ApplicationPoolIdentity:

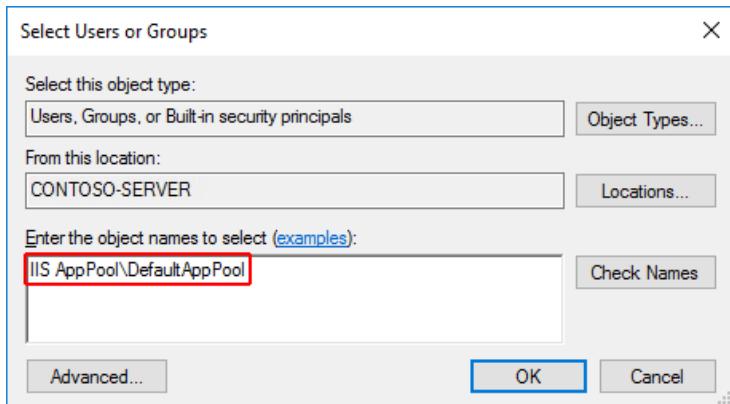


IIS 管理进程使用 Windows 安全系统中应用池的名称创建安全标识符。可使用此标识保护资源。但是, 此标识不是真实的用户帐户, 不会在 Windows 用户管理控制台中显示。

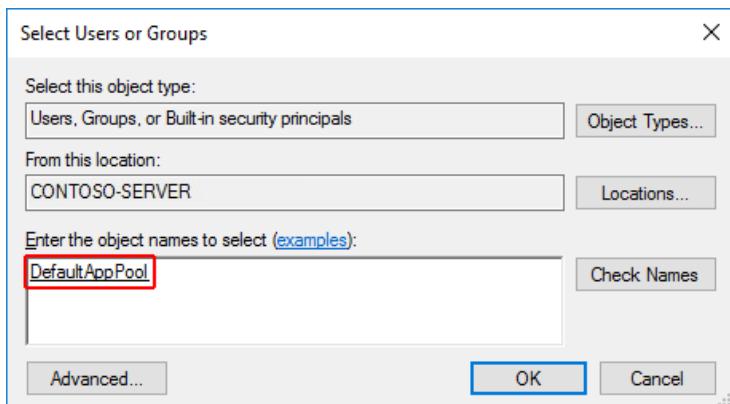
如果 IIS 工作进程需要对应用的高级访问权限, 请为包含该应用的目录修改访问控制列表 (ACL):

1. 打开 Windows 资源管理器并导航到目录。
2. 右键单击该目录, 然后选择“属性”。
3. 在“安全”选项卡下, 选择“编辑”按钮, 然后单击“添加”按钮。
4. 选择“位置”按钮, 并确保该系统处于选中状态。
5. 在“输入要选择的对象名称”区域中输入“`IIS AppPool\<app_pool_name>`”。选择“检查名称”按钮。有关 DefaultAppPool, 请检查使用 IIS AppPool\DefaultAppPool 的名称。当选择“检查名称”按钮时, 对象名称区域中会显示 DefaultAppPool 的值。无法直接在对象名称区域中输入应用池名称。检查对象名称时, 请

使用 IIS AppPool\<app_pool_name> 格式。



6. 选择“确定”。



7. 默认情况下应授予读取 & 执行权限。根据需要请提供其他权限。

也可使用 ICACLS 工具在命令提示符处授予访问权限。以 DefaultAppPool 为例，使用以下命令：

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

有关详细信息，请参阅 [icacls](#) 主题。

面向 IIS 管理员的部署资源

在 IIS 文档中深入了解 IIS。

[IIS 文档](#)

了解 .NET Core 应用部署模型。

[.NET Core 应用程序部署](#)

了解 ASP.NET Core 模块如何使 Kestrel Web 服务器将 IIS 或 IIS Express 用作反向代理服务器。

[ASP.NET Core 模块](#)

了解如何配置 ASP.NET Core 模块以托管 ASP.NET Core 应用。

[ASP.NET Core 模块配置参考](#)

了解已发布的 ASP.NET Core 应用的目录结构。

[目录结构](#)

了解适用于 ASP.NET Core 应用的活动和非活动 IIS 模块以及如何管理 IIS 模块。

[IIS 模块](#)

了解如何诊断 ASP.NET Core 应用的 IIS 部署的问题。

[疑难解答](#)

了解在 IIS 上托管 ASP.NET Core 应用的常见错误。

[Azure 应用服务和 IIS 的常见错误参考](#)

其他资源

- [ASP.NET Core 简介](#)
- [Microsoft IIS 官方网站](#)
- [Windows Server 技术内容库](#)

解决在 IIS 上的 ASP.NET 核心

2018/4/10 • 8 min to read • [Edit Online](#)

作者: Luke Latham

本文说明了如何诊断 ASP.NET Core 应用启动问题使用承载时 Internet 信息服务 (IIS)。这篇文章中的信息适用于在 Windows Server 和 Windows 桌面上的 IIS 中承载。

在 Visual Studio 中, ASP.NET Core 项目默认为 IIS Express 承载在调试过程。A 502.5 进程失败本地调试可能会使用本主题中的建议的 troubleshooted 时发生。

其他故障排除主题:

[对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)

尽管应用程序服务使用 [ASP.NET 核心模块](#) 和 IIS 承载应用程序, 请参阅说明特定于应用程序服务的专用的主题。

[处理错误](#)

了解如何在本地系统上的开发过程中处理 ASP.NET Core 应用中的错误。

[了解如何使用 Visual Studio 进行调试](#)

本主题介绍 Visual Studio 调试器的功能。

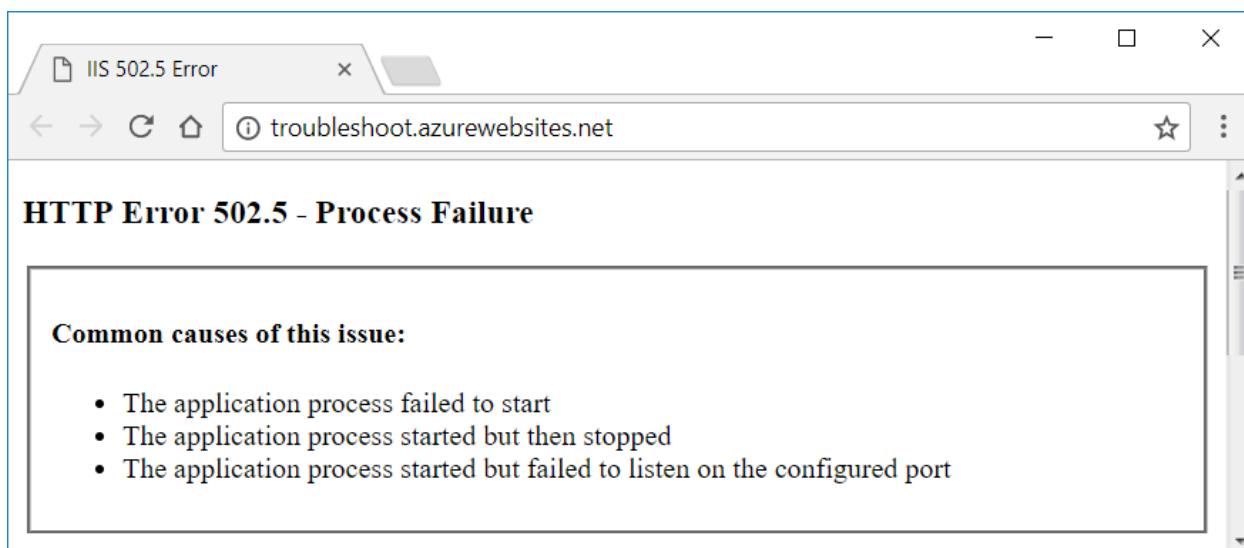
应用程序启动错误

502.5 进程故障

工作进程将失败。不启动应用程序。

ASP.NET 核心模块尝试启动工作进程, 但它无法启动。进程启动失败的原因通常根据中的条目 [应用程序事件日志](#) 和 [ASP.NET 核心模块 stdout 日志](#)。

502.5 进程失败当承载或应用程序配置错误导致工作进程失败时返回错误页:



500 内部服务器错误

启动该应用程序, 但某个错误阻止在完成请求的服务器。

启动过程中或创建响应时, 应用程序的代码中出现此错误。响应可能包含任何内容, 或响应可能显示为 500 内部服务器错误浏览器中。应用程序事件日志通常表明应用程序正常启动。从服务器的角度来看, 这是正确的。应用程序未启动, 但它无法生成有效的响应。在命令提示符下运行 [应用程序](#) 服务器上或 [启用 ASP.NET 核心模块 stdout 日志](#) 以解决该问题。

连接重置

如果发送标头后，将发生错误，则服务器尝试发送太晚**500 内部服务器**错误发生错误时。响应的复杂对象的序列化期间发生错误时经常会出现这种情况。此类型的错误显示为**连接重置**客户端上的错误。[应用程序日志记录](#)可以帮助解决这些类型的错误。

默认启动限制

ASP.NET 核心模块配置了默认值`startupTimeLimit`的 120 秒。时保留为默认值，应用可能需要最多两分钟，以启动之前模块记录进程失败。有关模块的配置的信息，请参阅[aspNetCore 元素的特性的](#)。

应用程序启动错误疑难解答

应用程序事件日志

访问应用程序事件日志：

1. 打开开始菜单，搜索事件查看器，然后选择事件查看器应用。
2. 在事件查看器，打开**Windows 日志**节点。
3. 选择**应用程序**以打开应用程序事件日志。
4. 搜索与失败应用相关联的错误。错误都具有值*IIS AspNetCore 模块*或*IIS Express AspNetCore 模块*中源列。

在命令提示符下运行应用程序

许多启动错误未生成应用程序事件日志中的有用信息。你可以通过在命令提示符下运行应用，在主机系统上找到某些错误的原因。

依赖于框架的部署

如果在应用程序处于[framework 相关部署](#)：

1. 在命令提示符处，导航到部署文件夹，并通过执行应用程序的程序集与运行应用`dotnet.exe`。在下面的命令中，应用程序的程序集的名称替换`<程序集_名称>`：`dotnet .\<assembly_name>.dll`。
2. 控制台应用程序中，显示任何错误，从输出写入到控制台窗口。
3. 如果向应用程序发出请求时出现错误，请向主机和其中 Kestrel 倾听的端口发出请求。使用默认主机和开机自检，请发出请求以 `http://localhost:5000/`。如果应用程序通常响应 Kestrel 终结点地址，该问题更可能是相关的反向代理配置和不太可能在应用内。

自包含的部署

如果在应用程序处于[独立的部署](#)：

1. 在命令提示符处，导航到部署文件夹并运行应用程序的可执行文件。在下面的命令中，应用程序的程序集的名称替换`<程序集_名称>`：`<assembly_name>.exe`。
2. 控制台应用程序中，显示任何错误，从输出写入到控制台窗口。
3. 如果向应用程序发出请求时出现错误，请向主机和其中 Kestrel 倾听的端口发出请求。使用默认主机和开机自检，请发出请求以 `http://localhost:5000/`。如果应用程序通常响应 Kestrel 终结点地址，该问题更可能是相关的反向代理配置和不太可能在应用内。

ASP.NET 核心模块 `stdout` 日志

用于启用和查看 `stdout` 日志：

1. 导航到主机系统上的站点的部署文件夹。
2. 如果日志文件夹不存在、创建文件夹。有关如何启用 MSBuild 的说明创建日志部署中的文件夹自动，请参阅[目录结构](#)主题。
3. 编辑`web.config`文件。设置`stdoutLogEnabled`到 `true` 和更改`stdoutLogFile`路径以指向日志文件夹（例如，`.\logs\stdout`）。`stdout` 在路径中是日志文件名称前缀。时间戳、进程 id 和文件扩展名自动添加了时创建

日志。使用 `stdout` 作为文件名称前缀，典型的日志文件名为 `stdout_20180205184032_5412.log`。

4. 保存已更新 `web.config` 文件。
5. 向应用程序发出的请求。
6. 导航到 日志文件 夹。查找并打开最新的标准输出日志。
7. 研究错误的日志。

重要提示！ 禁用完成故障排除时，日志记录 `stdout`。

1. 编辑 `web.config` 文件。
2. 设置 `stdoutLogEnabled` 到 `false`。
3. 保存该文件。

警告

若要禁用 `stdout` 日志可能会导致应用程序或服务器失败。日志文件大小或创建的日志文件数没有限制。

对于例程日志记录在 ASP.NET Core 应用程序，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅[第三方日志记录提供程序](#)。

启用开发人员异常页

`ASPNETCORE_ENVIRONMENT` 环境变量可以添加到 `web.config` 在开发环境中运行应用程序。只要环境不重写中的应用程序启动 `UseEnvironment` 主机生成器中，在设置环境变量允许[开发人员异常页](#) 显示应用程序运行时。

```
<aspNetCore processPath="dotnet"
    arguments=".\\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile=".\\logs\\stdout">
<environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
</environmentVariables>
</aspNetCore>
```

设置环境变量中的 `ASPNETCORE_ENVIRONMENT` 建议只在过渡和测试不公开到 Internet 的服务器上使用。删除从该环境变量 `web.config` 诊断后的文件。有关设置环境变量的信息 `web.config`，请参阅[environmentVariables 子元素的 aspNetCore](#)。

常见的启动错误

请参阅[ASP.NET 核心常见错误参考](#)。中的参考主题介绍了大部分常见防止应用程序启动的问题。

慢速或悬挂应用程序

当应用程序响应速度很慢或挂起的请求时，获取并分析[转储文件](#)。可以使用任何以下工具获取转储文件：

- [ProcDump](#)
- [DebugDiag](#)
- WinDbg:[下载适用于 Windows 调试工具](#)，调试使用 WinDbg

远程调试

请参阅[在 Visual Studio 2017 远程 IIS 计算机上的远程调试 ASP.NET Core](#) Visual Studio 文档中。

Application Insights

Application Insights 提供从托管的 IIS，包括错误日志记录和报告功能的应用程序的遥测。Application Insights 只能针对在应用程序启动时应用的日志记录功能变得可用之后发生的错误报告。有关详细信息，请参阅[Application Insights for ASP.NET Core](#)。

其他故障排除建议

有时会正常运行的应用程序升级任一 .NET 核心 SDK 在应用内的开发计算机或包版本之后立即失败。在某些情况下，不同的包可能在执行主要升级时中断应用。可以按以下说明来修复大部分这些问题：

1. 删除 `bin` 和 `obj` 文件夹。
2. 在包的缓存清除 `*%userprofile%\nuget\包` 和 `%localappdata%\Nuget\v3 缓存*`。
3. 还原和重新生成项目。
4. 确认已完全重新部署应用程序之前删除以前部署到服务器上。

提示

清除包缓存的简便方法是执行 `dotnet nuget locals all --clear` 从命令提示符。

清除包缓存还可通过使用 `nuget.exe` 工具并执行命令 `nuget locals all -clear`。`nuget.exe` 必须从单独获取和不与 Windows 桌面操作系统的捆绑的安装 NuGet 网站。

其他资源

- [ASP.NET Core 中的错误处理简介](#)
- [Azure 应用服务和 IIS 上 ASP.NET Core 的常见错误参考](#)
- [ASP.NET Core 模块配置参考](#)
- [对 Azure 应用服务上的 ASP.NET Core 进行故障排除](#)

ASP.NET 核心模块配置参考

2018/4/27 • 12 min to read • [Edit Online](#)

通过 [Luke Latham](#), [Rick Anderson](#), 和 [Sourabh Shirhatti](#)

本文档将说明了如何配置 ASP.NET 核心模块用于承载 ASP.NET Core 应用。有关 ASP.NET 核心模块和安装说明简介, 请参阅[ASP.NET 核心模块概述](#)。

Web.config 配置

使用配置 ASP.NET 核心模块 `aspNetCore` 部分 `system.webServer` 在站点的节点 `web.config` 文件。

以下 `web.config` 发布文件以便进行 [framework 相关部署](#) 和配置 ASP.NET 核心模块, 以处理站点请求:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\\logs\\stdout" />
  </system.webServer>
</configuration>
```

以下 `web.config` 为发布 [独立的部署](#):

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\\MyApp.exe"
      stdoutLogEnabled="false"
      stdoutLogFile=".\\logs\\stdout" />
  </system.webServer>
</configuration>
```

当应用程序部署到 [Azure App Service](#)、`stdoutLogFile` 路径设置为 `\?\%home%\LogFiles\stdout`。路径将保存到 `stdout` 日志 `LogFiles` 文件夹, 它是一个位置自动创建的服务。

请参阅[子应用程序配置](#)的与配置相关的重要说明 `web.config` 子应用程序中的文件。

AspNetCore 元素的特性

特性	描述	默认
<code>arguments</code>	可选的字符串属性。 可执行文件中指定的自变量 <code>processPath</code> 。	

特性	描述	默认
<code>disableStartUpErrorPage</code>	<p>“真”或“假”。</p> <p>如果为 true, 502.5-进程失败页被禁止显示, 并且 502 状态代码页中配置web.config优先。</p>	<code>false</code>
<code>forwardWindowsAuthToken</code>	<p>“真”或“假”。</p> <p>如果为 true, 该令牌将转发到侦听作为每个请求的标头 MS ASPNETCORE WINAUTHTOKEN 的 %aspnetcore_port% 的子进程。它是该进程可以在每个请求此令牌上调用 CloseHandle 责任。</p>	<code>true</code>
<code>processPath</code>	<p>必需的字符串属性。</p> <p>将启动侦听 HTTP 请求的进程的可执行文件的路径。支持相对路径。如果路径以开始 ., 考虑使用该路径是相对站点根。</p>	
<code>rapidFailsPerMinute</code>	<p>可选的整数属性。</p> <p>指定在指定的进程的次数processPath允许每分钟崩溃。如果超出此限制, 该模块将停止启动剩余秒数的进程。</p>	<code>10</code>
<code>requestTimeout</code>	<p>可选的 timespan 属性。</p> <p>指定为其 ASP.NET 核心模块等待侦听 %aspnetcore_port% 的进程响应的持续时间。</p> <p>中附带的版本的 ASP.NET 核心 2.0 或更早版本, ASP.NET 核心模块的版本 <code>requestTimeout</code> 必须指定整分钟数, 否则它将默认为 2 分钟。</p>	<code>00:02:00</code>
<code>shutdownTimeLimit</code>	<p>可选的整数属性。</p> <p>正常关闭的可执行文件的模块等待的秒数的持续时间时 <code>app_offline.htm</code> 检测到文件。</p>	<code>10</code>

特性	描述	默认
<code>startupTimeLimit</code>	<p>可选的整数属性。</p> <p>持续时间以启动侦听端口的进程的可执行文件的模块等待的秒数。如果超过此时间限制，该模块可终止进程。模块将尝试重新启动该过程，在它接收新请求，并将继续尝试重新启动此过程在后续的传入请求，除非应用程序启动失败时rapidFailsPerMinute次数在最后一个滚动分钟。</p>	<code>120</code>
<code>stdoutLogEnabled</code>	<p>可选布尔属性。</p> <p>如果为 true，stdout和stderr中指定的进程的processPath重定向到中指定的文件stdoutLogFile。</p>	<code>false</code>
<code>stdoutLogFile</code>	<p>可选的字符串属性。</p> <p>为其指定的相对或绝对文件路径stdout和stderr中指定的进程从processPath记录。相对路径是相对于站点的根目录。从任何路径<code>.</code>是相对于站点根和所有其他路径被视为绝对路径。路径中提供的任何文件夹必须存在于要创建的日志文件的模块的顺序。使用下划线分隔符、时间戳、进程 ID 和文件扩展名 (<i>log</i>) 添加到最后一个段stdoutLogFile路径。如果<code>.\logs\stdout</code>提供一个值，作为示例 stdout 日志另存为<code>stdout_20180205194132_1934.log</code>中日志文件夹保存在 19:41:32 并且进程 ID 为 1934 年 2/5/2018 年。</p>	<code>aspnetcore-stdout</code>

特性	描述	默认
<code>arguments</code>	<p>可选的字符串属性。</p> <p>可执行文件中指定的自变量processPath。</p>	
<code>disableStartUpErrorPage</code>	<p>“真”或“假”。</p> <p>如果为 true，502.5-进程失败页被禁止显示，并且 502 状态代码页中配置web.config优先。</p>	<code>false</code>

特性	描述	默认
<code>forwardWindowsAuthToken</code>	<p>“真”或“假”。</p> <p>如果为 true, 该令牌将转发到侦听作为每个请求的标头 MS ASPNETCORE WINAUTHTOKEN 的 %aspnetcore_port% 的子进程。它是该进程可以在每个请求此令牌上调用 CloseHandle 责任。</p>	<code>true</code>
<code>processPath</code>	<p>必需的字符串属性。</p> <p>将启动侦听 HTTP 请求的进程的可执行文件的路径。支持相对路径。 如果路径以开始 ., 考虑使用该路径是相对站点根。</p>	
<code>rapidFailsPerMinute</code>	<p>可选的整数属性。</p> <p>指定在指定的进程的次数 processPath 允许每分钟崩溃。如果超出此限制, 该模块将停止启动剩余秒数的进程。</p>	<code>10</code>
<code>requestTimeout</code>	<p>可选的 timespan 属性。</p> <p>指定为其 ASP.NET 核心模块等待侦听 %aspnetcore_port% 的进程响应的持续时间。</p> <p>中附带的版本的 ASP.NET 核心 2.1 或更高版本, ASP.NET 核心模块的版本 <code>requestTimeout</code> 中小时、分钟和秒为单位指定。</p>	<code>00:02:00</code>
<code>shutdownTimeLimit</code>	<p>可选的整数属性。</p> <p>正常关闭的可执行文件的模块等待的秒数的持续时间时 <code>app_offline.htm</code> 检测到文件。</p>	<code>10</code>
<code>startupTimeLimit</code>	<p>可选的整数属性。</p> <p>持续时间以启动侦听端口的进程的可执行文件的模块等待的秒数。如果超过此时间限制, 该模块可终止进程。模块将尝试重新启动该过程, 在它接收新请求, 并将继续尝试重新启动此过程在后续的传入请求, 除非应用程序启动失败时 rapidFailsPerMinute 次数在最后一个滚动分钟。</p>	<code>120</code>

特性	描述	默认
<code>stdoutLogEnabled</code>	可选布尔属性。 如果为 true, <code>stdout</code> 和 <code>stderr</code> 中指定的进程的 <code>processPath</code> 重定向到中指定的文件 <code>stdoutLogFile</code> 。	<code>false</code>
<code>stdoutLogFile</code>	可选的字符串属性。 为其指定的相对或绝对文件路径 <code>stdout</code> 和 <code>stderr</code> 中指定的进程从 <code>processPath</code> 记录。相对路径是相对于站点的根目录。从任何路径 <code>.</code> 是相对于站点根和所有其他路径被视为绝对路径。路径中提供的任何文件夹必须存在于要创建的日志文件的模块的顺序。使用下划线分隔符、时间戳、进程 ID 和文件扩展名 (<code>log</code>) 添加到最后一个段 <code>stdoutLogFile</code> 路径。如果 <code>.\logs\stdout</code> 提供一个值, 作为示例 stdout 日志另存为 <code>stdout_20180205194132_1934.log</code> 中日志文件夹保存在 19:41:32 并且进程 ID 为 1934 年 2/5/2018 年。	<code>aspnetcore-stdout</code>

设置环境变量

环境变量可以为中的过程指定`processPath`属性。指定的环境变量`environmentVariable`的子元素`environmentVariables`集合元素。在本部分中设置的环境变量优先于系统环境变量。

下面的示例设置两个环境变量。`ASPNETCORE_ENVIRONMENT` 配置应用程序的环境以`Development`。开发人员可能将此值暂时设置`web.config`文件以便强制[开发人员异常页](#)以便在调试应用程序异常时加载。`CONFIG_DIR` 是一种用户定义的环境变量中, 开发人员曾读取启动窗体中用于加载应用程序的配置文件的路径上的值的代码。

```
<aspNetCore processPath="dotnet"
            arguments=".\\MyApp.dll"
            stdoutLogEnabled="false"
            stdoutLogFile="\\?\%home%\LogFiles\stdout">
<environmentVariables>
  <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  <environmentVariable name="CONFIG_DIR" value="f:\\application_config" />
</environmentVariables>
</aspNetCore>
```

警告

只能设置`ASPNETCORE_ENVIRONMENT` environment 变量`Development`过渡和测试不到不受信任的网络, 例如 Internet 可访问的服务器上。

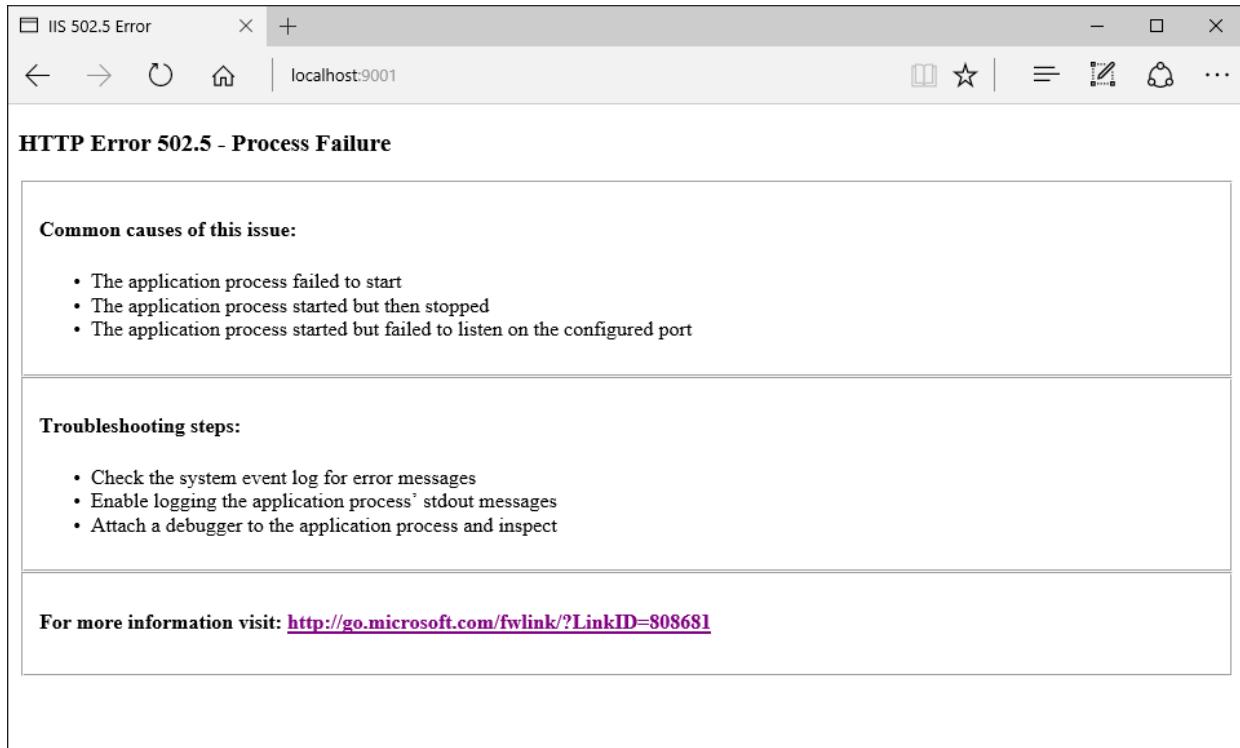
app_offline.htm

如果具有名称的文件`app_offline.htm`中检测到一个应用程序的根目录下 ASP.NET 核心模块尝试正常关闭应用程序, 并停止处理传入请求。如果应用程序中定义的秒数后仍在运行`shutdownTimeLimit`, ASP.NET 核心模块将终止正在运行的进程。

虽然 `app_offline.htm` 存在文件，则 ASP.NET 核心模块响应请求通过发回的内容 `app_offline.htm` 文件。当 `app_offline.htm` 删除文件，则下一个请求启动应用程序。

启动错误页

如果 ASP.NET 核心模块无法启动后端进程或后端进程启动但不能在配置的端口上侦听 502.5 进程失败状态代码页将出现。若要禁止显示此页并还原为默认 IIS 502 状态代码页，请使用 `disableStartupErrorPage` 属性。有关配置自定义错误消息的详细信息，请参阅 [HTTP 错误 <httpErrors>](#)。



日志创建和重定向

ASP.NET 核心模块将重定向 `stdout` 和 `stderr` 的日志磁盘如果 `stdoutLogEnabled` 和 `stdoutLogFile` 属性 `aspNetCore` 元素设置。中的任何文件夹 `stdoutLogFile` 路径必须存在于要创建的日志文件的模块的顺序。应用程序池必须具有写入访问权限日志将写入其中的位置（使用 `IIS AppPool\<app_pool_name>` 提供写入权限）。

日志不轮换，除非进程回收/重新启动时发生。它负责的托管商来限制日志使用的磁盘空间。

应用程序启动问题的故障排除，仅建议使用 `stdout` 日志。不要使用用于常规应用程序日志记录的 `stdout` 日志。对于例程日志记录在 ASP.NET Core 应用程序，使用限制日志文件大小和旋转日志的日志记录库。有关详细信息，请参阅 [第三方日志记录提供程序](#)。

创建日志文件时，将自动添加时间戳和文件扩展名。日志文件名称由后面追加时间戳、进程 ID 和文件扩展名 (`.log`) 到的最后一段 `stdoutLogFile` 路径（通常 `stdout`）由下划线分隔。如果 `stdoutLogFile` 路径以结束 `stdout`, pid 为 1934 在 19:42:32 2/5/2018 年上创建的应用程序日志中的文件名称 `stdout_20180205194132_1934.log`。

下面的示例 `aspNetCore` 元素会配置的 Azure App Service 中承载的应用的标准输出日志记录。本地路径或网络共享路径是可接受的本地日志记录。确认应用程序池用户标识有权写入提供的路径。

```
<aspNetCore processPath="dotnet"
  arguments=".\\MyApp.dll"
  stdoutLogEnabled="true"
  stdoutLogFile="\\?\%home%\LogFiles\stdout">
</aspNetCore>
```

请参阅 [web.config 配置](#) 有关的示例 `aspNetCore` 中的元素 `web.config` 文件。

代理配置使用 HTTP 协议和配对令牌

在 ASP.NET 核心模块和 Kestrel 之间创建的代理服务器使用 HTTP 协议。使用 HTTP 是一种性能优化，其中模块和 Kestrel 之间的通信发生在上环回地址从网络接口中移出。没有任何风险的窃听模块和 Kestrel 从服务器中移出的位置之间的通信。

配对令牌用于保证 Kestrel 收到的请求已由 IIS 代理且不来自某些其他源。创建并设置环境变量到配对的令牌 (`ASPNETCORE_TOKEN`) 由模块。此外，配对令牌还设置到每个代理请求的标头 (`MS.AspNetCoreToken`)。IIS 中间件检查它所接收的每个请求，以确认配对令牌标头值与环境变量值相匹配。如果令牌值不匹配，则将记录请求并拒绝该请求。配对的令牌的环境变量和模块和 Kestrel 之间的通信无法访问从服务器中移出的位置。如果不知道配对令牌值，攻击者就无法提交绕过 IIS 中间件中的检查的请求。

ASP.NET 核心模块与 IIS 共享配置

ASP.NET 核心模块安装程序使用的特权运行系统帐户。安装程序的本地系统帐户不具有修改权限使用 IIS 共享配置的共享路径，因为达到拒绝访问错误时尝试进行配置中的模块设置 `applicationHost.config` 共享上。在将非 IIS 共享配置，请按照下列步骤：

1. 禁用 IIS 共享的配置。
2. 运行安装程序。
3. 导出已更新 `applicationHost.config` 到共享的文件。
4. 重新启用 IIS 共享的配置。

模块版本和承载捆绑安装程序日志

若要确定已安装 ASP.NET 核心模块的版本：

1. 在托管系统上，导航到 `%windir%\System32\inetsrv`。
2. 找到 `aspnetcore.dll` 文件。
3. 右键单击该文件并选择 **属性** 从上下文菜单。
4. 选择 **详细信息** 选项卡。**文件版本** 和 **产品版本** 表示已安装的模块版本。

在找到承载捆绑安装程序日志模块 `c:\用户\%username%\AppData\本地\Temp`。将文件命名为 `dd_DotNetCoreWinSvrHosting_<时间戳>_000_AspNetCoreModule_x64.log`。

模块、架构和配置文件位置

模块

IIS (x86/amd64):

- `%windir%\System32\inetsrv\aspnetcore.dll`
- `%windir%\SysWOW64\inetsrv\aspnetcore.dll`

IIS Express (x86/amd64):

- `%ProgramFiles%\IIS Express\aspnetcore.dll`
- `%ProgramFiles(x86)%\IIS Express\aspnetcore.dll`

架构

IIS

- `%windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml`

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml

配置

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- .vs\config\applicationHost.config

可以通过搜索找到文件`aspnetcore.dll`中`applicationHost.config`文件。IIS express, `applicationHost.config`文件不存在默认情况下。在创建文件`<application_root>\.vs\`配置时从任何 web 应用程序项目开始在 Visual Studio 解决方案。

Visual Studio 中针对 ASP.NET Core 的开发时 IIS 支持

2018/5/18 • 3 min to read • [Edit Online](#)

通过Sourabh Shirhatti和Luke Latham

本指南介绍了Visual Studio支持的用于调试在IIS后面运行Windows Server上的ASP.NET Core应用。本主题将指导完成启用此功能，并设置项目。

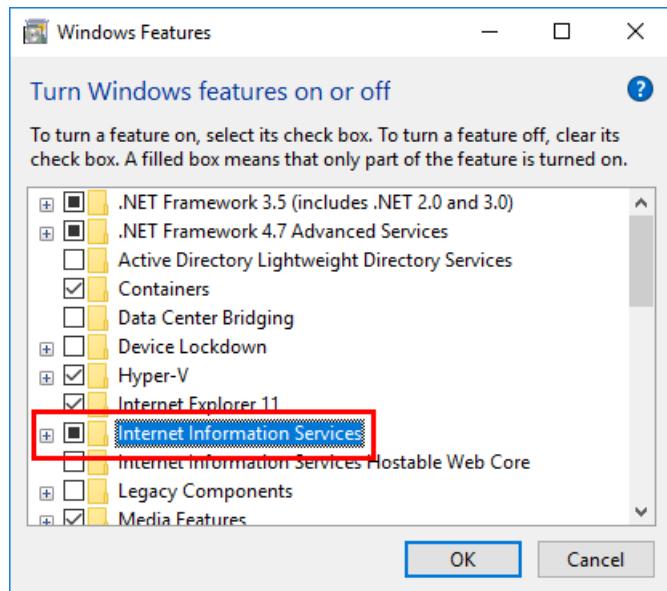
系统必备

[Visual Studio for Windows](#)

- **ASP.NET and web development workload**
- **.NET Core cross-platform development workload**
- X.509 security certificate

启用 IIS

1. 导航到“控制面板”>“程序”>“程序和功能”>“打开或关闭Windows功能”(位于屏幕左侧)。
2. 选择**Internet Information Services**复选框。



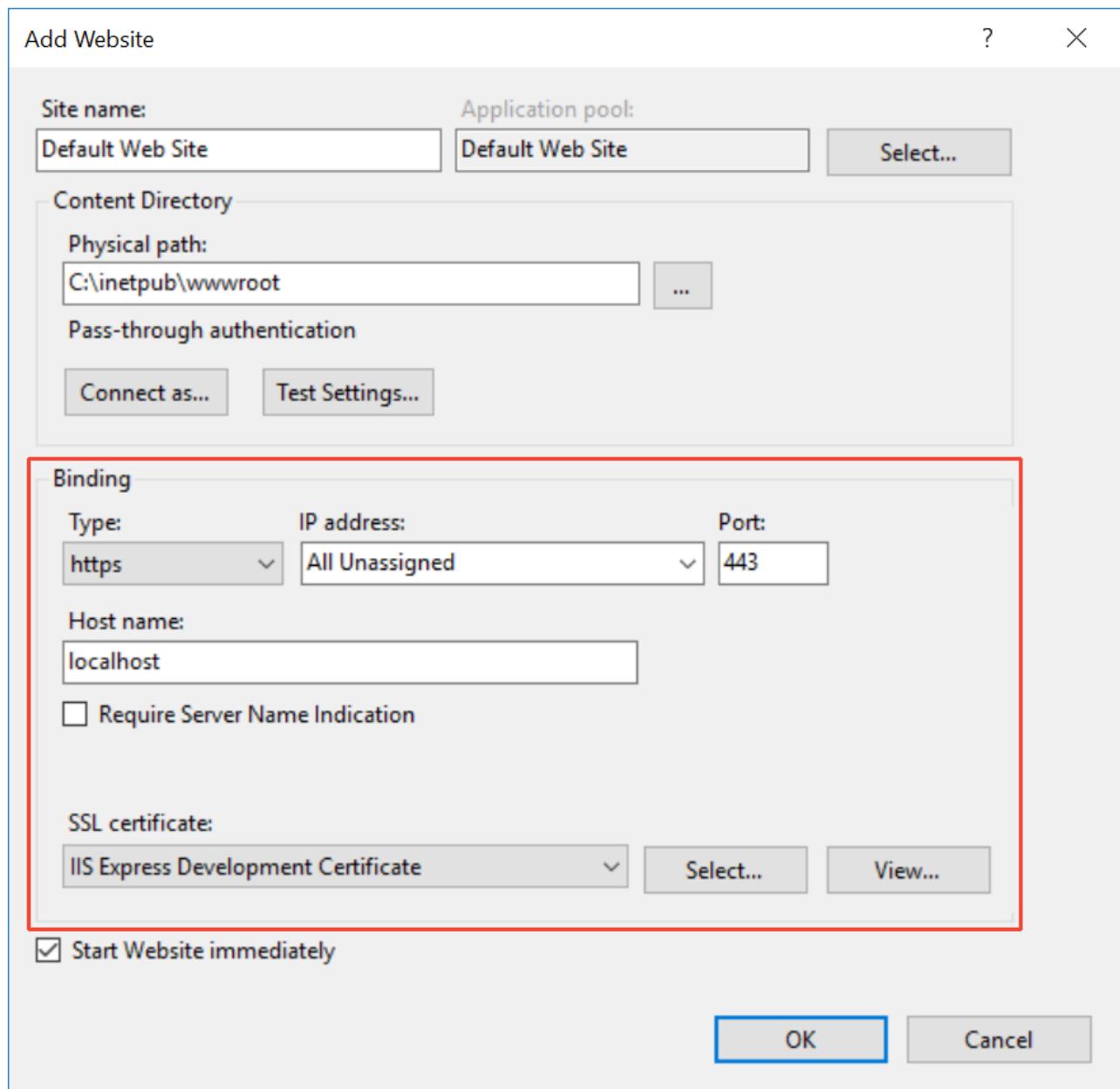
IIS 安装可能需要重新启动系统。

配置 IIS

IIS 必须拥有的网站进行以下配置：

- 应用程序的发布配置文件 URL 主机名称相匹配的主机名称。
- 端口 443 与分配的证书的绑定。

例如，**主机名**添加的网站设置为"localhost"(启动配置文件将还使用"localhost"本主题中的更高版本)。端口设置为"443"(HTTPS)。**IIS Express**开发证书分配给网站，但任何有效的证书工作原理：

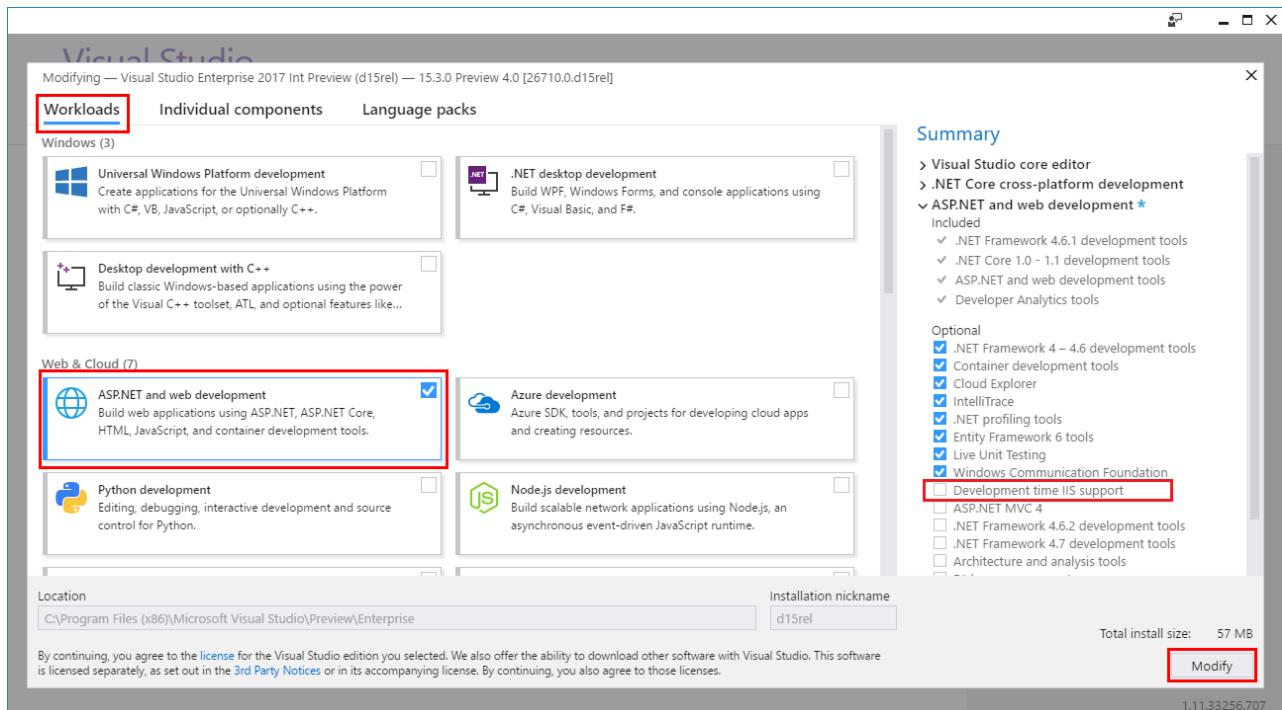


如果 IIS 安装已具有**Default Web Site** 使用与应用程序的发布配置文件 URL 主机名匹配的主机名：

- 添加为端口 443 (HTTPS) 的端口绑定。
- 向网站分配一个有效的证书。

启用 Visual Studio 中的开发时间 IIS 支持

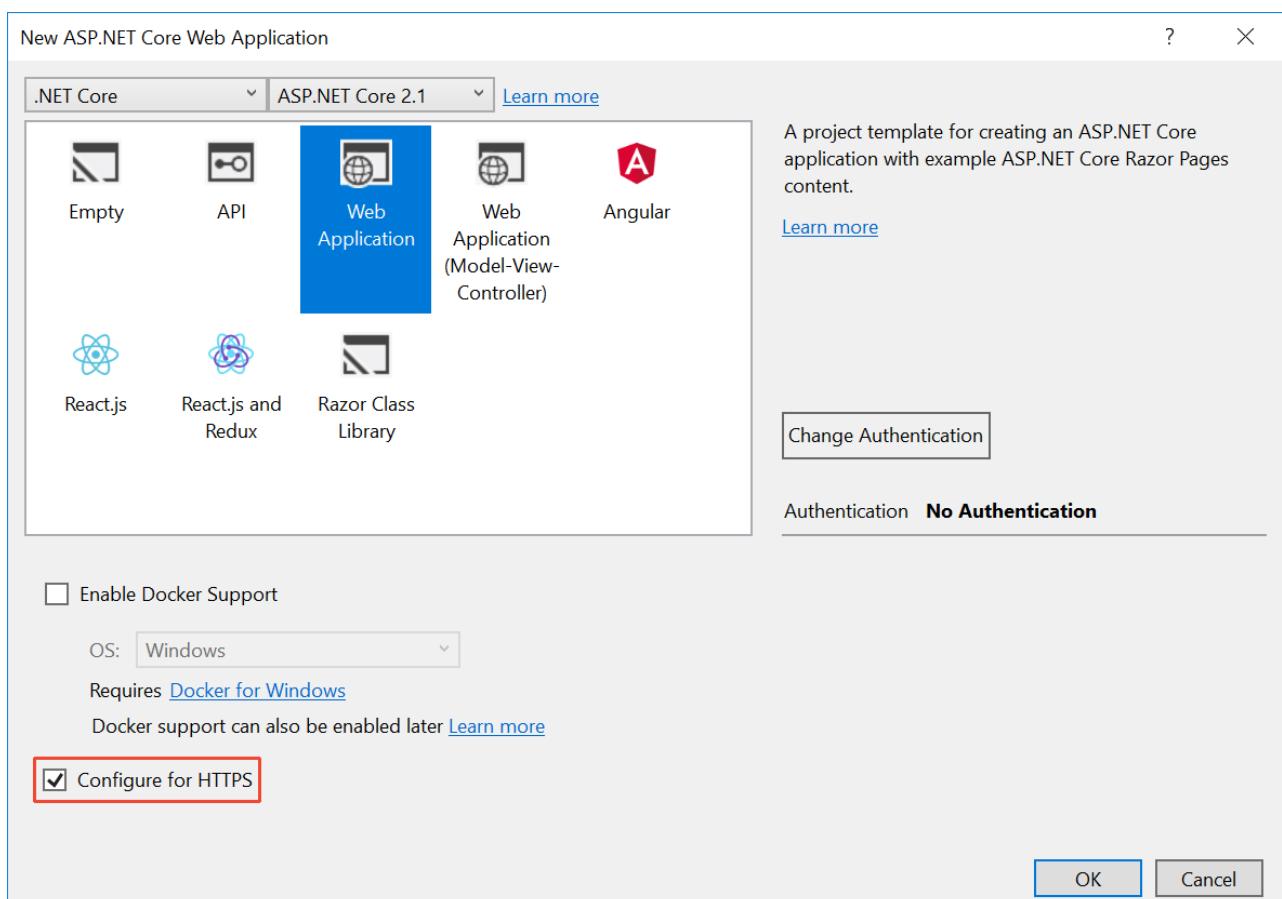
- 启动 Visual Studio 安装程序。
- 选择**IIS 支持的开发时间组件**。列出为可选中了该组件摘要面板**ASP.NET** 和 **web 开发工作负载**。组件安装**ASP.NET 核心模块**，即反向代理配置中运行 ASP.NET Core 后面 IIS 的应用所需的本机 IIS 模块。



配置项目

HTTPS 重定向

对于新项目中，选中复选框以针对 **HTTPS 配置** 中新 **ASP.NET 核心 Web 应用程序** 窗口：



在现有项目中，使用 **HTTPS 重定向** 中的中间件 `Startup.Configure` 通过调用 `UseHttpsRedirection` 扩展方法：

```

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

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

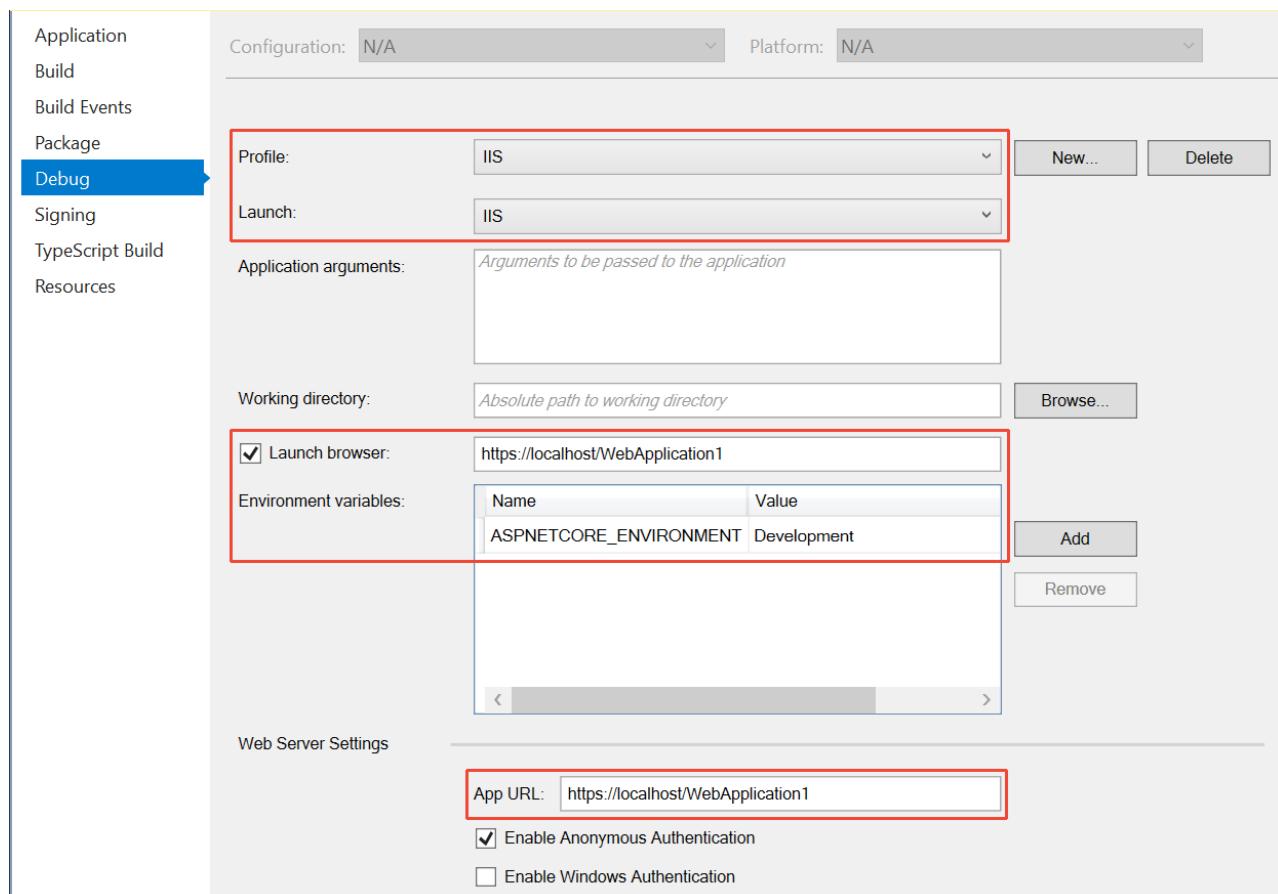
    app.UseMvc();
}

```

IIS 启动配置文件

创建新的发布配置文件添加开发时间 IIS 支持：

- 有关配置文件，选择新建按钮。在弹出窗口中将该配置文件“IIS”。选择确定创建配置文件。
- 有关启动设置中选择IIS从列表中。
- 选中的复选框启动浏览器和提供的终结点 URL。使用 HTTPS 协议。本示例使用 <https://localhost/WebApplication1>。
- 在环境变量部分中，选择添加按钮。环境变量提供的密钥 ASPNETCORE_ENVIRONMENT 和的值 Development。
- 在Web 服务器设置区域中，设置应用程序 URL。本示例使用 <https://localhost/WebApplication1>。
- 保存配置文件。

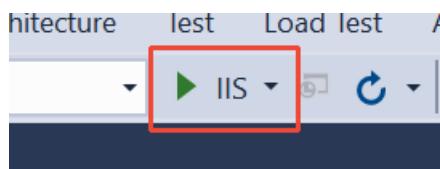


或者，手动添加到的启动配置文件[launchSettings.json](#)应用程序中的文件：

```
{  
  "iisSettings": {  
    "windowsAuthentication": false,  
    "anonymousAuthentication": true,  
    "iis": {  
      "applicationUrl": "https://localhost/WebApplication1",  
      "sslPort": 0  
    }  
  },  
  "profiles": {  
    "IIS": {  
      "commandName": "IIS",  
      "launchBrowser": true,  
      "launchUrl": "https://localhost/WebApplication1",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    }  
  }  
}
```

运行项目

在 VS UI 中，设置为运行按钮**IIS**分析，并选择按钮以启动应用程序：



如果未以管理员身份运行 visual Studio 可能会提示重新启动。如果出现提示，请重启 Visual Studio。

如果将使用的不受信任的开发证书，浏览器可能需要你针对不受信任的证书创建例外。

其他资源

- [使用 IIS 在 Windows 上托管 ASP.NET Core](#)
- [ASP.NET Core 模块简介](#)
- [ASP.NET Core 模块配置参考](#)
- [Enforce HTTPS](#)

与 ASP.NET 核心的 IIS 模块

2018/4/27 • 6 min to read • [Edit Online](#)

作者:Luke Latham

反向代理配置中，由 IIS 承载 ASP.NET Core 应用。一些本机 IIS 模块和所有 IIS 管理模块不是可用于处理请求的 ASP.NET Core 应用。在许多情况下，ASP.NET Core 提供 IIS 本机和托管模块的功能的替代方法。

本机模块

此表指示在对 ASP.NET Core 应用的反向代理请求上正常运行的本机 IIS 模块。

模块	与 ASP.NET CORE 应用功能	ASP.NET 核心选项
匿名身份验证 <code>AnonymousAuthenticationModule</code>	是	
基本身份验证 <code>BasicAuthenticationModule</code>	是	
客户端证书映射身份验证 <code>CertificateMappingAuthenticationModule</code>	是	
CGI <code>CgiModule</code>	否	
配置验证 <code>ConfigurationValidationModule</code>	是	
HTTP 错误 <code>CustomErrorModule</code>	否	状态代码页中间件
自定义日志记录 <code>CustomLoggingModule</code>	是	
默认文档 <code>DefaultDocumentModule</code>	否	默认文件中间件
摘要式身份验证 <code>DigestAuthenticationModule</code>	是	
目录浏览 <code>DirectoryListingModule</code>	否	目录浏览中间件
动态压缩 <code>DynamicCompressionModule</code>	是	响应压缩中间件
跟踪 <code>FailedRequestsTracingModule</code>	是	ASP.NET 核心日志记录

模块	与 ASP.NET CORE 应用功能	ASP.NET 核心选项
文件缓存 <code>FileCacheModule</code>	否	响应缓存中间件
HTTP 缓存功能 <code>HttpCacheModule</code>	否	响应缓存中间件
HTTP 日志记录 <code>HttpLoggingModule</code>	是	ASP.NET 核心日志记录 实现: elmah.io , Loggr , NLog , Serilog
HTTP 重定向 <code>HttpRedirectionModule</code>	是	URL 重写中间件
IIS 客户端证书映射身份验证 <code>IISCertificateMappingAuthenticationModule</code>	是	
IP 和域限制 <code>IpRestrictionModule</code>	是	
ISAPI 筛选器 <code>IsapiFilterModule</code>	是	中间件
ISAPI <code>IsapiModule</code>	是	中间件
协议支持 <code>ProtocolSupportModule</code>	是	
请求筛选 <code>RequestFilteringModule</code>	是	URL 重写中间件 <code>IRule</code>
请求监视器 <code>RequestMonitorModule</code>	是	
URL 重写 <code>RewriteModule</code>	是的†	URL 重写中间件
服务器端包括 <code>ServerSideIncludeModule</code>	否	
静态压缩 <code>StaticCompressionModule</code>	否	响应压缩中间件
静态内容 <code>StaticFileModule</code>	否	静态文件中间件
令牌缓存 <code>TokenCacheModule</code>	是	
URI 缓存 <code>UriCacheModule</code>	是	

模块	与 ASP.NET CORE 应用功能	ASP.NET 核心选项
URL 授权 UrlAuthorizationModule	是	ASP.NET 核心标识
Windows 身份验证 WindowsAuthenticationModule	是	

+ URL 重写模块的 `isFile` 和 `isDirectory` 匹配类型不使用 ASP.NET Core 应用中的更改由于[目录结构](#)。

托管的模块

托管模块是不能与托管的 ASP.NET Core 应用时应用程序池的.NET CLR 版本设置为无托管代码。ASP.NET Core 提供几种情况，中间件的备选项。

模块	ASP.NET 核心选项
AnonymousIdentification	
DefaultAuthentication	
FileAuthorization	
FormsAuthentication	Cookie 身份验证中间件
OutputCache	响应缓存中间件
配置文件	
RoleManager	
ScriptModule-4.0	
会话	会话中间件
UrlAuthorization	
UrlMappingsModule	URL 重写中间件
UrlRoutingModule-4.0	ASP.NET 核心标识
WindowsAuthentication	

IIS 管理器应用程序更改

当使用 IIS 管理器配置设置，`web.config` 更改应用程序文件。如果部署的应用程序并包括`web.config`，做使用 IIS 管理器的任何更改将被覆盖的部署`web.config`文件。如果在更改服务器的`web.config`文件中，复制已更新`web.config`于立即出现在本地项目的服务器上的文件。

禁用 IIS 模块

如果必须为一个应用程序的补充，到应用程序的禁用的服务器级别配置的 IIS 模块`web.config`文件可以禁用该模块。将模块保留原位和停用它（如果可用）使用配置设置，或者从应用程序中删除模块。

模块停用

多个模块提供配置设置，从而使这些要禁用但不从应用程序移除模块。这是最简单且最快速方式停用模块。例如，可以使用禁用 HTTP 重定向模块 `<httpRedirect>` 中的元素 `web.config`:

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

禁用使用配置设置的模块的详细信息，请按照中的链接子元素部分 IIS `<system.webServer>`。

模块删除

如果选择加入以移除模块中的设置与 `web.config`、解锁模块和解锁**`<模块>`** 部分 `web.config` 第一个：

1. 解锁的服务器级别的模块。选择在 IIS 管理器中的 IIS 服务器连接侧栏。打开模块中 IIS 区域。在列表中选择该模块。在操作侧栏右侧，选择解锁。解锁任意多个模块，因为你打算移除 `web.config` 更高版本。
2. 部署应用程序而无需**`<模块>`** 主题中 `web.config`。如果应用程序部署与 `web.config` 包含**`<模块>`** 部分不具有解锁部分首先在 IIS 管理器，配置管理器的情况下将引发异常无法解除锁定节。因此，部署应用程序而无需**`<模块>`** 部分。
3. 解锁**`<模块>`** 部分 `web.config`。在连接栏中，选择在网站站点。在管理区域中，打开配置编辑器。使用导航控件选择 `system.webServer/modules` 部分。在操作侧栏右侧，选择解锁部分。
4. 此时，`<模块>` 部分可以添加到 `web.config` 文件**`<删除>`** 要移除从模块元素应用程序。多个**`<删除>`** 可添加元素以移除多个模块。如果 `web.config` 服务器上进行更改，立即对相同的更改进行项目的 `web.config` 文件在本地。删除这种方式的模块不会影响服务器上的其他应用的模块使用。

```
<configuration>
  <system.webServer>
    <modules>
      <remove name="MODULE_NAME" />
    </modules>
  </system.webServer>
</configuration>
```

IIS 模块也可能会删除与 `Appcmd.exe`。提供 `MODULE_NAME` 和 `APPLICATION_NAME` 命令中：

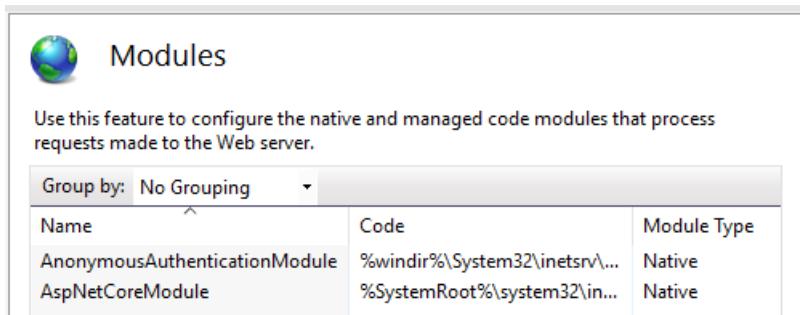
```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

例如，删除 `DynamicCompressionModule` 从默认网站：

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule /app.name:"Default Web Site"
```

最小模块配置

若要运行 ASP.NET Core 应用所需的唯一模块是匿名身份验证模块和 ASP.NET 核心模块。



The screenshot shows the 'Modules' section of the IIS Manager. At the top, there's a message: 'Use this feature to configure the native and managed code modules that process requests made to the Web server.' Below this is a table with two rows. The columns are 'Name', 'Code', and 'Module Type'. The first row shows 'AnonymousAuthenticationModule' with code '%windir%\System32\inetsrv\...' and type 'Native'. The second row shows 'AspNetCoreModule' with code '%SystemRoot%\system32\in...' and type 'Native'. A dropdown menu 'Group by:' is set to 'No Grouping'.

Name	Code	Module Type
AnonymousAuthenticationModule	%windir%\System32\inetsrv\...	Native
AspNetCoreModule	%SystemRoot%\system32\in...	Native

URI 缓存模块 (`UriCacheModule`) 在 URL 级别的缓存网站配置为允许 IIS。如果没有此模块, IIS 必须读取和分析对每个请求的配置, 即使重复请求相同的 URL。解析配置每个请求将导致对显著的性能产生负面影响。虽然 *URI 缓存模块并非是严格要求为托管的 ASP.NET Core 应用程序运行, 我们建议, 所有 ASP.NET Core 部署都启用 URI 缓存模块。*

HTTP 缓存模块 (`HttpCacheModule`) 实现 IIS 输出缓存以及缓存 HTTP.sys 高速缓存中的项的逻辑。如果没有此模块内容不再在内核模式下, 缓存和缓存配置文件将被忽略。通常移除 HTTP 缓存模块具有对性能和资源使用情况的负面影响。虽然 *HTTP 缓存模块并非是严格要求为托管的 ASP.NET Core 应用程序运行, 我们建议, 所有 ASP.NET Core 部署都启用 HTTP 缓存模块。*

其他资源

- [使用 IIS 在 Windows 上进行托管](#)
- [简介 IIS 体系结构: 在 IIS 中的模块](#)
- [IIS 模块概述](#)
- [自定义 IIS 7.0 角色和模块](#)
- [IIS <system.webServer>](#)

在 Windows 服务中的主机 ASP.NET 核心

2018/5/17 • 5 min to read • [Edit Online](#)

作者: Tom Dykstra

没有使用 IIS 是在运行承载 ASP.NET Core 应用在 Windows 上的推荐的方式[Windows 服务](#)。当托管为 Windows 服务，应用程序可以自动启动之后启动重新启动和崩溃而无需人工干预。

[查看或下载示例代码\(如何下载\)](#)。有关说明如何运行示例应用程序，请参阅示例的*README.md*文件。

系统必备

- 应用程序必须在.NET Framework 运行时上运行。在 *.csproj* 文件中，指定为相应值[TargetFramework](#)和[RuntimeIdentifier](#)。以下是一个示例：

```
<PropertyGroup>
  <TargetFramework>net461</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
</PropertyGroup>
```

当在 Visual Studio 中创建项目，请使用[ASP.NET 核心应用程序 \(.NET Framework\) 模板](#)。

- 如果应用程序接收来自 Internet (而不仅仅是从内部网络) 的请求，它必须使用[HTTP.sys](#) web 服务器 (以前称为[WebListener](#)对于 ASP.NET Core 1.x 应用程序) 而不是[Kestrel](#)。IIS 被建议用于为反向代理服务器的 Kestrel 边缘部署。有关详细信息，请参阅[何时结合使用 Kestrel 和反向代理](#)。

入门

本部分介绍将现有的 ASP.NET Core 项目设置为在服务中运行所需的最小更改。

1. 安装 NuGet 程序包[Microsoft.AspNetCore.Hosting.WindowsServices](#)。

2. 进行中的以下更改 `Program.Main`：

- 调用 `host.RunAsService` 而不是 `host.Run`。
- 如果代码调用 `UseContentRoot`，到发布位置而不是使用路径 `Directory.GetCurrentDirectory()`。
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public static void Main(string[] args)
{
    var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
    var pathToContentRoot = Path.GetDirectoryName(pathToExe);

    var host = WebHost.CreateDefaultBuilder(args)
        .UseContentRoot(pathToContentRoot)
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    host.RunAsService();
}
```

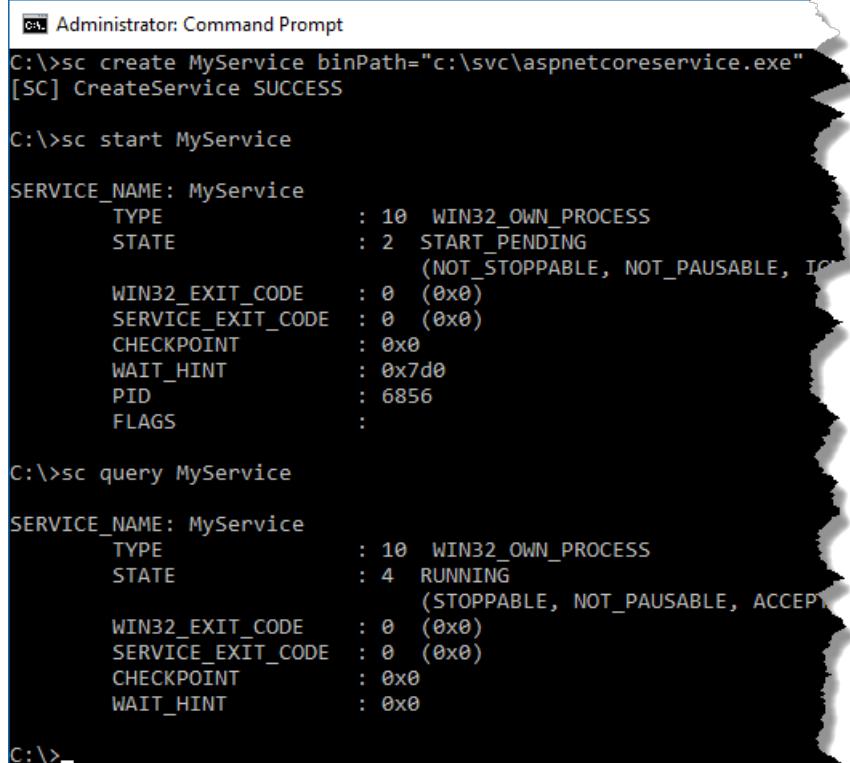
3. 将应用发布到的文件夹。使用[dotnet 发布](#)或[Visual Studio 发布配置文件](#), 它发布到的文件夹。

4. 通过创建和启动服务测试。

使用管理特权才能使用打开命令行界面[sc.exe](#)命令行工具来创建和启动服务。如果该服务名为 MyService, 发布到 `c:\svc`, 以及名为 AspNetCoreService, 命令:

```
sc create MyService binPath="c:\svc\aspnetcoreservice.exe"
sc start MyService
```

`binPath` 值是应用程序的可执行文件, 其中包括可执行文件名称的路径。



```
C:\>sc create MyService binPath="c:\svc\aspnetcoreservice.exe"
[SC] CreateService SUCCESS

C:\>sc start MyService

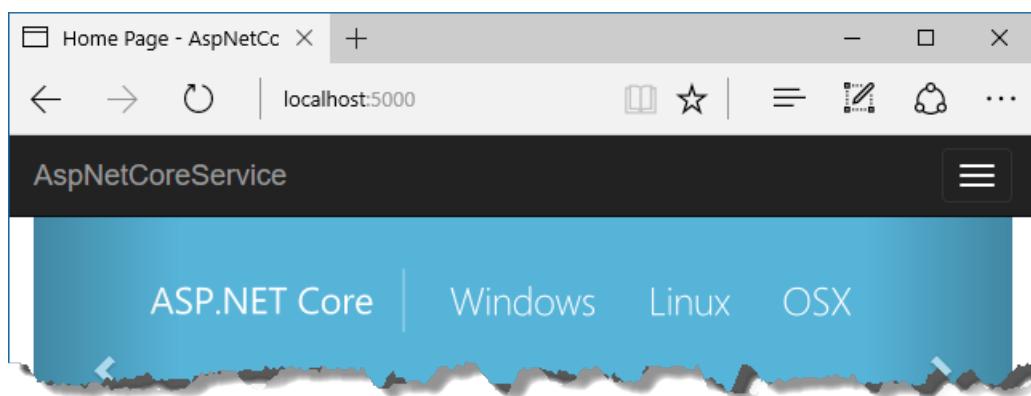
SERVICE_NAME: MyService
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 2   START_PENDING
                          (NOT_STOPPABLE, NOT_PAUSABLE, IGNORE_HANG)
    WIN32_EXIT_CODE    : 0   (0x0)
    SERVICE_EXIT_CODE  : 0   (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x7d0
    PID                : 6856
    FLAGS              :

C:\>sc query MyService

SERVICE_NAME: MyService
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 4   RUNNING
                          (STOPPABLE, NOT_PAUSABLE, ACCEPT_HANG)
    WIN32_EXIT_CODE    : 0   (0x0)
    SERVICE_EXIT_CODE  : 0   (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0

C:\>
```

在这些命令完成后, 浏览到相同的路径作为控制台应用程序运行时(默认情况下, <http://localhost:5000>):



提供服务的外部运行的方法

很容易地测试和调试时运行外部服务, 因此通常将调用的代码添加 `RunAsService` 仅在某些情况下。例如, 应用程序可以作为包含的控制台应用程序运行 `--console` 命令行自变量, 或如果附加调试器:

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public static void Main(string[] args)
{
    bool isService = true;
    if (Debugger.IsAttached || args.Contains("--console"))
    {
        isService = false;
    }

    var pathToContentRoot = Directory.GetCurrentDirectory();
    if (isService)
    {
        var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
        pathToContentRoot = Path.GetDirectoryName(pathToExe);
    }

    var host = WebHost.CreateDefaultBuilder(args)
        .UseContentRoot(pathToContentRoot)
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    if (isService)
    {
        host.RunAsService();
    }
    else
    {
        host.Run();
    }
}

```

处理停止和启动事件

若要处理 `OnStarting`，`OnStarted`，和 `OnStopping` 事件，进行以下其他更改：

1. 创建一个类，派生自 `WebHostService`：

```

internal class CustomWebHostService : WebHostService
{
    public CustomWebHostService(IWebHost host) : base(host)
    {
    }

    protected override void OnStarting(string[] args)
    {
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        base.OnStopping();
    }
}

```

2. 创建扩展方法 `IWebHost`，通过自定义 `WebHostService` 到 `ServiceBase.Run`：

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

3. 在 `Program.Main`，调用新的扩展方法，`RunAsCustomService`，而不是 `RunAsService`：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public static void Main(string[] args)
{
    bool isService = true;
    if (Debugger.IsAttached || args.Contains("--console"))
    {
        isService = false;
    }

    var pathToContentRoot = Directory.GetCurrentDirectory();
    if (isService)
    {
        var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
        pathToContentRoot = Path.GetDirectoryName(pathToExe);
    }

    var host = WebHost.CreateDefaultBuilder(args)
        .UseContentRoot(pathToContentRoot)
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    if (isService)
    {
        host.RunAsCustomService();
    }
    else
    {
        host.Run();
    }
}
```

如果自定义 `WebHostService` 代码需要从依赖关系注入（如记录器）服务，以获取从 `Services` 属性 `IWebHost`：

```
internal class CustomWebHostService : WebHostService
{
    private ILogger _logger;

    public CustomWebHostService(IWebHost host) : base(host)
    {
        _logger = host.Services.GetRequiredService<ILogger<CustomWebHostService>>();
    }

    protected override void OnStarting(string[] args)
    {
        _logger.LogDebug("OnStarting method called.");
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        _logger.LogDebug("OnStarted method called.");
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        _logger.LogDebug("OnStopping method called.");
        base.OnStopping();
    }
}
```

代理服务器和负载均衡器方案

与请求来自 Internet 或公司网络进行交互和代理服务器后面或负载平衡器的服务可能需要其他配置。有关详细信息，请参阅[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

鸣谢

本文是已发布的源的帮助下编写的：

- [作为 Windows 服务承载 ASP.NET 核心](#)
- [如何托管 Windows 服务中将 ASP.NET 核心](#)

使用 Nginx 在 Linux 上托管 ASP.NET Core

2018/5/14 • 11 min to read • [Edit Online](#)

作者 : Sourabh Shirhatti

本指南介绍如何在 Ubuntu 16.04 服务器上设置生产就绪 ASP.NET Core 环境。

注意

对于 Ubuntu 14.04 *supervisord* 建议为用于监视 Kestrel 进程的解决方案。*systemd* 在 Ubuntu 14.04 上不可用。请参阅[本文档的以前版本](#)。

本指南：

- 将现有 ASP.NET Core 应用程序反向代理服务器后面。
- 将设置反向代理服务器将请求转发到 Kestrel web 服务器。
- 可确保在作为后台进程启动时运行的 web 应用。
- 配置有助于重新启动 web 应用的进程管理工具。

系统必备

1. 使用具有 sudo 特权的标准用户帐户访问 Ubuntu 16.04 服务器
2. 现有的 ASP.NET Core 应用程序

通过应用程序复制

运行 `dotnet publish` 从要打包到一个自包含的目录，可以在服务器上运行的应用程序的开发环境。

将 ASP.NET Core 应用程序复制到使用任何工具集成到组织的工作流（例如，SCP，FTP）服务器。测试应用，例如：

- 从命令行中，运行 `dotnet <app_assembly>.dll`。
- 在浏览器中，导航到 `http://<serveraddress>:<port>` 以确认应用在 Linux 上正常运行。

配置反向代理服务器

反向代理是为动态 web 应用程序提供服务的常见设置。反向代理终止 HTTP 请求，并将其转发到 ASP.NET 核心应用程序。

为何使用反向代理服务器？

Kestrel 非常适合从 ASP.NET Core 提供动态内容。但是，web 服务功能不是为 IIS、Apache 或 Nginx 例如与服务器的功能丰富。反向代理服务器可以卸载例如提供静态内容、缓存请求、压缩请求和从 HTTP 服务器的 SSL 终止的工作。反向代理服务器可能驻留在专用计算机上，也可能与 HTTP 服务器一起部署。

鉴于此指南的目的，使用 Nginx 的单个实例。它与 HTTP 服务器一起运行在同一服务器上。根据要求，可以选择不同的安装。

因为通过反向代理转发请求，使用从转发标头 Middleware `Microsoft.AspNetCore.HttpOverrides` 包。中间件更新 `Request.Scheme`，使用 `x-Forwarded-Proto` 标头，因此该重定向 Uri 和其他安全策略正常工作。

当使用任何类型的身份验证中间件，转发标头中间件必须运行第一个。此顺序可确保身份验证中间件可使用的标头值并生成正确的重定向 Uri。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

调用 `UseForwardedHeaders` 中的方法 `Startup.Configure` 之前调用 `UseAuthentication` 或类似的身份验证方案中间件。配置为转发的中间件 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

如果没有 `ForwardedHeadersOptions` 指定到中间件，要转发的默认标头是 `None`。

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。有关详细信息，请参阅[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

安装 Nginx

```
sudo apt-get install nginx
```

注意

如果将安装可选 Nginx 模块，生成从源 Nginx 可能需要。

使用 `apt-get` 安装 Nginx。安装程序创建一个 System V init 脚本，该脚本运行 Nginx 作为系统启动时的守护程序。因为是首次安装 Nginx，通过运行以下命令显式启动：

```
sudo service nginx start
```

确认浏览器显示 Nginx 的默认登陆页。

配置 Nginx

若要将 Nginx 配置为转发请求向 ASP.NET Core 应用程序的反向代理，修改 `/etc/nginx/sites-available/default`。在文本编辑器中打开它，并将内容替换为以下内容：

```
server {
    listen      80;
    server_name example.com *.example.com;
    location / {
        proxy_pass          http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header    Upgrade $http_upgrade;
        proxy_set_header    Connection keep-alive;
        proxy_set_header    Host $http_host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

如果没有 `server_name` Nginx 的匹配项，使用默认服务器。如果定义了默认的服务器，配置文件中的第一个服务器将是默认服务器。最佳做法，将添加在配置文件中返回 444 状态代码的特定的默认服务器。默认服务器配置示例是：

```
server {
    listen 80 default_server;
    # listen [::]:80 default_server deferred;
    return 444;
}
```

与上述配置文件和默认服务器, Nginx 接受主机标头使用的端口 80 上的公共流量 `example.com` 或 `*.example.com`。不会获取与这些主机不匹配的请求转发到 Kestrel。Nginx 将匹配的请求转发到在 Kestrel `http://localhost:5000`。请参阅[nginx 如何处理请求](#)有关详细信息。

警告

如果未能指定合适`server_name` 指令公开您的应用程序安全漏洞。子域通配符绑定 (例如, `*.example.com`) 不会带来安全风险, 若要控制整个父域 (相对于 `.com`, 这是易受攻击)。有关详细信息, 请参阅 [rfc7230 第 5.4 条](#)。

一旦建立 Nginx 配置, 运行 `sudo nginx -t` 若要验证的配置文件的语法。如果配置文件测试成功, 强制 Nginx 以便通过运行选取更改 `sudo nginx -s reload`。

监视应用程序

服务器已设置为转发到发出的请求 `http://<serveraddress>:80` Kestrel 在上运行 ASP.NET Core 应用到 `http://127.0.0.1:5000`。但是, Nginx 未设置来管理 Kestrel 进程。`systemd` 可以用于创建服务文件以启动和监视基础的 web 应用。`systemd` 是一个 init 系统, 可以提供用于启动、停止和管理进程的许多强大的功能。

创建服务文件

创建服务定义文件:

```
sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

下面是应用程序的示例服务文件:

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/aspnetcore/hellomvc
ExecStart=/usr/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
Restart=always
RestartSec=10 # Restart service after 10 seconds if dotnet service crashes
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

[Install]
WantedBy=multi-user.target
```

注意: 如果用户 `www` 数据未使用的配置, 此处定义的用户必须创建第一次并且为文件提供的适当的所有权。**注意:** Linux 具有区分大小写的文件系统。搜索配置文件中的“生产”结果设置 `ASPNETCORE_ENVIRONMENT appsettings.json`, 而不 `appsettings.production.json`。

注意

必须为要读取环境变量的配置提供程序转义某些值（例如，SQL 连接字符串）。使用以下命令以生成在配置文件中的正确转义的值以供使用：

```
systemd-escape "<value-to-escape>"
```

保存该文件并启用该服务。

```
sudo systemctl enable kestrel-hellomvc.service
```

启动服务并验证它正在运行。

```
sudo systemctl start kestrel-hellomvc.service
sudo systemctl status kestrel-hellomvc.service

● kestrel-hellomvc.service - Example .NET Web API App running on Ubuntu
   Loaded: loaded (/etc/systemd/system/kestrel-hellomvc.service; enabled)
   Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
     Main PID: 9021 (dotnet)
        CGroup: /system.slice/kestrel-hellomvc.service
                  └─9021 /usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
```

配置反向代理和管理通过 systemd Kestrel，web 应用完全配置，并且可以从处的本地计算机上的浏览器访问 `http://localhost`。它也是可从远程计算机，禁止任何防火墙可能阻止访问。检查响应标头，`Server` 标头将显示 ASP.NET Core 应用程序提供的 Kestrel。

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

查看日志

因为 web 应用使用 Kestrel 管理使用 `systemd`，到集中式日志记录所有事件和进程。但是，此日志包含由 `systemd` 管理的所有服务和进程的全部条目。若要查看特定于 `kestrel-hellomvc.service` 的项，请使用以下命令：

```
sudo journalctl -fu kestrel-hellomvc.service
```

有关进一步筛选，使用时间选项（如 `--since today`、`--until 1 hour ago`）或这些选项的组合可以减少返回的条目数。

```
sudo journalctl -fu kestrel-hellomvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

保护应用程序

启用 AppArmor

Linux 安全模块 (LSM) 是一个框架，是从 Linux 2.6 Linux 内核的一部分。LSM 支持安全模块的不同实现。AppArmor 是实现强制访问控制系统的 LSM，它允许将程序限制在一组有限的资源内。确保已启用并成功配置

AppArmor。

配置防火墙

关闭所有未使用的外部端口。通过为配置防火墙提供命令行接口，不复杂的防火墙 (ufw) 为 `iptables` 提供了前端。验证 `ufw` 配置为允许所需的任何端口上的流量。

```
sudo apt-get install ufw  
sudo ufw enable  
  
sudo ufw allow 80/tcp  
sudo ufw allow 443/tcp
```

保护 Nginx

Nginx 的默认分配不启用 SSL。若要启用其他安全功能，请从源生成。

下载源并安装生成依赖项

```
# Install the build dependencies  
sudo apt-get update  
sudo apt-get install build-essential zlib1g-dev libpcre3-dev libssl-dev libxml2-dev libgd2-xpm-dev libgeoip-dev libgoogle-perf-tools-dev libperl-dev  
  
# Download Nginx 1.10.0 or latest  
wget http://www.nginx.org/download/nginx-1.10.0.tar.gz  
tar zxf nginx-1.10.0.tar.gz
```

更改 Nginx 响应名称

编辑 `src/http/ngx_http_header_filter_module.c`:

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;  
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

配置选项和生成

正则表达式需要 PCRE 库。正则表达式用于 `ngx_http_rewrite_module` 的位置指令。`http_ssl_module` adds HTTPS 协议支持。

请考虑使用类似的 web 应用程序防火墙 *ModSecurity* 来加强应用程序。

```
./configure  
--with-pcre=../pcre-8.38  
--with-zlib=../zlib-1.2.8  
--with-http_ssl_module  
--with-stream  
--with-mail=dynamic
```

配置 SSL

- 配置服务器以侦听端口上的 HTTPS 流量 `443` 通过指定由受信任证书颁发机构 (CA) 颁发的有效证书。
- 通过采用的一些做法在下面的示例所示加强安全 `/etc/nginx/nginx.conf` 文件。示例包括选择更强的密码并将通过 HTTP 的所有流量重定向到 HTTPS。
- 添加 `HTTP Strict-Transport-Security` (HSTS) 标头可确保由客户端发起的所有后续请求都仅通过 HTTPS。
- 不添加 Strict 传输安全标头或选择适当 `max-age` 如果将在将来禁用 SSL。

添加 `/etc/nginx/proxy.conf` 配置文件:

```
proxy_redirect      off;
proxy_set_header   Host      $host;
proxy_set_header   X-Real-IP   $remote_addr;
proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header   X-Forwarded-Proto $scheme;
client_max_body_size 10m;
client_body_buffer_size 128k;
proxy_connect_timeout 90;
proxy_send_timeout 90;
proxy_read_timeout 90;
proxy_buffers 32 4k;
```

编辑 /etc/nginx/nginx.conf 配置文件。示例包含一个配置文件中的 `http` 和 `server` 部分。

```
http {
    include      /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens off;

    sendfile on;
    keepalive_timeout 29; # Adjust to the lowest possible value that makes sense for your use case.
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream hellomvc{
        server localhost:5000;
    }

    server {
        listen *:80;
        add_header Strict-Transport-Security max-age=15768000;
        return 301 https://$host$request_uri;
    }

    server {
        listen *:443      ssl;
        server_name     example.com;
        ssl_certificate /etc/ssl/certs/testCert.crt;
        ssl_certificate_key /etc/ssl/certs/testCert.key;
        ssl_protocols TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;
        ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
        ssl_ecdh_curve secp384r1;
        ssl_session_cache shared:SSL:10m;
        ssl_session_tickets off;
        ssl_stapling on; #ensure your cert is capable
        ssl_stapling_verify on; #ensure your cert is capable

        add_header Strict-Transport-Security "max-age=63072000; includeSubdomains; preload";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass  http://hellomvc;
            limit_req  zone=one burst=10 nodelay;
        }
    }
}
```

保护 Nginx 免受点击劫持的侵害

点击劫持是收集受感染用户的点击数的恶意技术。点击劫持诱使受害者(访问者)点击受感染的网站。使用 X-框架的选项以确保网站的安全。

编辑 nginx.conf 文件：

```
sudo nano /etc/nginx/nginx.conf
```

添加行 `add_header X-Frame-Options "SAMEORIGIN";` 并保存文件，然后重新启动 Nginx。

MIME 类型探查

此标头可阻止大部分浏览器通过 MIME 方式探查来自已声明内容类型的响应，因为标头会指示浏览器不要替代响应内容类型。使用 `nosniff` 选项后，如果服务器认为内容是“文本/html”，则浏览器将其显示为“文本/html”。

编辑 nginx.conf 文件：

```
sudo nano /etc/nginx/nginx.conf
```

添加行 `add_header X-Content-Type-Options "nosniff";` 并保存文件，然后重新启动 Nginx。

使用 Apache 在 Linux 上托管 ASP.NET Core

2018/5/14 • 11 min to read • [Edit Online](#)

作者 : Shayne Boyer

使用本指南中，了解如何设置Apache为反向代理服务器上CentOS 7将 HTTP 流量重定向到 ASP.NET 核心 web 应用程序上运行Kestrel。Mod_proxy 扩展和相关的模块创建服务器的反向代理。

系统必备

1. 具有 sudo 特权的标准用户帐户运行 CentOS 7 服务器
2. ASP.NET Core 应用

发布应用

发布应用程序作为独立的部署CentOS 7 运行时的版本配置中 (centos.7-x64)。内容复制bin/Release/netcoreapp2.0/centos.7-x64/publish使用 SCP、FTP 或其他文件传输方法向服务器的文件夹。

注意

在生产部署场景，持续集成工作流执行的工作的发布应用程序和复制到服务器的资产。

配置代理服务器

反向代理是为动态 web 应用程序提供服务的常见设置。反向代理终止 HTTP 请求，并将其转发到 ASP.NET 应用程序。

代理服务器，则其中一个客户端将请求转发到另一个服务器而不是本身满足请求。反向代理转发到固定的目标，通常代表任意客户端。在本指南中，Apache 正在配置为在同一服务器上运行 Kestrel 为提供 ASP.NET Core 应用程序服务的反向代理。

因为通过反向代理转发请求，使用从转发标头 Middleware Microsoft.AspNetCore.HttpOverrides包。中间件更新Request.Scheme，使用X-Forwarded-Proto 标头，因此该重定向 Uri 和其他安全策略正常工作。

当使用任何类型的身份验证中间件，转发标头中间件必须运行第一个。此顺序可确保身份验证中间件可使用的标头值并生成正确的重定向 Uri。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

调用UseForwardedHeaders中的方法 Startup.Configure 之前调用UseAuthentication或类似的身份验证方案中间件。配置为转发的中间件 X-Forwarded-For 和 X-Forwarded-Proto 标头：

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

如果没有ForwardedHeadersOptions指定到中间件，要转发的默认标头是None。

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。有关详细信息，请参阅[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)。

安装 Apache

更新 CentOS 程序包添加到其最新稳定版本：

```
sudo yum update -y
```

使用单个在 CentOS 上安装 Apache web 服务器 `yum` 命令：

```
sudo yum -y install httpd mod_ssl
```

运行该命令后输出的示例：

```
Downloading packages:
httpd-2.4.6-40.el7.centos.4.x86_64.rpm | 2.7 MB  00:00:01
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Installing : httpd-2.4.6-40.el7.centos.4.x86_64      1/1
Verifying   : httpd-2.4.6-40.el7.centos.4.x86_64      1/1

Installed:
httpd.x86_64 0:2.4.6-40.el7.centos.4

Complete!
```

注意

在此示例中，输出将反映 `httpd.86_64`，由于 CentOS 7 版本是 64 位。若要验证 Apache 的安装位置，请从命令提示符运行 `whereis httpd`。

配置 Apache 用于反向代理

Apache 的配置文件位于 `/etc/httpd/conf.d/` 目录内。具有的所有文件 `.conf` 扩展处理除了中的模块配置文件按字母顺序 `/etc/httpd/conf.modules.d/`，其中包含的任何配置所需加载模块文件。

创建一个配置文件，名为 `hellomvc.conf`，应用程序：

```
<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ServerName www.example.com
    ServerAlias *.example.com
    ErrorLog ${APACHE_LOG_DIR}hellomvc-error.log
    CustomLog ${APACHE_LOG_DIR}hellomvc-access.log common
</VirtualHost>
```

`VirtualHost` 块可以出现多次，在服务器上的一个或多个文件。在前面的配置文件中，Apache 接受公共端口 80 上的流量。域 `www.example.com` 正在提供服务，与 `*.example.com` 别名解析为同一网站。请参阅[基于名称的虚拟主机支持](#)有关详细信息。请求是服务器的代理针对端口 5000 上 127.0.0.1 的根目录。对于双向通信，`ProxyPass` 和 `ProxyPassReverse` 所需。

警告

如果未能指定合适 `ServerName` 指令中 `VirtualHost` 块公开您的应用程序安全漏洞。子域通配符绑定 (例如, `*.example.com`) 不会带来安全风险, 若要控制整个父域 (相对于 `*.com`, 这是易受攻击)。有关详细信息, 请参阅 [rfc7230 第 5.4 条](#)。

可以每个配置日志记录 `VirtualHost` 使用 `ErrorLog` 和 `CustomLog` 指令。`ErrorLog` 是服务器用来记录错误的位置和 `CustomLog` 设置的文件名和日志文件格式。在这种情况下, 这是记录请求信息的位置。没有为每个请求的一行。

将文件保存与测试配置。如果一切正常, 响应应为 `Syntax [OK]`。

```
sudo service httpd configtest
```

重新启动 Apache:

```
sudo systemctl restart httpd
sudo systemctl enable httpd
```

监视应用程序

Apache 现在已设置为转发到发出的请求 `http://localhost:80` Kestrel 在上运行 ASP.NET Core 应用到 `http://127.0.0.1:5000`。但是, Apache 未设置来管理 Kestrel 进程。使用 `systemd` 和创建服务文件以启动和监视基础的 web 应用。`systemd` 是一个 init 系统, 可以提供用于启动、停止和管理进程的许多强大的功能。

创建服务文件

创建服务定义文件:

```
sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

应用程序示例服务文件:

```
[Unit]
Description=Example .NET Web API App running on CentOS 7

[Service]
WorkingDirectory=/var/aspnetcore/hellomvc
ExecStart=/usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
Restart=always
# Restart service after 10 seconds if dotnet service crashes
RestartSec=10
SyslogIdentifier=dotnet-example
User=apache
Environment=ASPNETCORE_ENVIRONMENT=Production

[Install]
WantedBy=multi-user.target
```

注意

用户—如果用户 `apache` 未使用的配置, 用户必须创建第一次并且为文件提供的适当的所有权。

注意

必须为要读取环境变量的配置提供程序转义某些值（例如，SQL 连接字符串）。使用以下命令以生成在配置文件中的正确转义的值以供使用：

```
systemd-escape "<value-to-escape>"
```

保存该文件并启用该服务：

```
sudo systemctl enable kestrel-hellomvc.service
```

启动服务，然后验证它正在运行：

```
sudo systemctl start kestrel-hellomvc.service
sudo systemctl status kestrel-hellomvc.service

● kestrel-hellomvc.service - Example .NET Web API App running on CentOS 7
   Loaded: loaded (/etc/systemd/system/kestrel-hellomvc.service; enabled)
   Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
     Main PID: 9021 (dotnet)
       CGroup: /system.slice/kestrel-hellomvc.service
               └─9021 /usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
```

使用反向代理配置和通过管理的 Kestrel *systemd*, web 应用进行了完全配置，并且可以从处的本地计算机上的浏览器访问 `http://localhost`。检查响应标头，**服务器**标头指示 ASP.NET Core 应用由 Kestrel：

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

查看日志

因为 web 应用使用 Kestrel 管理使用 *systemd*, 到集中式日志记录事件和进程。但是，此日志包含的服务和由托管的进程的所有条目 *systemd*。若要查看特定于 `kestrel-hellomvc.service` 的项，请使用以下命令：

```
sudo journalctl -fu kestrel-hellomvc.service
```

时间筛选，使用命令中指定时间选项。例如，使用 `--since today` 来筛选出存在当天或 `--until 1 hour ago` 来查看前一小时的条目。有关详细信息，请参阅[journalctl 手册页](#)。

```
sudo journalctl -fu kestrel-hellomvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

保护应用程序

配置防火墙

*Firewalld*是动态的守护程序，来管理具有对网络区域支持的防火墙。端口和数据包筛选仍可通过 *iptables* 管理。*Firewalld*默认情况下应安装。`yum` 可以用于安装包或验证已安装。

```
sudo yum install firewalld -y
```

使用 `firewalld` 以打开仅为应用程序所需的端口。在此示例中，使用的是端口 80 和 443。以下命令永久设置端口 80 和 443 以打开：

```
sudo firewall-cmd --add-port=80/tcp --permanent  
sudo firewall-cmd --add-port=443/tcp --permanent
```

重新加载防火墙设置。检查可用的服务和默认区域中的端口。选项均可通过检查 `firewall-cmd -h`。

```
sudo firewall-cmd --reload  
sudo firewall-cmd --list-all
```

```
public (default, active)  
interfaces: eth0  
sources:  
services: dhcpcv6-client  
ports: 443/tcp 80/tcp  
masquerade: no  
forward-ports:  
icmp-blocks:  
rich rules:
```

SSL 配置

若要配置 SSL，Apache *如果* 使用模块。当 `httpd` 模块已安装，*如果* 同时安装模块。如果它未安装，请使用 `yum` 以将其添加到配置。

```
sudo yum install mod_ssl
```

若要强制实施 SSL，请安装 `mod_rewrite` 模块以启用 URL 重写：

```
sudo yum install mod_rewrite
```

修改 `hellomvc.conf` 文件启用 URL 重写且安全的端口 443 上的通信：

```
<VirtualHost *:80>  
    RewriteEngine On  
    RewriteCond %{HTTPS} !=on  
    RewriteRule ^/?(.*) https://{$SERVER_NAME}/ [R,L]  
</VirtualHost>  
  
<VirtualHost *:443>  
    ProxyPreserveHost On  
    ProxyPass / http://127.0.0.1:5000/  
    ProxyPassReverse / http://127.0.0.1:5000/  
    ErrorLog /var/log/httpd/hellomvc-error.log  
    CustomLog /var/log/httpd/hellomvc-access.log common  
    SSLEngine on  
    SSLProtocol all -SSLv2  
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4  
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt  
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key  
</VirtualHost>
```

注意

此示例中使用的本地生成的证书。**SSLCertificateFile**应为域名称的主证书文件。**SSLCertificateKeyFile**密钥文件时应生成创建 CSR。**SSLCertificateChainFile**应中间证书文件（如果有），由证书颁发机构提供的。

保存该文件并测试配置：

```
sudo service httpd configtest
```

重新启动 Apache：

```
sudo systemctl restart httpd
```

其他 Apache 建议

其他标头

为了保护免受恶意攻击，有几个标头应被修改或添加。确保 `mod_headers` 安装模块：

```
sudo yum install mod_headers
```

从 clickjacking 攻击的安全 Apache

[Clickjacking](#)，也称为 [UI 改变现有攻击](#)，是一种恶意攻击其中诱骗网站访问者不是它们目前正在访问单击的链接或另一页上的按钮。使用 `X-FRAME-OPTIONS` 以确保网站的安全。

编辑 `httpd.conf` 文件：

```
sudo nano /etc/httpd/conf/httpd.conf
```

将行添加 `Header append X-FRAME-OPTIONS "SAMEORIGIN"`。保存该文件。重启 Apache。

MIME 类型探查

`X-Content-Type-Options` 标头会阻止从 Internet Explorer [MIME 探查](#)(确定文件的 `Content-Type` 从该文件的内容)。如果服务器设置 `Content-Type` 标头到 `text/html` 与 `nosniff` 选项集，Internet Explorer 呈现作为内容 `text/html` 不考虑文件的内容。

编辑 `httpd.conf` 文件：

```
sudo nano /etc/httpd/conf/httpd.conf
```

将行添加 `Header set X-Content-Type-Options "nosniff"`。保存该文件。重启 Apache。

负载平衡

此示例演示如何在同一实例计算机上的 CentOS 7 和 Kestrel 上设置和配置 Apache。为了不具有单点故障;使用 `mod_proxy_balancer` 和修改 `VirtualHost` 将允许用于管理 Apache 代理服务器后面的 web 应用的多个实例。

```
sudo yum install mod_proxy_balancer
```

在配置文件中下面所示的其他实例 `hellomvc` 应用已设置为在端口 5001 上运行。代理部分设置与平衡器配置具有两个成员进行负载平衡 `byrequests`。

```

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https:// %{SERVER_NAME}/ [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPass / balancer://mycluster/
    ProxyPassReverse / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5001/

    <Proxy balancer://mycluster>
        BalancerMember http://127.0.0.1:5000
        BalancerMember http://127.0.0.1:5001
        ProxySet lbmethod=byrequests
    </Proxy>

    <Location />
        SetHandler balancer
    </Location>
    ErrorLog /var/log/httpd/helломvc-error.log
    CustomLog /var/log/httpd/helломvc-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>

```

速率限制

使用`mod_ratelimit`中, 附带`httpd`模块, 客户端的带宽可将限制:

```
sudo nano /etc/httpd/conf.d/ratelimit.conf
```

示例文件为 600 KB/秒根位置下限制带宽:

```

<IfModule mod_ratelimit.c>
    <Location />
        SetOutputFilter RATE_LIMIT
        SetEnv rate-limit 600
    </Location>
</IfModule>

```

在 Docker 容器中托管 ASP.NET Core

2018/4/10 • 1 min to read • [Edit Online](#)

下面的文章可用于了解如何在 Docker 中托管 ASP.NET Core 应用：

[容器和 Docker 简介](#)

容器化是软件开发的一种方法，通过该方法可将应用程序或服务、其依赖项及其配置一起打包为容器映像。了解相关内容。可对该映像进行测试，然后将其部署到主机。

[什么是 Docker](#)

了解如何将 Docker 作为一种开源项目，用于将应用自动部署为可在云或本地运行的便携式独立容器。

[Docker 术语](#)

了解 Docker 技术的术语和定义。

[Docker 容器、映像和注册表](#)

了解如何将 Docker 容器映像存储在映像注册表中，以实现跨环境的一致部署。

[为 .NET Core 应用程序生成 Docker 映像](#)

了解如何生成和 Docker 化 ASP.NET Core 应用。了解由 Microsoft 维护的 Docker 映像并检查用例。

[Visual Studio Tools for Docker](#)

Visual Studio 2017 支持在用于 Windows 的 Docker 上生成、调试和运行面向 .NET Framework 或 .NET Core 的 ASP.NET Core 应用。了解相关内容。Windows 和 Linux 容器均受支持。

[发布到 Docker 映像](#)

了解如何通过 Visual Studio Tools for Docker 扩展使用 PowerShell 将 ASP.NET Core 应用部署到 Azure 上的 Docker 主机。

[配置 ASP.NET Core 以使用代理服务器和负载均衡器](#)

对于托管在代理服务器和负载均衡器后方的应用，可能需要附加配置。通过代理传递的请求通常会遮盖初始请求相关信息，例如方案和客户端 IP。可能必须将请求相关的一些信息手动转发给应用。

使用 ASP.NET Core 的 Visual Studio Tools for Docker

2018/3/12 • 7 min to read • [Edit Online](#)

Visual Studio 2017 支持生成、调试和运行面向.NET Core 应用的容器化的 ASP.NET Core。Windows 和 Linux 容器均受支持。

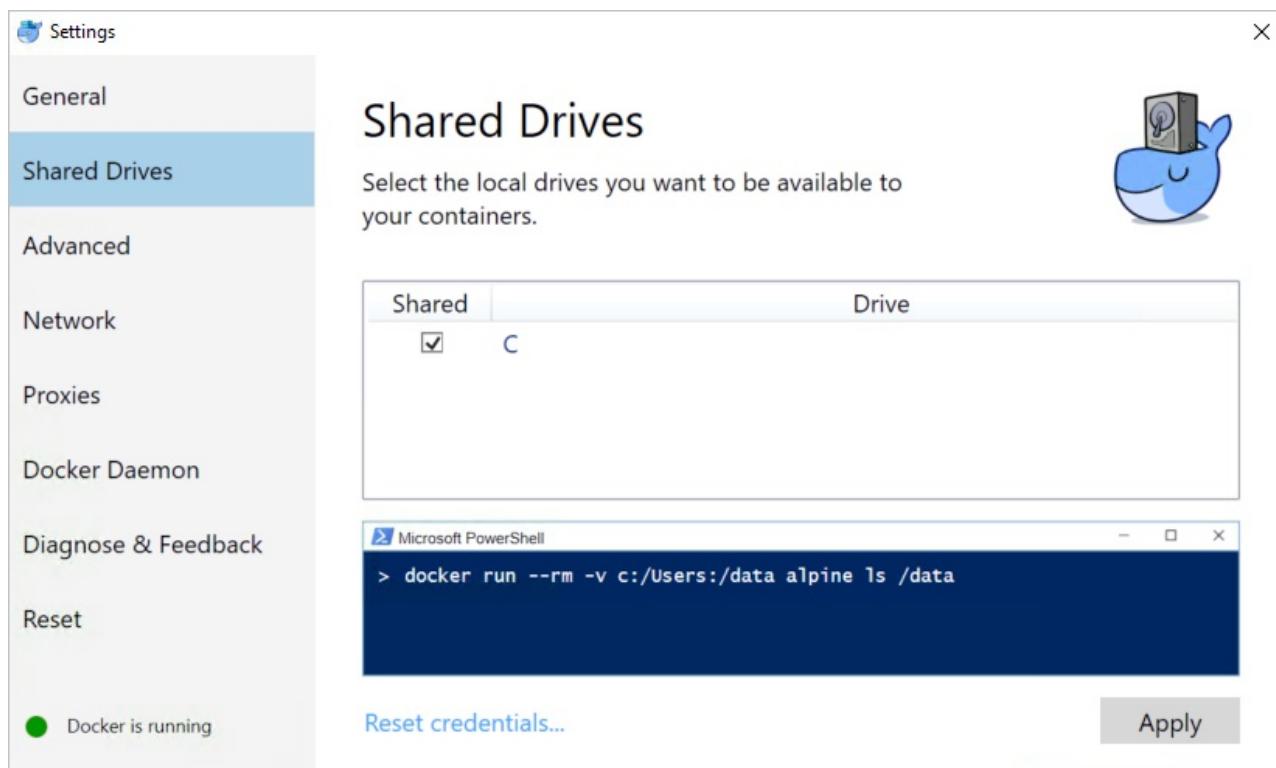
系统必备

- 具有“.NET Core 跨平台开发”工作负载的 [Visual Studio 2017](#)
- [Docker for Windows](#)

安装和设置

要安装 Docker，请查看 [Docker for Windows: 安装须知](#)了解相关信息，并安装 Docker for Windows。

Docker for Windows 中的[共享驱动器](#)必须配置为支持卷映射和调试。右键单击系统托盘 Docker 图标，选择[设置...](#)，然后选择[共享驱动器](#)。选择 Docker 其中存储文件的驱动器。选择[应用](#)。



提示

Visual Studio 2017 版本 15.6 及更高版本时提示[共享驱动器未配置](#)。

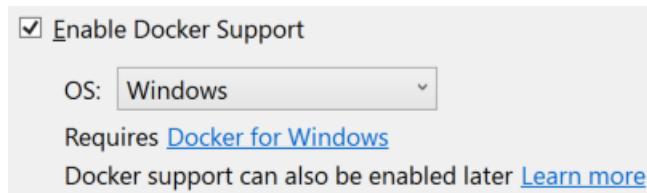
向应用添加 Docker 支持

若要将 Docker 支持添加到 ASP.NET 核心项目，项目必须为目标.NET 核心。支持 Linux 和 Windows 容器。

当将 Docker 支持添加到项目中，选择 Windows 或 Linux 容器。Docker 主机必须运行类型相同的容器。要更正在运行的 Docker 实例中的容器类型，请右键单击系统托盘中的 Docker 图标，再选择“切换到 Windows 容器...”或“切换到 Linux 容器...”。

新应用

使用 ASP.NET Core Web 应用程序项目模板创建新应用时, 请选中“启用 Docker 支持”复选框:



如果目标框架是.NET 核心OS下拉列表允许针对选择的容器类型。

现有应用

Visual Studio Tools for Docker 不支持向面向.NET Framework 的现有 ASP.NET Core 项目添加 Docker。对于面向 .NET Core 的 ASP.NET Core 项目, 可通过两个选项使用工具添加 Docker 支持。在 Visual Studio 中打开项目, 然后选择以下选项之一:

- 从“项目”菜单中选择“Docker 支持”。
- 右键单击解决方案资源管理器中的项目, 然后选择“添加” > “Docker 支持”。

Docker 资产概述

Visual Studio Tools for Docker 向解决方案中添加 docker-compose 项目, 包括以下各项:

- .dockerignore: 包含文件和目录模式的列表, 排除生成上下文的时间。
- docker-compose.yml: 基本 [Docker Compose](#) 文件, 用于定义要分别通过 `docker-compose build` 和 `docker-compose run` 生成和运行的映像集合。
- docker compose.override.yml: 一个可选文件, 通过 Docker Compose 读取, 包含服务的配置替代。Visual Studio 执行 `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` 以合并这些文件。

Dockerfile, 用作创建最终 Docker 映像的方案, 添加到项目根目录。请参阅 [Dockerfile 引用](#), 了解其中的命令。此特定 Dockerfile 使用[多阶段生成](#), 该生成包含四个不同的命名生成阶段:

```
FROM microsoft/aspnetcore:2.0-nanoserver-1709 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0-nanoserver-1709 AS build
WORKDIR /src
COPY *.sln .
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore
COPY ..
WORKDIR /src/HelloDockerTools
RUN dotnet build -c Release -o /app

FROM build AS publish
RUN dotnet publish -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

Dockerfile 基于 [microsoft/aspnetcore](#) 映像。此基础映像包括 ASP.NET Core NuGet 包, 已对此包进行了预实时编译, 以提高启动性能。

*Docker-compose.yml*文件包含的项目运行时创建的映像的名称:

```

version: '3'

services:
  hellodockertools:
    image: hellodockertools
    build:
      context: .
    dockerfile: HelloDockerTools\ Dockerfile

```

在前面的示例中，应用在“调试”模式下运行时，`image: hellodockertools` 生成映像 `hellodockertools:dev`。应用在“发布”模式下运行时，生成 `hellodockertools:latest` 映像。

映像名称加上前缀 Docker Hub 用户名（例如，`dockerhubusername/hellodockertools`）如果该图像将推送到注册表。或者，更改要包括私有注册表 URL 的映像名称（例如，`privateregistry.domain.com/hellodockertools`）根据配置。

调试

在工具栏的调试下拉列表中选择“Docker”，然后开始调试应用。“输出”窗口的“Docker”视图显示发生的以下操作：

- *Microsoft/aspnetcore*（如果尚未在缓存中）获取运行时映像。
- *Microsoft/aspnetcore-build* 生成 / 发布编译的映像（如果尚未在缓存中）获取。
- ASPNETCORE_ENVIRONMENT 环境变量设置为位于 `Development` 容器内。
- 已公开端口 80，并已映射为 localhost 动态分配的端口。该端口由 Docker 主机确定，并且可以使用 `docker ps` 进行查询。
- 应用程序复制到容器。
- 使用调试程序附加到使用动态分配端口容器启动默认浏览器。

生成的 Docker 映像是开发人员使用应用程序的映像 `microsoft/aspnetcore` 作为基本映像的映像。在“包管理器控制台”(PMC) 窗口中运行 `docker images` 命令。显示计算机上的映像：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hellodockertools	latest	f8f9d6c923e2	About an hour ago	391MB
hellodockertools	dev	85c5ffee5258	About an hour ago	389MB
microsoft/aspnetcore-build	2.0-nanoserver-1709	d7cce94e3eb0	15 hours ago	1.86GB
microsoft/aspnetcore	2.0-nanoserver-1709	8872347d7e5d	40 hours ago	389MB

注意

开发人员映像作为缺少的应用程序内容，调试配置使用卷装载提供迭代的体验。要推送映像，请使用“发布”配置。

在 PMC 中运行 `docker ps` 命令。请注意，应用使用容器运行：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
baf9a678c88d	hellodockertools:dev	"C:\\remote_debugge..."	21 seconds ago	Up 19 seconds

编辑并继续

对静态文件和 Razor 视图的更改会自动更新，无需执行编译步骤。进行更改，保存并在浏览器中刷新，以查看更新。

对代码文件进行修改需要在容器内进行编译和重启 Kestrel。更改后，使用 CTRL+F5 执行该过程并在容器内启动应用。Docker 容器不重新生成或停止。在 PMC 中运行 `docker ps` 命令。请注意，截至 10 分钟前，原始容器仍在运

行：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
baf9a678c88d	hellodockertools:dev	"C:\\\\remote_debugge..."	10 minutes ago	Up 10 minutes
0.0.0.0:37630->80/tcp	dockercompose4642749010770307127_hellodockertools_1			

发布 Docker 映像

应用程序的开发和调试周期完成后，Visual Studio Tools for Docker 帮助创建应用程序的生产映像。将配置下拉列表更改为“发布”，然后生成应用。此工具生成的映像最新标记，可以推送到的私有注册表或 Docker Hub。

在 PMC 中运行 `docker images` 命令，查看映像列表：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hellodockertools	latest	4cb1fca533f0	19 seconds ago	391MB
hellodockertools	dev	85c5ffee5258	About an hour ago	389MB
microsoft/aspnetcore-build	2.0-nanoserver-1709	d7cce94e3eb0	16 hours ago	1.86GB
microsoft/aspnetcore	2.0-nanoserver-1709	8872347d7e5d	40 hours ago	389MB

注意

`docker images` 命令返回存储库名称和标记标识为 `<none>` (上面未列出) 的中间映像。这些未命名映像由[多阶段生成 Dockerfile](#) 生成。它们可提高生成最终映像的效率 — 发生更改时，仅重新生成必要的层。当不再需要中间的图像时，则删除它们使用 `docker rmi` 命令。

可能希望生产或发布映像的大小比开发映像小。由于卷映射中，调试器和应用程序已运行从本地计算机和不在容器内。最新映像已打包必要的应用代码，以在主机上运行应用。因此，增量是应用程序代码的大小。

配置 ASP.NET 核心以使用代理服务器和负载平衡器

2018/5/4 • 11 min to read • [Edit Online](#)

通过[Luke Latham](#)和[Chris 跨](#)

对于 ASP.NET Core 推荐的配置，在使用 IIS/ASP.NET 核心模块、Nginx 或 Apache 承载应用程序。代理服务器、负载平衡器和其他网络设备中通常在它到达应用程序之前遮盖有关请求的信息：

- 当通过 HTTP HTTPS 请求代理时，原始的方案 (HTTPS) 将丢失，并且必须转发标头中。
- 因为应用程序收到请求时从该代理并不是其源 true 在 Internet 或公司网络上的，还必须在头转发发起的客户端 IP 地址。

此信息可能会在中处理的请求，例如重定向、身份验证、链接生成、策略评估和客户端 geolocation 十分重要。

转发的标头

按照约定，代理将在 HTTP 头中的信息转发。

HEADER	描述
X-Forwarded-For	包含有关客户端启动的请求和后续代理的代理链中的信息。此参数可能包含 IP 地址（和（可选）端口号）。代理服务器链中的第一个参数指示客户端第一次发出请求。后续代理标识符遵循。链中的最后一个代理不在列表中的参数。最后一个代理服务器的 IP 地址和（可选）端口号时，都可用作在传输层的远程 IP 地址。
X-Forwarded-Proto	原始的方案 (HTTP/HTTPS) 的值。如果请求具有遍历多个代理服务器，则这也可能的方案的列表。
X-Forwarded-Host	主机标头字段的原始值。通常情况下，代理无需修改的主机标头。请参阅 Microsoft 安全公告 CVE-2018年-0787年 有关一个提升特权漏洞，它会影响的系统代理不验证其中或 restict 主机为已知良好的值的标头信息。

转发标头中间件，从[Microsoft.AspNetCore.HttpOverrides](#)打包、读取这些标头和关联的字段中填充上[HttpContext](#)。

中间件更新中：

- `HttpContext.Connection.RemoteIpAddress` – 使用设置 `X-Forwarded-For` 标头值。其他设置会影响该中间件的设置如何 `RemoteIpAddress`。有关详细信息，请参阅[转发标头中间件选项](#)。
- `HttpContext.Request.Scheme` – 使用设置 `X-Forwarded-Proto` 标头值。
- `HttpContext.Request.Host` – 使用设置 `X-Forwarded-Host` 标头值。

请注意，并非所有网络设备都添加 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头，而无需其他配置。如果代理的请求不包含这些标头，在达到应用程序时，请查阅设备制造商提供的指导。

转发标头中间件[默认设置](#)可以配置。默认设置如下：

- 只有一个代理应用程序和请求的源之间。
- 已知的代理的配置和已知网络仅环回地址。

IIS/IIS Express 和 ASP.NET Core 模块

IIS 和 ASP.NET 核心模块后面运行应用时，默认情况下，通过 IIS 集成中间件启用转发标头中间件。转发的标头中间件激活到 ASP.NET 核心模块由于信任问题与转发标头一起运行第一个与特定的受限配置中间件管道中（例如，[IP 欺骗](#)）。该中间件配置为转发 `X-Forwarded-For` 和 `X-Forwarded-Proto` 标头且被限制到单个 `localhost` 代理。如果不需要附加配置，请参阅[转发标头中间件选项](#)。

其他代理服务器和负载平衡器方案

除了使用 IIS 集成中间件，默认情况下不启用转发标头中间件。必须为转发过程标头包含应用程序启用转发标头中间件 `UseForwardedHeaders`。如果未启用该中间件后 `ForwardedHeadersOptions` 到中间件，默认值指定 `ForwardedHeadersOptions.ForwardedHeaders` 是 `ForwardedHeaders.None`。

配置与中间件 `ForwardedHeadersOptions` 转发 `X-Forwarded-For` 和 `X-Forwarded-Proto` 中的标头 `Startup.ConfigureServices`。调用 `UseForwardedHeaders` 中的方法 `Startup.Configure` 在调用其他中间件之前：

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

    services.Configure<ForwardedHeadersOptions>(options =>
    {
        options.ForwardedHeaders =
            ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseForwardedHeaders();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();
    // In ASP.NET Core 1.x, replace the following line with: app.UseIdentity();
    app.UseAuthentication();
    app.UseMvc();
}
```

注意

如果没有 `ForwardedHeadersOptions` 中指定 `Startup.ConfigureServices` 或直接向与该扩展方法 `UseForwardedHeaders` (`IApplicationBuilder`, `ForwardedHeadersOptions`)，默认值标头以转发 `ForwardedHeaders.None`。
`ForwardedHeadersOptions.ForwardedHeaders` 属性必须使用要转发的标头进行配置。

转发的标头中间件选项

`ForwardedHeadersOptions` 控制转发标头中间件的行为：

```

services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-Custom-Header-Name";
});

```

选项	描述
ForwardedForHeaderName	<p>使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XForwardedForHeaderName</code>。</p> <p>默认值为 <code>X-Forwarded-For</code>。</p>
ForwardedHeaders	<p>标识应处理的转发器。请参阅 ForwardedHeaders 枚举 有关应用的字段的列表。分配给此属性的典型值为 <code>ForwardedHeaders.XForwardedFor ForwardedHeaders.XForwardedProto</code></p> <ul style="list-style-type: none"> ◦ <p>默认值是 <code>ForwardedHeaders.None</code>。</p>
ForwardedHostHeaderName	<p>使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XForwardedHostHeaderName</code>。</p> <p>◦</p> <p>默认值为 <code>X-Forwarded-Host</code>。</p>
ForwardedProtoHeaderName	<p>使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XForwardedProtoHeaderName</code>。</p> <p>◦</p> <p>默认值为 <code>X-Forwarded-Proto</code>。</p>
ForwardLimit	<p>限制在处理标头中的项数。设置为 <code>null</code> 禁用了限制，但这只应该在 <code>KnownProxies</code> 或 <code>KnownNetworks</code> 配置。</p> <p>默认值为 1。</p>
KnownNetworks	<p>地址范围的已知的代理，以接受从转发标头。提供使用无类别域际路由选择 (CIDR) 表示法的 IP 范围。</p> <p>默认值是 <code>IList<ip 网络></code> 包含为单个条目 <code>IPAddress.Loopback</code>。</p>
KnownProxies	<p>已知的代理，以接受从转发标头的地址。使用 <code>KnownProxies</code> 以指定确切的 IP 地址匹配。</p> <p>默认值是 <code>IList<IPAddress></code> 包含为单个条目 <code>IPAddress.IPv6Loopback</code>。</p>
OriginalForHeaderName	<p>使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XOriginalForHeaderName</code>。</p> <p>默认值为 <code>X-Original-For</code>。</p>

选项	描述
OriginalHostHeaderName	使用而不是指定的此属性指定的标 头ForwardedHeadersDefaults.XOriginalHostHeaderName。 默认值为 <code>X-Original-Host</code> 。
OriginalProtoHeaderName	使用而不是指定的此属性指定的标 头ForwardedHeadersDefaults.XOriginalProtoHeaderName。 默认值为 <code>X-Original-Proto</code> 。
RequireHeaderSymmetry	要求的数量的标头的值进行之间同 步ForwardedHeadersOptions.ForwardedHeaders正在处理。 ASP.NET 核心 1.x 中的默认值 <code>true</code> 。默认值在 ASP.NET 核 心 2.0 或更高版本是 <code>false</code> 。

选项	描述
AllowedHosts	限制的主机 <code>X-Forwarded-Host</code> 到提供的值的标头。 <ul style="list-style-type: none">使用序号忽略 case 比较值。端口号，则必须排除它。如果列表为空，则允许所有主机。顶级通配符 <code>*</code> 允许所有非空主机。子域通配符允许使用，但不匹配的根域。例如，<code>*.contoso.com</code> 匹配子域 <code>foo.contoso.com</code> 但不是根域 <code>contoso.com</code>。允许使用 Unicode 主机名，但转换为 Punycode 匹配。IPv6 地址 必须包括边界方括号，而且必须在 传统的窗 体(例如，<code>[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]</code>)。IPv6 地址不是特殊情况，若要检查不同格式之间的逻辑相等性，并且在执行任何规范化。不限制允许的主机可能允许攻击者欺骗由服务生成的链接。 默认值为空 <code>IList<字符串 ></code> 。
ForwardedForHeaderName	使用而不是指定的此属性指定的标 头ForwardedHeadersDefaults.XForwardedForHeaderName。 默认值为 <code>X-Forwarded-For</code> 。
ForwardedHeaders	标识应处理的转发器。请参阅 ForwardedHeaders 枚举 有关 应用的字段的列表。分配给此属性的典型值为 <code>ForwardedHeaders.XForwardedFor </code> <code>ForwardedHeaders.XForwardedProto</code> 。 默认值是 <code>ForwardedHeaders.None</code> 。
ForwardedHostHeaderName	使用而不是指定的此属性指定的标 头ForwardedHeadersDefaults.XForwardedHostHeaderName 。 默认值为 <code>X-Forwarded-Host</code> 。

选项	描述
ForwardedProtoHeaderName	使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XForwardedProtoHeaderName</code> 。 默认值为 <code>X-Forwarded-Proto</code> 。
ForwardLimit	限制在处理标头中的项数。设置为 <code>null</code> 禁用了限制, 但这只应该在 <code>KnownProxies</code> 或 <code>KnownNetworks</code> 配置。 默认值为 1。
KnownNetworks	地址范围的已知的代理, 以接受从转发标头。提供使用无类别域际路由选择 (CIDR) 表示法的 IP 范围。 默认值是 <code>IList<ip 网络></code> 包含为单个条目 <code>IPAddress.Loopback</code> 。
KnownProxies	已知的代理, 以接受从转发标头的地址。使用 <code>KnownProxies</code> 以指定确切的 IP 地址匹配。 默认值是 <code>IList<IPAddress></code> 包含为单个条目 <code>IPAddress.IPv6Loopback</code> 。
OriginalForHeaderName	使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XOriginalForHeaderName</code> 。 默认值为 <code>X-Original-For</code> 。
OriginalHostHeaderName	使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XOriginalHostHeaderName</code> 。 默认值为 <code>X-Original-Host</code> 。
OriginalProtoHeaderName	使用而不是指定的此属性指定的标头 <code>ForwardedHeadersDefaults.XOriginalProtoHeaderName</code> 。 默认值为 <code>X-Original-Proto</code> 。
RequireHeaderSymmetry	要求的数量的标头的值进行之间同步 <code>ForwardedHeadersOptions.ForwardedHeaders</code> 正在处理。 ASP.NET 核心 1.x 是中的默认值 <code>true</code> 。默认值在 ASP.NET 核心 2.0 或更高版本是 <code>false</code> 。

方案和用例

当无法添加转发标头和所有请求并安全

在某些情况下, 它可能不能将转发的头添加到代理到应用程序的请求。如果代理强制实施所有的公共外部请求 HTTPS, 方案可以手动设置 `Startup.Configure` 在使用任何类型的中间件之前:

```
app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next();
});
```

此代码可以使用环境变量或在开发或过渡环境中的其他配置设置来禁用。

处理基路径和更改请求路径的代理

某些代理传递路径不变，但与应用，以便路由应删除的基路径可正常工作。

[UsePathBaseExtensions](#).[UsePathBase](#)中间件将拆分到路径[HttpRequest.Path](#)和到的应用程序基路径[HttpRequest.PathBase](#)。

如果`/foo`作为传递代理路径是应用程序基路径`/foo/api/1`，中间件集`Request.PathBase`到`/foo`和`Request.Path`到`/api/1`使用以下命令：

```
app.UsePathBase("/foo");
```

原始路径和基路径都将重新应用时按相反的顺序再次调用该中间件。中间件订单处理的详细信息，请参阅[中间件](#)。

如果代理修剪路径（例如，转发`/foo/api/1`到`/api/1`），修补程序将重定向，并通过设置请求的链接[PathBase](#)属性：

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next();
});
```

如果代理添加路径数据，放弃路径重定向和链接使用的进行修复的一部分[StartsWithSegments](#)（[PathString](#), [PathString](#)）并将分配给[路径](#)属性：

```
app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next();
});
```

疑难解答

标头不转发按预期方式，当启用[日志记录](#)。如果日志未提供足够的信息来解决该问题，枚举服务器收到的请求标头。可以对应用程序响应使用内联中间件编写标头：

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/plain";

        // Request method, scheme, and path
        await context.Response.WriteAsync(
            $"Request Method: {context.Request.Method}{Environment.NewLine}");
        await context.Response.WriteAsync(
            $"Request Scheme: {context.Request.Scheme}{Environment.NewLine}");
        await context.Response.WriteAsync(
            $"Request Path: {context.Request.Path}{Environment.NewLine}");

        // Headers
        await context.Response.WriteAsync($"Request Headers:{Environment.NewLine}");

        foreach (var header in context.Request.Headers)
        {
            await context.Response.WriteAsync($"{header.Key}: " +
                $"{header.Value}{Environment.NewLine}");
        }

        await context.Response.WriteAsync(Environment.NewLine);

        // Connection: RemoteIp
        await context.Response.WriteAsync(
            $"Request RemoteIp: {context.Connection.RemoteIpAddress}");
    });
}
```

确保转发-X * 预期值与服务器收到标头。如果给定的标头中有多个值，请注意按相反的顺序从右到左的转发标头中间件进程标头。

请求的原始远程 IP 必须匹配中的条目 KnownProxies 或 KnownNetworks 列出 X-转发-对于处理之前。这就限制了标头欺骗不接受来自不受信任代理转发器。

其他资源

- [Microsoft 安全公告 CVE-2018年-0787: ASP.NET Core 提升特权漏洞](#)

Visual Studio 发布 ASP.NET 核心应用程序部署的配置文件

2018/4/27 • 14 min to read • [Edit Online](#)

作者: [Sayed Ibrahim Hashimi](#) 和 [Rick Anderson](#)

本文档重点介绍使用 Visual Studio 2017 创建和使用发布配置文件。可以从 MSBuild 和 Visual Studio 2017 运行使用 Visual Studio 创建的发布配置文件。有关发布到 Azure 的说明, 请参阅[使用 Visual Studio 将 ASP.NET Core Web 应用发布到 Azure App Service](#)。

以下项目文件已使用命令创建 `dotnet new mvc` :

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
  </ItemGroup>

</Project>
```

`<Project>` 元素的 `Sdk` 属性完成以下任务:

- 导入中的属性文件 `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.props` 开头。
- 在结束时从 `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets` 导入目标文件。

`MSBuildSDKsPath` (装有 Visual Studio 2017 Enterprise) 的默认位置是 `%programfiles(x86)%\Microsoft Visual Studio\2017\Enterprise\MSBuild\Sdks` 文件夹。

`Microsoft.NET.Sdk.Web` SDK 依赖于:

- `Microsoft.NET.Sdk.Web.ProjectSystem`
- `Microsoft.NET.Sdk.Publish`

这将导致产生以下属性和要导入的目标:

- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.props`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web.ProjectSystem\Sdk\Sdk.targets`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.props`
- `$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Publish\Sdk\Sdk.targets`

发布目标导入右侧集的目标根据使用的发布方法。

MSBuild 或 Visual Studio 将项目加载, 出现以下高级操作:

- 生成项目
- 计算要发布的文件
- 将文件发布到目标

计算项目项

加载项目时，将计算项目项(文件)。`item type` 属性确定如何处理该文件。默认情况下，.cs 文件包含在 `Compile` 项列表内。会对 `Compile` 项列表中的文件进行编译。

`Content` 项列表包含发布除了生成输出的文件。默认情况下，文件与模式匹配的 `wwwroot/**` 都将纳入 `Content` 项。`wwwroot/**` 组合模式匹配中的所有文件 `wwwroot` 文件夹和子文件夹。若要显式将文件添加到发布列表中，添加文件直接在 `.csproj` 文件中所示 [包含文件](#)。

选择时发布 Visual Studio 中或从命令行发布时的按钮：

- 计算属性/项目(需要生成的文件)。
- **Visual Studio** 仅：还原 NuGet 包。(用户需要在 CLI 上执行显式还原。)
- 生成项目。
- 计算发布项(需要发布的文件)。
- 该项目发布 (计算的文件复制到发布目标)。

当 ASP.NET Core 项目引用 `Microsoft.NET.Sdk.Web` 在项目文件中，`app_offline.htm` 文件放置在 web 应用程序目录的根目录。该文件存在时，ASP.NET Core 模块会在部署过程中正常关闭该应用并提供 `app_offline.htm` 文件。有关详细信息，请参阅 [ASP.NET Core 模块配置参考](#)。

基本的命令行发布

命令行发布适用于所有.NET 核心支持的平台，而且不需要 Visual Studio。在下面的示例中 [dotnet 发布](#) 从项目目录运行命令 (其中包含 `.csproj` 文件)。如果未在项目文件夹中，显式传入的项目文件路径。例如：

```
dotnet publish C:\Webs\Web1
```

运行以下命令以创建并发布 Web 应用：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
dotnet new mvc  
dotnet publish
```

[Dotnet 发布](#) 命令生成类似于以下输出：

```
C:\Webs\Web1>dotnet publish  
Microsoft (R) Build Engine version 15.3.409.57025 for .NET Core  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\Web1.dll  
Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp2.0\publish\
```

默认发布文件夹为 `bin\$(Configuration)\netcoreapp<version>\publish`。默认值为 `$(Configuration)` 是调试。在前面的示例中，`<TargetFramework>` 是 `netcoreapp2.0`。

`dotnet publish -h` 显示用于发布的帮助信息。

以下命令指定 `Release` 生成和发布目录：

```
dotnet publish -c Release -o C:\MyWebs\test
```

`Dotnet publish` 命令调用 MSBuild，调用 `Publish` 目标。任何参数传递给 `dotnet publish` 传递到 MSBuild。`-c` 参数映射到 `Configuration` MSBuild 属性。`-o` 参数映射到 `OutputPath`。

可以使用以下格式之一传递 MSBuild 属性：

- `p:<NAME>=<VALUE>`
- `/p:<NAME>=<VALUE>`

以下命令将 `Release` 版本发布到网络共享：

```
dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb
```

网络共享通过正斜杠指定 (`//r8/`) 并适用于所有支持 .NET Core 的平台。

确认用于部署的发布应用未在运行。如果应用正在运行，`publish` 文件夹中的文件会被锁定。部署不会发生，因为无法复制锁定的文件。

发布配置文件

本部分使用 Visual Studio 2017 创建发布配置文件。创建后，从 Visual Studio 或命令行发布是可用。

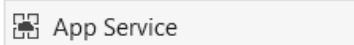
发布配置文件可以简化发布的过程中，并可以存在任意数量的配置文件。在 Visual Studio 中创建的发布配置文件，通过选择以下路径之一：

- 右键单击解决方案资源管理器中的项目并选择发布。
- 选择发布 <文件的内容> 从生成菜单。

发布显示在应用程序容量页的选项卡。如果项目缺少的发布配置文件，将显示的以下页面：

x

Pick a publish target



App Service



Azure Virtual Machines



IIS, FTP, etc



Folder

Azure App Service

Fully managed, and highly scalable cloud environment

Create New

Select Existing

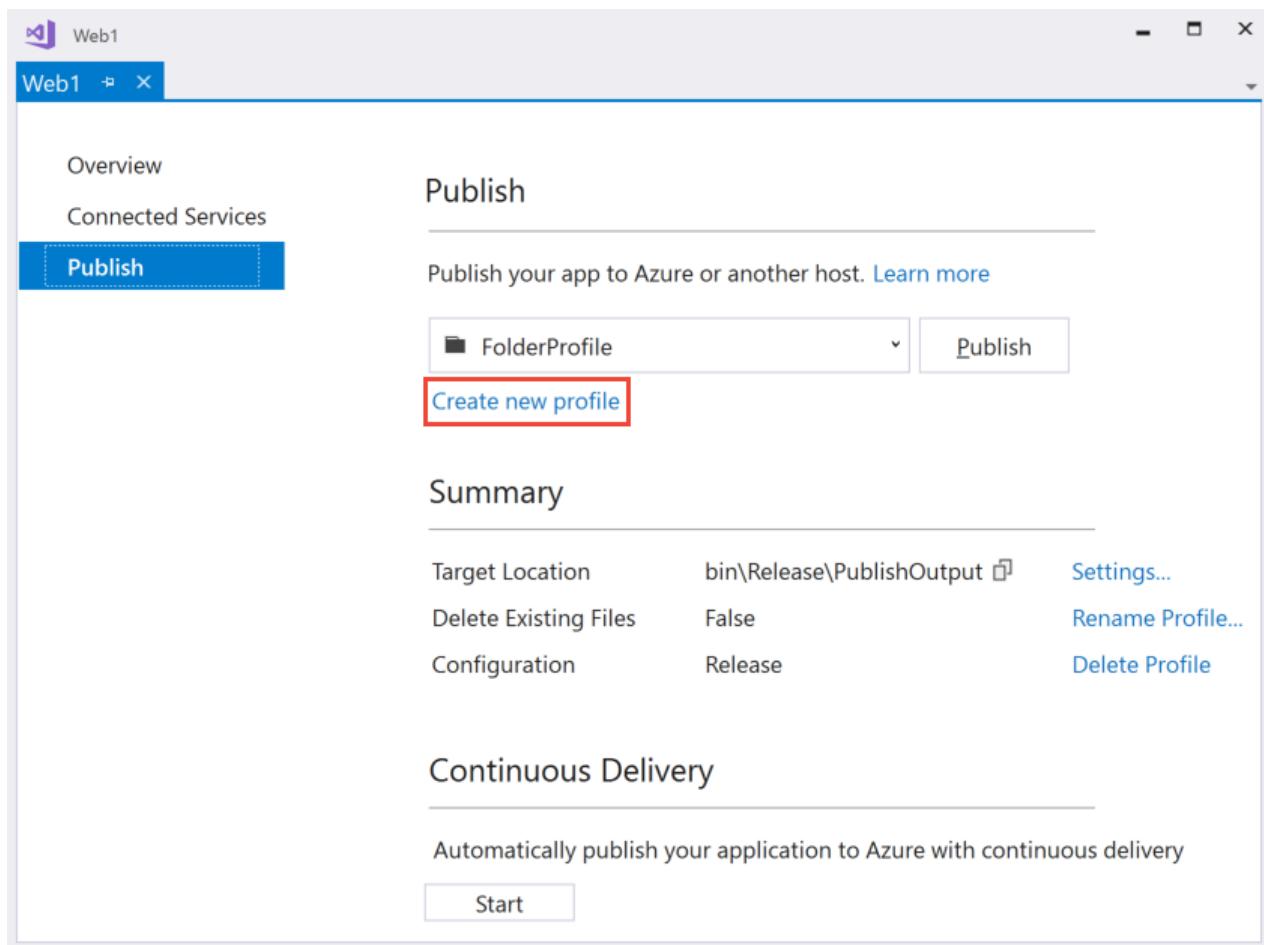
Import Profile...

Create Profile ▾

Cancel

当文件夹是选中，请指定文件夹路径来存储已发布的资产。默认文件夹是`bin\Release\PublishOutput`。单击创建配置文件按钮以完成。

发布配置文件创建后，发布选项卡上更改。下拉列表中显示新创建的配置文件。单击创建新的配置文件创建另一个新的配置文件。



发布向导支持以下发布目标：

- Azure 应用服务
- Azure 虚拟机
- IIS、FTP、等（适用于任何 web 服务器）
- 文件夹
- 导入配置文件

有关详细信息，请参阅[哪些发布选项是适合我。](#)

使用 Visual Studio 中，创建的发布配置文件时属性/PublishProfiles/<profile_name>.pubxml 创建 MSBuild 文件。.Pubxml 文件是 MSBuild 文件，包含发布配置设置。可以更改此文件为自定义生成和发布过程。通过发布过程读取此文件。`<LastUsedBuildConfiguration>` 是特殊的因为它的全局属性，且不应是在生成中导入任何文件中。请参阅[MSBuild：如何设置配置属性](#)有关详细信息。

当发布到 Azure 的目标，.pubxml 文件包含你的 Azure 订阅标识符。与该目标类型，不建议将此文件添加到源代码管理。在发布到非 Azure 目标时，则可以安全地签入 .pubxml 文件。

敏感信息（如发布密码）在上加密每用户/计算机级别。存储在属性/PublishProfiles/<profile_name>.pubxml.user 文件。此文件可以存储敏感信息，因为它不应签入源控件。

有关如何发布 web 应用程序在 ASP.NET Core 上的概述，请参阅[主机并将其部署](#)。MSBuild 任务和发布 ASP.NET Core 应用所需的目标是开放源代码在<https://github.com/aspnet/websdk>。

`dotnet publish` 可以使用文件夹，MS Deploy，和 Kudu 发布配置文件：

(可跨平台运行) 的文件夹：

```
dotnet publish WebApplication.csproj /p:PublishProfile=<FolderProfileName>
```

MSDeploy (当前此仅适用于 Windows 由于 MSDeploy 不跨平台中) :

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployProfileName> /p:Password=<DeploymentPassword>
```

MSDeploy 包 (当前此仅适用于 Windows 由于 MSDeploy 不跨平台中) :

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployPackageProfileName>
```

在前面的示例中, 不传递 `deployonbuild` 到 `dotnet publish`。

有关详细信息, 请参阅[Microsoft.NET.Sdk.Publish](#)。

`dotnet publish` 支持 Kudu API 从任何平台发布到 Azure。Visual Studio 发布 Kudu API, 但它支持通过 WebSDK 发布到 Azure 的跨平台的支持。

添加到的发布配置文件属性/PublishProfiles文件夹使用以下内容:

```
<Project>
<PropertyGroup>
  <PublishProtocol>Kudu</PublishProtocol>
  <PublishSiteName>nodewebapp</PublishSiteName>
  <UserName>username</UserName>
  <Password>password</Password>
</PropertyGroup>
</Project>
```

运行以下命令以压缩发布内容, 然后将其发布到 Azure 中使用 Kudu API:

```
dotnet publish /p:PublishProfile=Azure /p:Configuration=Release
```

使用发布配置文件时, 请设置以下 MSBuild 属性:

- `DeployOnBuild=true`
- `PublishProfile=<Publish profile name>`

发布与名为配置文件时 *FolderProfile*, 可以执行以下命令之一:

- `dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`
- `msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`

在调用时 `dotnet 生成`, 它调用 `msbuild` 来运行生成和发布过程。调用 `dotnet build` 或 `msbuild` 相当时传递文件夹配置文件中。调用时 MSBuild 直接在 Windows 上, 使用 MSBuild 的.NET Framework 版本。MSDeploy 目前仅限于在 Windows 计算机上进行发布。在非文件夹配置文件上调用 `dotnet build` 时, 会调用 MSBuild, 并且 MSBuild 在非文件夹配置文件上使用 MSDeploy。在非文件夹配置文件上调用 `dotnet build` 时, 会调用 MSBuild(使用 MSDeploy)并导致失败(即使在 Windows 平台上运行也是如此)。若要使用非文件夹配置文件进行发布, 请直接调用 MSBuild。

以下文件夹发布配置文件通过 Visual Studio 创建, 并被发布到网络共享:

```

<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework>netcoreapp1.1</PublishFramework>
    <ProjectGuid>c30c453c-312e-40c4-aec9-394a145dee0b</ProjectGuid>
    <publishUrl>\r8\Release\AdminWeb</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
</Project>

```

请注意，`<LastUsedBuildConfiguration>` 已设置为 `Release`。从 Visual Studio 发布时，在启动发布过程后将使用该值设置 `<LastUsedBuildConfiguration>` 配置属性值。`<LastUsedBuildConfiguration>` 配置属性是特殊，并且不应在导入的 MSBuild 文件中重写。从命令行，可以重写此属性。

使用.NET Core CLI:

```
dotnet build -c Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

使用 MSBuild:

```
msbuild /p:Configuration=Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

从命令行发布到 MSDeploy 终结点

可以使用 MSBuild 的.NET 核心 CLI 完成发布。`dotnet publish` 在 .NET Core 的上下文中运行。`msbuild` 命令需要.NET Framework，限制为 Windows 环境。

使用 MSDeploy 发布的最简单的方法是，首先在 Visual Studio 2017 中创建发布配置文件，然后从命令行中使用配置文件。

在下面的示例中，创建一个 ASP.NET 核心 web 应用（使用 `dotnet new mvc`），并使用 Visual Studio 添加 Azure 发布配置文件。

运行 `msbuild` 从 VS 2017 的开发人员命令提示符。开发人员命令提示已正确 `msbuild.exe` 在与某些 MSBuild 变量集其路径中。

MSBuild 使用以下语法：

```
msbuild <path-to-project-file> /p:DeployOnBuild=true /p:PublishProfile=<Publish Profile> /p:Username=<USERNAME> /p:Password=<PASSWORD>
```

获取 `password` 从 * <发布名称>。PublishSettings 文件。下载。PublishSettings* 从文件：

- 解决方案资源管理器：在 Web 应用上右键单击并选择下载发布配置文件。
- Azure 门户：单击 **Get** 发布配置文件 Web 应用上概述面板。

可在发布配置文件中找到 `<Username>`。

下面的示例使用 `Web11112-Web` 部署发布配置文件：

```
msbuild "C:\Webs\Web1\Web1.csproj" /p:DeployOnBuild=true  
/p:PublishProfile="Web11112 - Web Deploy" /p:Username="$Web11112"  
/p:Password=""
```

排除文件

在发布 ASP.NET Core Web 应用时，生成项目和 `wwwroot` 文件夹的内容包括在内。`msbuild` 支持通配模式。例如，以下 `<Content>` 元素排除的所有文本 (`.txt`) 从文件 `wwwroot/content` 文件夹和所有子文件夹。

```
<ItemGroup>  
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never" />  
</ItemGroup>
```

前面的标记可以添加到发布配置文件或 `.csproj` 文件。添加到 `.csproj` 文件时，会将该规则添加到项目中的所有发布配置文件中。

以下 `<MsDeploySkipRules>` 元素排除中的所有文件 `wwwroot/content` 文件夹：

```
<ItemGroup>  
  <MsDeploySkipRules Include="CustomSkipFolder">  
    <ObjectName>dirPath</ObjectName>  
    <AbsolutePath>wwwroot\\content</AbsolutePath>  
  </MsDeploySkipRules>  
</ItemGroup>
```

`<MsDeploySkipRules>` 不会删除跳过部署站点中的目标。`<Content>` 从部署站点删除目标的文件和文件夹。例如，假设已部署的 web 应用程序有以下文件：

- `Views/Home/About1.cshtml`
- `Views/Home/About2.cshtml`
- `Views/Home/About3.cshtml`

如果以下 `<MsDeploySkipRules>` 添加元素，不会在部署站点上删除这些文件。

```
<ItemGroup>  
  <MsDeploySkipRules Include="CustomSkipFile">  
    <ObjectName>filePath</ObjectName>  
    <AbsolutePath>Views\\Home\\About1.cshtml</AbsolutePath>  
  </MsDeploySkipRules>  
  
  <MsDeploySkipRules Include="CustomSkipFile">  
    <ObjectName>filePath</ObjectName>  
    <AbsolutePath>Views\\Home\\About2.cshtml</AbsolutePath>  
  </MsDeploySkipRules>  
  
  <MsDeploySkipRules Include="CustomSkipFile">  
    <ObjectName>filePath</ObjectName>  
    <AbsolutePath>Views\\Home\\About3.cshtml</AbsolutePath>  
  </MsDeploySkipRules>  
</ItemGroup>
```

前面 `<MsDeploySkipRules>` 元素防止跳过不会部署的文件。它们在部署后，它不会删除这些文件。

以下 `<Content>` 元素删除在部署站点的目标的文件：

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never" />
</ItemGroup>
```

使用命令行部署前面 `<Content>` 元素生成下面的输出：

```
MSDeployPublish:
  Starting Web deployment task from source:
manifest(C:\Webs\Web1\obj\Release\netcoreapp1.1\PubTmp\Web1.SourceManifest.xml) to Destination: auto().
  Deleting file (Web11112\Views\Home\About1.cshtml).
  Deleting file (Web11112\Views\Home\About2.cshtml).
  Deleting file (Web11112\Views\Home\About3.cshtml).
  Updating file (Web11112\web.config).
  Updating file (Web11112\Web1.deps.json).
  Updating file (Web11112\Web1.dll).
  Updating file (Web11112\Web1.pdb).
  Updating file (Web11112\Web1.runtimeconfig.json).
  Successfully executed Web deployment task.
  Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

包含文件

以下标记包括映像文件夹到在项目目录外的 `wwwroot/images` 发布站点文件夹：

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotnetPublishFiles Include="@(_CustomFiles)">
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%{filename}%(Extension)</DestinationRelativePath>
  </DotnetPublishFiles>
</ItemGroup>
```

可以将标记添加到 `.csproj` 文件或发布配置文件。如果将其添加到 `.csproj` 文件，它包含在项目中每个发布配置文件中。

以下突出显示的标记显示如何：

- 将文件从项目外部复制到 `wwwroot` 文件夹。
- 排除 `wwwroot\Content` 文件夹。
- 排除 `Views\Home\About2.cshtml`。

```

<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>afa9f185-7ce0-4935-9da1-ab676229d68a</ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
  <ItemGroup>
    <ResolvedFileToPublish Include=".\\ReadMe2.MD">
      <RelativePath>wwwroot\\ReadMe2.MD</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\\Content\\**\\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\\Home\\About2.cshtml" CopyToPublishDirectory="Never" />
  </ItemGroup>
</Project>

```

请参阅 [WebSDK 自述文件](#)，了解更多部署示例。

在发布前或发布后运行目标

内置 `BeforePublish` 和 `AfterPublish` 目标执行目标之前或之后发布目标。将以下元素添加到要记录控制台消息之前和之后发布的发布配置文件：

```

<Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
  <Message Text="Inside BeforePublish" Importance="high" />
</Target>
<Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
  <Message Text="Inside AfterPublish" Importance="high" />
</Target>

```

将发布到使用不受信任的证书的服务器

添加 `<AllowUntrustedCertificate>` 属性值为 `True` 到发布配置文件：

```

<PropertyGroup>
  <AllowUntrustedCertificate>True</AllowUntrustedCertificate>
</PropertyGroup>

```

Kudu 服务

若要在 Azure App Service web 应用程序部署中查看的文件，使用[Kudu 服务](#)。追加 `scm` 令牌 web 应用名。例如：

URL	结果
<code>http://mysite.azurewebsites.net/</code>	Web 应用
<code>http://mysite.scm.azurewebsites.net/</code>	Kudu 服务

选择[调试控制台](#)菜单项以查看、编辑、删除或添加文件。

其他资源

- [Web 部署](#)(MSDeploy) 简化了部署 web 应用和到 IIS 服务器的网站。
- <https://github.com/aspnet/websdk>: 文件的问题并请求适用于部署的功能。

ASP.NET 核心目录结构

2018/5/14 • 2 min to read • [Edit Online](#)

作者: [Luke Latham](#)

在 ASP.NET 核, 发布的应用程序目录中, 发布, 组成应用程序文件、配置文件、静态资产、包和运行时 (有关自包含的部署)。

应用类型	目录结构
依赖于框架的部署	<ul style="list-style-type: none">● 发布†<ul style="list-style-type: none">○ 日志†(除非需要接收 stdout 日志可选)○ 视图†(MVC 应用程序; 如果不预编译视图)○ 页†(MVC 或 Razor 页应用; 如果不预编译页)○ wwwroot†○ *.dll 文件○ <程序集名称>.deps.json○ <程序集名称>.dll○ <程序集名称>.pdb○ <程序集名称>.PrecompiledViews.dll○ <程序集名称>.PrecompiledViews.pdb○ <程序集名称>.runtimeconfig.json○ web.config (IIS 部署)
自包含的部署	<ul style="list-style-type: none">● 发布†<ul style="list-style-type: none">○ 日志†(除非需要接收 stdout 日志可选)○ refst†○ 视图†(MVC 应用程序; 如果不预编译视图)○ 页†(MVC 或 Razor 页应用; 如果不预编译页)○ wwwroot†○ *.dll 文件○ <程序集名称>.deps.json○ <程序集名称>.exe○ <程序集名称>.pdb○ <程序集名称>.PrecompiledViews.dll○ <程序集名称>.PrecompiledViews.pdb○ <程序集名称>.runtimeconfig.json○ web.config (IIS 部署)

†指示目录

发布目录表示内容的根路径, 也称为应用程序基路径, 部署。任何名称提供给发布目录中的服务器上部署的应用程序, 其位置用作托管的应用程序服务器的物理路径。

Wwwroot 目录中, 如果存在, 仅包含静态资产。

Stdout 日志可以使用以下两种方法之一部署为创建目录:

- 添加以下 `<Target>` 项目文件的元素:

```
<Target Name="CreateLogsFolder" AfterTargets="Publish">
  <MakeDir Directories="$(PublishDir)Logs"
    Condition="!Exists('$(PublishDir)Logs')" />
  <WriteLinesToFile File="$(PublishDir)Logs\log"
    Lines="Generated file"
    Overwrite="True"
    Condition="!Exists('$(PublishDir)Logs\log')" />
</Target>
```

`<MakeDir>` 元素创建一个空日志中已发布的输出文件夹。元素使用 `PublishDir` 属性来确定用于创建文件夹的目标位置。多种部署方法，例如 Web 部署，在部署过程中跳过空文件夹。`<WriteLinesToFile>` 元素生成的文件中日志文件夹中，这可确保部署到服务器的文件夹。请注意，如果工作进程不具有写访问权限的目标文件夹的文件夹创建可能仍会失败。

- 以物理方式创建日志目录部署中的服务器。

部署目录中需要读取/Execute 权限。日志目录需要读/写权限。其中写入文件的其他目录需要读/写权限。

Azure App Service 和 ASP.NET Core 与 IIS 的常见错误参考

2018/5/18 • 11 min to read • [Edit Online](#)

作者: [Luke Latham](#)

以下不是错误的完整列表。如果你遇到此处未列出的错误[打开一个新问题](#)与再现错误的详细说明。

收集以下信息:

- 浏览器的行为
- 应用程序事件日志条目
- ASP.NET 核心模块 stdout 日志条目

比较以下常见错误的信息。如果找到匹配项, 请按照故障排除建议。

重要事项

要注意, 对于使用 **ASP.NET 核心 2.1 预览版本**

请参阅[到 Azure App Service 部署 ASP.NET Core 预览版](#)。

安装程序无法获取 VC++ Redistributable

- 安装程序异常: 0x80072efd 或 0x80072f76 - 未指定的错误
- 安装程序日志异常[†]: 0x80072efd 或 0x80072f76 错误:未能执行 EXE 包

[†]此日志位于 C:\Users\{USER}\AppData\Local\Temp\dd_DotNetCoreWinSvrHosting_{timestamp}.log。

疑难解答:

- 如果安装承载捆绑时, 系统不具有 Internet 访问权限, 此异常时发生安装已阻止获取Microsoft Visual c ++ 2015年可再发行组件。获取从安装[Microsoft Download Center](#)。如果安装失败, 服务器可能不会收到托管依赖于框架的部署 (FDD) 所需.NET Core 运行时。如果托管 FDD, 确认在程序中安装了运行时&功能。如果需要获取从运行时安装[.NET 所有下载](#)。安装运行时后, 重启系统, 或通过从命令提示符依次执行 net stop was /y 和 net start w3svc 来重启 IIS。

OS 升级删除了 32 位 ASP.NET Core 模块

- 应用程序日志: 未能加载模块 DLL C:\WINDOWS\system32\inetsrv\aspnetcore.dll。该数据是个错误。

疑难解答:

- OS 升级期间不会保留 C:\Windows\SysWOW64\inetsrv 目录中的非 OS 文件。如果之前安装 ASP.NET 核心模块操作系统升级, 然后任何应用程序池运行时在 32 位模式下 OS 升级之后, 遇到此问题。在 OS 升级后修复 ASP.NET Core 模块。请参阅[安装.NET 核心承载捆绑](#)。选择修复安装运行时。

平台与 RID 冲突

- 浏览器: HTTP 错误 502.5 - 进程失败
- 应用程序日志: 应用程序 'MACHINE/WEBROOT/APPHOST/{程序集}' 与物理根 c:{路径}'无法使用命令

行启动进程"c:\{路径} {程序集}。{exe | dll}"，错误代码 ="0x80004005: ff。

- **ASP.NET 核心模块日志**：未经处理的异常：System.BadImageFormatException：无法加载文件或程序集 {程序集}.dll。试图加载的程序的格式不正确。

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅[故障排除](#)。
- 确认 `<PlatformTarget>` 中 `.csproj` 不会与 RID 冲突。例如，未指定 `<PlatformTarget>` 的 `x86` 和发布与的 RID `win10-x64`，通过使用 `dotnet publish -c win10-x64` 或通过设置 `<RuntimeIdentifiers>` 中 `.csproj` 到 `win10-x64`。项目在不显示任何警告或错误的情况下发布，但会失败，且系统上出现上述记录的异常。
- 如果此异常发生在 Azure 应用程序部署升级应用程序时，以及手动部署更新的程序集，从以前的部署中删除所有文件。部署升级的应用时，延迟的不兼容程序集可能造成 `System.BadImageFormatException` 异常。

URI 终结点错误或网站已停止

- 浏览器：ERR_CONNECTION_REFUSED
- 应用程序日志：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 确认正在使用正确的 URI 终结点应用程序。请检查绑定中。
- 确认 IIS 网站不在已停止状态。

已禁用 CoreWebEngine 或 W3SVC 服务器功能

- **OS 异常**：必须安装 IIS 7.0 CoreWebEngine 和 W3SVC 功能，以便使用 ASP.NET Core 模块。

疑难解答：

- 确认已启用适当的角色和功能。请参阅 [IIS 配置](#)。

不正确的网站物理路径或缺少应用程序

- 浏览器：403 禁止访问 - 访问被拒绝 --或者-- 403.14 禁止访问 - Web 服务器被配置为不列出此目录的内容。
- 应用程序日志：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 检查 IIS 网站基本设置和物理应用文件夹。确认应用程序是否在 IIS 网站的文件夹中物理路径。

角色不正确、未安装模块或权限不正确

- 浏览器：500.19 内部服务器错误 - 无法访问请求的页面，因为该页面的相关配置数据无效。
- 应用程序日志：没有任何条目
- **ASP.NET Core 模块日志**：未创建日志文件

疑难解答：

- 确认启用了适当的角色。请参阅 [IIS 配置](#)。
- 检查“程序和功能”，确认已安装 Microsoft ASP.NET Core 模块。如果已安装程序列表中没有 Microsoft ASP.NET Core 模块，请安装该模块。请参阅 [安装.NET 核心承载捆绑](#)。
- 请确保应用程序池 > 进程模型 > 标识设置为**ApplicationPoolIdentity**或自定义标识具有访问应用程序的部署文件夹的正确权限。

不正确 processPath、缺少 PATH 变量、承载捆绑未安装，不重新启动系统/IIS、VC + + 可再发行组件未安装，或 dotnet.exe 访问冲突

- 浏览器：**HTTP 错误 502.5 - 进程失败
- 应用程序日志：**应用程序 'MACHINE/WEBROOT/APPHOST/{程序集}' 与物理根 c:\{路径}'无法使用命令行启动进程"。{程序集}.exe"，错误代码 = "0x80070002: 0"。
- ASP.NET Core 模块日志：**已创建日志文件，但该日志文件为空

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅 [故障排除](#)。
- 检查 **processPath** 属性 `<aspNetCore>` 中的元素 `web.config` 以确认它是 `dotnet` 为依赖于框架的部署 (FDD) 或 `.{程序集}.exe` 自包含的部署 (SCD)。
- 对于 FDD，可能无法通过路径设置访问 `dotnet.exe`。确认系统路径设置中存在 *C:\Program Files\dotnet*。
- 对于 FDD，应用程序池的用户标识可能无法访问 `dotnet.exe`。确认应用程序池用户标识具有访问 C:\Program Files\dotnet 目录的权限。确认没有拒绝规则配置上的应用程序池用户标识 C:\Program Files\dotnet 和应用程序目录。
- FDD 可能已部署，.NET 核心安装，无需重新启动 IIS。重启服务器，或通过从命令提示符依次执行 `net stop was /y` 和 `net start w3svc` 来重启 IIS。
- FDD 可能已不在主机系统上安装.NET Core 运行时的情况下部署。如果尚未安装.NET Core 运行时，运行 [.NET 核心承载捆绑安装](#) 系统上。请参阅 [安装.NET 核心承载捆绑](#)。如果尝试在未连接到 Internet 的系统上安装.NET Core 运行时，获取从运行时 [.NET 所有下载](#) 和运行承载捆绑的安装程序安装 ASP.NET 核心模块。重启系统，或通过从命令提示符依次执行 `net stop was /y` 和 `net start w3svc` 来重启 IIS，完成安装。
- FDD 可能已部署和 [Microsoft Visual c + + 2015 年可再发行组件 \(x64\)](#) 系统上未安装。获取从 [安装 Microsoft Download Center](#)。

`<aspNetCore>` 元素的参数不正确

- 浏览器：**HTTP 错误 502.5 - 进程失败
- 应用程序日志：**应用程序 'MACHINE/WEBROOT/APPHOST/{程序集}' 与物理根 c:\{路径}'无法使用命令行启动进程"dotnet"。{程序集}.dll，错误代码 = "0x80004005: 80008081"。
- ASP.NET 核心模块日志：**要执行的应用程序不存在：路径 {程序集}.dll

疑难解答：

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息，请参阅 [故障排除](#)。
- 检查 **参数属性** `<aspNetCore>` 中的元素 `web.config` 要确认它是否是 (a)。`{程序集}.dll` 依赖于框架的部署

(FDD); 或 (b) 不存在空字符串 (参数 = ""), 或应用程序的自变量列表 (参数 ="arg1, arg2, ...")为独立部署 (SCD)。

缺少 .NET Framework 版本

- 浏览器: 502.3 错误网关 - 尝试路由请求时出现连接错误。
- 应用程序日志: ErrorCode = 应用程序 ' MACHINE/WEBROOT/APPHOST / {程序集} 与物理根 c:\{路径}'无法使用命令行启动进程"dotnet"。{程序集}.dll, 错误代码 ="0x80004005: 80008081。
- **ASP.NET Core 模块日志:** 缺少方法、文件或程序集异常。在异常中指定的方法、文件或程序集是 .NET Framework 方法、文件或程序集。

疑难解答:

- 安装系统缺少的 .NET Framework 版本。
- 对于依赖于框架的部署 (FDD), 确认正确的运行时系统上安装。如果项目从 1.1 升级到 2.0, 部署到主机系统, 并且此异常结果, 请确保 2.0 framework 是在主机系统。

应用程序池已停止

- 浏览器: 503 服务不可用
- 应用程序日志: 没有任何条目
- **ASP.NET Core 模块日志:** 未创建日志文件

疑难解答

- 确认应用程序池不在已停止状态。

未实现 IIS 集成中间件

- 浏览器: HTTP 错误 502.5 - 进程失败
- 应用程序日志: 应用程序 ' MACHINE/WEBROOT/APPHOST / {程序集} 与物理根 c:\{路径}'使用命令行创建过程"c:\{路径}\{程序集}。{exe | dll}"但损坏或未响应, 或者没有侦听给定的端口 {PORT}, 错误代码 ="0x800705b4"
- **ASP.NET Core 模块日志:** 已创建日志文件, 且该日志文件显示正常操作。

疑难解答

- 确认应用在 Kestrel 上本地运行。进程失败可能是由应用的内部问题导致的。有关详细信息, 请参阅[故障排除](#)。
 - 请确认:
 - IIS 集成中间件是 referencedby 调用 `UseIISIntegration` 上应用的方法 `WebHostBuilder` (ASP.NET Core 1.x)
 - 应用程序使用 `CreateDefaultBuilder` 方法 (ASP.NET Core 2.x)。
- 请参阅[中 ASP.NET 核心主机](#)有关详细信息。

子应用程序包括 <handlers> 部分

- 浏览器: HTTP 错误 500.19 - 内部服务器错误
- 应用程序日志: 没有任何条目

- **ASP.NET 核心模块日志**: 日志文件创建并显示有关根应用程序的正常操作。不会为订阅应用程序创建的日志文件。

疑难解答

- 确认子应用的 web.config 文件不包括 `<handlers>` 部分。

stdout 日志路径不正确

- 浏览器: 应用程序通常响应。
- 应用程序日志: 警告: 无法创建 stdoutLogFile \?
`\C:_apps\app_folder\bin\Release\netcoreapp2.0\win10-x64\publish\logs\path_doesnt_exist\stdout_8748_201831835937.log`, 错误代码 =-2147024893。
- **ASP.NET Core 模块日志**: 未创建日志文件

疑难解答

- `stdoutLogFile` 中指定的路径 `<aspNetCore>` 元素 `web.config` 不存在。有关详细信息, 请参阅[记录创建和重定向 ASP.NET 核心模块配置参考主题的部分](#)。

应用程序配置常见问题

- 浏览器: HTTP 错误 502.5 - 进程失败
- 应用程序日志: 应用程序 'MACHINE/WEBROOT/APPHOST/{程序集}' 与物理根 c:\{路径}\ 使用命令行创建过程"c:\{路径}\{程序集}.exe | dll}"但损坏或未响应, 或者没有侦听给定的端口 {PORT}, 错误代码 ="0x800705b4"
- **ASP.NET Core 模块日志**: 已创建日志文件, 但该日志文件为空

疑难解答

- 此常规异常指示进程无法启动, 很可能由于应用程序配置问题。引用[目录结构](#), 确认应用程序的部署文件和文件夹是适当和应用程序的配置文件是否存在并且包含正确的设置为应用程序和环境。有关详细信息, 请参阅[故障排除](#)。

ASP.NET Core 安全性概述

2018/4/10 • 3 min to read • [Edit Online](#)

通过 ASP.NET Core，开发者可轻松配置和管理其应用的安全性。ASP.NET Core 的功能包括管理身份验证、授权、数据保护、SSL 强制、应用机密、请求防伪保护及 CORS 管理。通过这些安全功能，可以生成安全可靠的 ASP.NET Core 应用。

ASP.NET Core 安全性功能

ASP.NET Core 提供许多用于保护应用安全的工具和库（包括内置标识提供程序），但也可使用第三方标志服务（如 Facebook、Twitter 或 LinkedIn）。利用 ASP.NET Core 可以轻松管理应用机密，无需将机密信息暴露在代码中就可存储和使用它们。

身份验证 vs 授权

身份验证是这样一个过程：由用户提供凭据，然后将其与存储在操作系统、数据库、应用或资源中的凭据进行比较。在授权过程中，如果凭据匹配，则用户身份验证成功，可执行已向其授权的操作。授权指判断允许用户执行的操作的过程。

对身份验证的另一种理解是将其看作进入某一空间（如服务器、数据库、应用或资源）的方式，而将授权看作用户可对该空间（服务器、数据库或应用）内的对象执行的操作。

软件中的常见漏洞

ASP.NET Core 和 EF 提供维护应用安全、预防安全漏洞的功能。下表中链接的文档详细介绍了在 Web 应用中避免最常见安全漏洞的技术：

- 跨站点脚本攻击
- SQL 注入式攻击
- 跨站点请求伪造 (CSRF)
- 打开重定向攻击

还应注意其他漏洞。有关详细信息，请参阅本文档中关于 ASP.NET Core 安全文档的部分。

ASP.NET Core 安全文档

- [身份验证](#)
 - [标识简介](#)
 - [启用使用 Facebook、Google 和其他外部提供程序的身份验证](#)
 - [通过 WS 联合身份验证启用身份验证](#)
 - [配置 Windows 身份验证](#)
 - [帐户确认和密码恢复](#)
 - [使用 SMS 设置双因素身份验证](#)
 - [在没有标识的情况下使用 cookie 身份验证](#)
 - [Azure Active Directory](#)
 - [将 Azure AD 集成到 ASP.NET Core Web 应用中](#)
 - [使用 Azure AD 从 WPF 应用调用 ASP.NET Core Web API](#)
 - [使用 Azure AD 在 ASP.NET Core Web 应用中调用 Web API](#)
 - [带有 Azure AD B2C 的 ASP.NET Core Web 应用](#)

- 使用 IdentityServer4 保护 ASP.NET Core 应用
- 授权
 - 介绍
 - 通过授权保护的用户数据创建应用
 - 简单授权
 - 基于角色的授权
 - 基于声明的授权
 - 基于策略的授权
 - 要求处理程序中的依赖关系注入
 - 基于资源的授权
 - 基于视图的授权
 - 使用方案限制标识
- 数据保护
 - 数据保护简介
 - 数据保护 API 入门
 - 使用者 API
 - 使用者 API 概述
 - 目标字符串
 - 目标层次结构和多租户
 - 哈希密码
 - 限制受保护负载的生存期
 - 取消保护已撤消其密钥的负载
 - 配置
 - 配置数据保护
 - 默认设置
 - 计算机范围的策略
 - 非 DI 感知方案
 - 扩展性 API
 - 核心加密扩展性
 - 密钥管理扩展性
 - 其他 API
 - 实现
 - 已验证的加密详细信息
 - 子项派生和已验证的加密
 - 上下文标头
 - 密钥管理
 - 密钥存储提供程序
 - 静态密钥加密
 - 密钥永久性和设置
 - 密钥存储格式
 - 短数据保护提供程序
 - 兼容性
 - 在 ASP.NET 中替换
- 通过授权保护的用户数据创建应用
- 在开发期间安全存储应用机密
- Azure Key Vault 配置提供程序

- 强制实施 SSL
- 防请求伪造
- 阻止打开重定向攻击
- 阻止跨站点脚本编写
- 启用跨域请求 (CORS)
- 在应用之间共享 Cookie

ASP.NET Core 中的身份验证

2018/3/19 • 1 min to read • [Edit Online](#)

- [社区 OSS 身份验证选项](#)
- [标识简介](#)
- [启用使用 Facebook、Google 和其他外部提供程序的身份验证](#)
- [通过 WS 联合身份验证启用身份验证](#)
- [在标识中启用 QR 代码生成](#)
- [配置 Windows 身份验证](#)
- [帐户确认和密码恢复](#)
- [使用 SMS 设置双因素身份验证](#)
- [在没有标识的情况下使用 cookie 身份验证](#)
- [Azure Active Directory
 - \[将 Azure AD 集成到 ASP.NET Core Web 应用中\]\(#\)
 - \[将 Azure AD B2C 集成到面向客户的 ASP.NET Core Web 应用中\]\(#\)
 - \[Integrate Azure AD B2C into an ASP.NET Core web API\\(将 Azure AD B2C 集成到 ASP.NET Core Web API 中\\)\]\(#\)
 - \[使用 Azure AD 从 WPF 应用调用 ASP.NET Core Web API\]\(#\)
 - \[使用 Azure AD 在 ASP.NET Core Web 应用中调用 Web API\]\(#\)](#)
- [使用 IdentityServer4 保护 ASP.NET Core 应用](#)
- [使用 Azure App Service 身份验证保护 ASP.NET Core 应用\(简易身份验证\)](#)
- [基于用个人用户帐户创建的项目的文章](#)

用于 ASP.NET Core 社区 OSS 身份验证选项

2018/3/15 • 1 min to read • [Edit Online](#)

此页包含用于 ASP.NET Core 提供社区，开放源代码身份验证选项。此页将定期更新为变得可用的新提供程序。

OSS 身份验证提供程序

下面的列表按字母顺序排列。

NAME	描述
AspNet.Security.OpenIdConnect.Server (ASOS)	ASOS 是一个低级别，协议首先 OpenID Connect 服务器框架为 ASP.NET Core 和 OWIN/Katana。
IdentityServer	IdentityServer 是 OpenID Connect 和 OAuth 2.0 的框架，用于 ASP.NET 核心，正式认证 OpenID foundation 和 .NET Foundation 监管下。有关详细信息，请参阅 欢迎 IdentityServer4 (文档) 。
OpenIddict	OpenIddict 是能够轻松使用 OpenID Connect 服务器 ASP.NET Core。
Cierge	Cierge 是 OpenID Connect 服务器处理用户注册、登录名、配置文件、管理、社交登录名，和的详细信息。

若要添加提供程序，[编辑此页](#)。

在 ASP.NET Core 上的标识简介

2018/5/17 • 11 min to read • [Edit Online](#)

通过[Pranav Rastogi](#), [Rick Anderson](#), [Tom Dykstra](#), [Jon Galloway](#)[艾力克 Reitan](#), 和[Steve Smith](#)

ASP.NET 核心标识是允许你向你的应用程序添加登录功能的成员身份系统。用户可以创建帐户和登录名使用的用户名和密码或它们可以使用如 Facebook、Google、Microsoft 帐户、Twitter 或其他外部登录提供程序。

你可以配置 ASP.NET 核心标识来使用 SQL Server 数据库来存储用户名、密码和配置文件数据。或者，你可以使用你自己的持久存储区，例如，Azure 表存储。本文档包含用于 Visual Studio 以及有关使用 CLI 的说明。

[查看或下载的示例代码。\(如何下载\)](#)

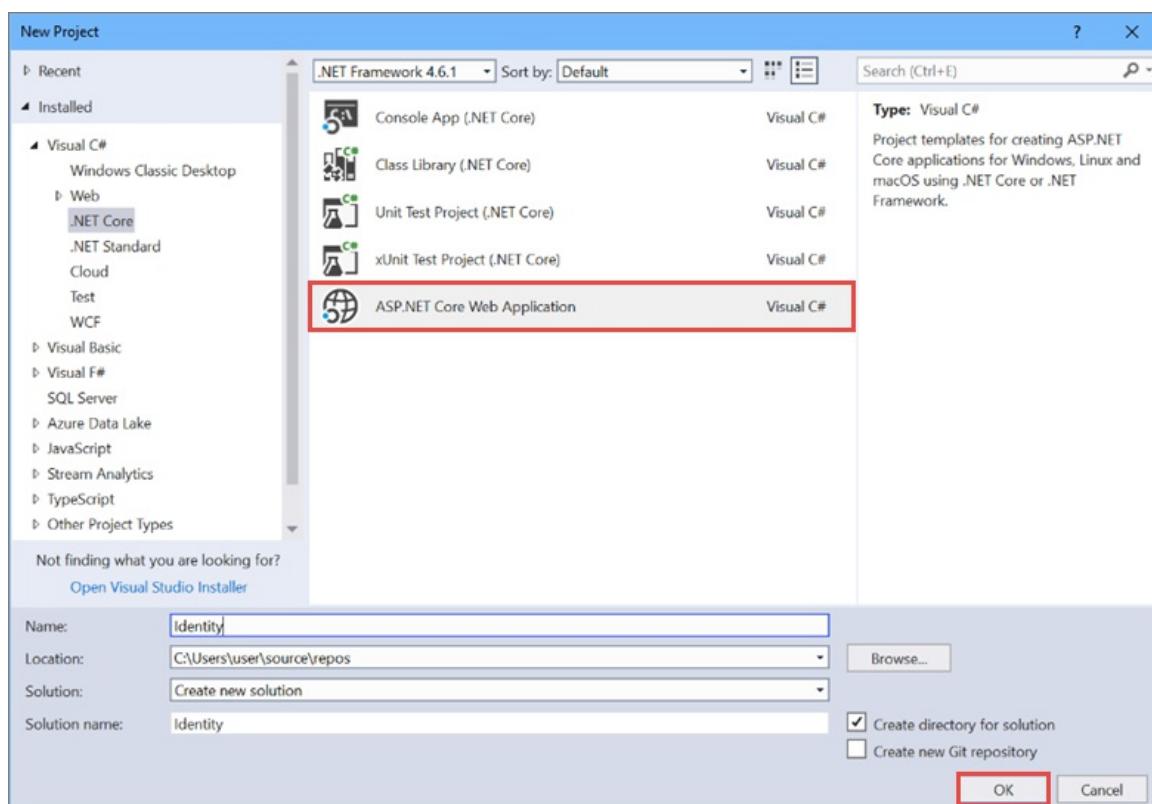
标识的概述

在本主题中，你将了解如何使用 ASP.NET 核心标识来添加功能，用于注册、登录，并注销用户。有关创建使用 ASP.NET 核心标识应用的更多详细说明，请参阅本文末尾的后续步骤部分。

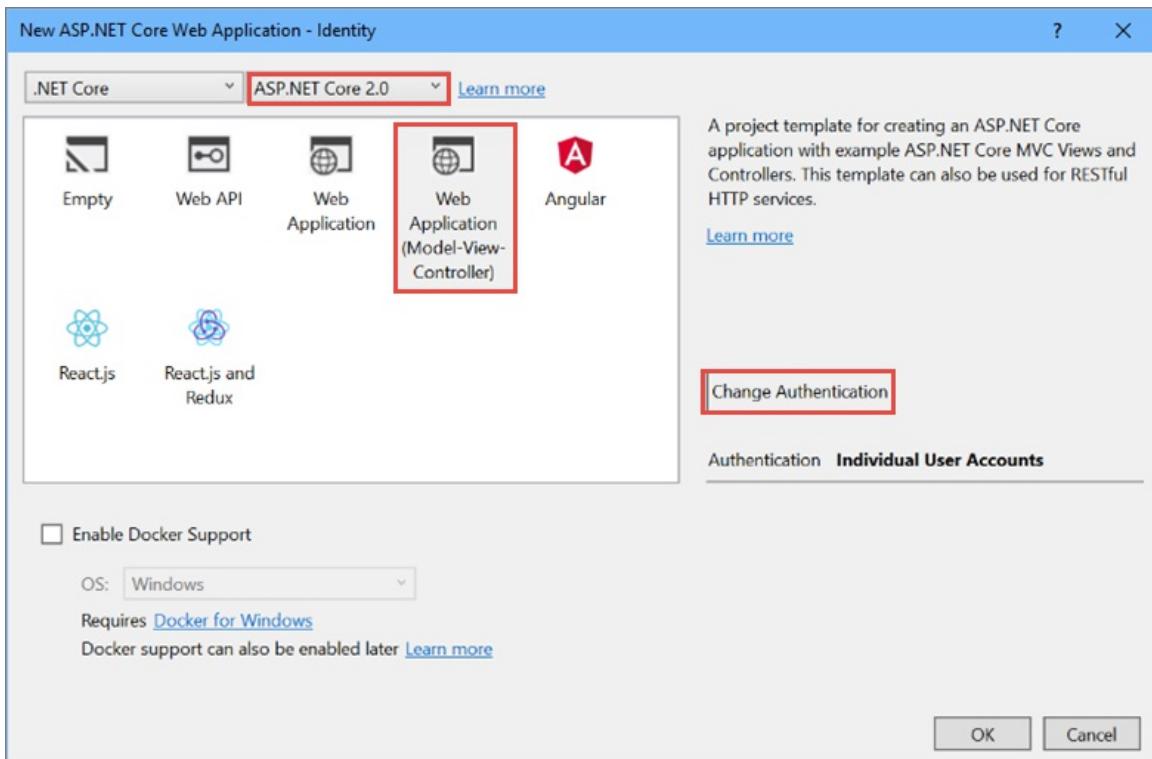
1. 使用单个用户帐户创建一个 ASP.NET 核心 Web 应用程序项目。

- [Visual Studio](#)
- [.NET Core CLI](#)

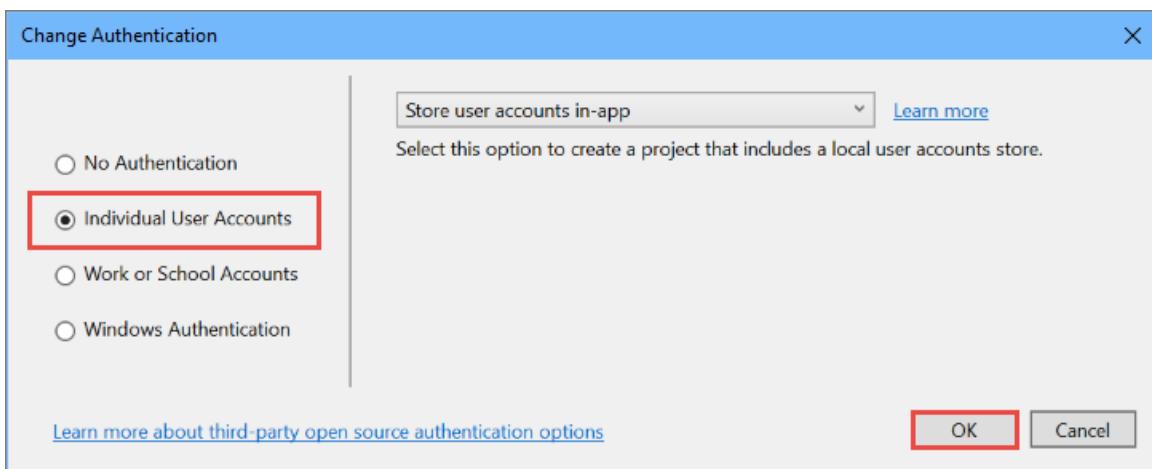
在 Visual Studio 中，选择**文件 > 新建 > 项目**。选择**ASP.NET 核心 Web 应用程序**单击**确定**。



选择 ASP.NET Core Web 应用程序（模型-视图-控制器）asp.net 核心 2.x，然后选择**更改身份验证**。



出现一个对话框，产品/服务身份验证选项。选择单个用户帐户单击确定以返回到上一个对话框。



选择单个用户帐户指示 Visual Studio 来创建模型、Viewmodel、视图、控制器和其他资产所需的身份验证作为项目模板的一部分。

2. 配置标识服务，并添加中的中间件 `Startup`。

标识服务添加到中的应用程序 `ConfigureServices` 中的方法 `Startup` 类：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;
        options.Password.RequiredUniqueChars = 6;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.AllowedForNewUsers = true;

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
        // If the LoginPath isn't set, ASP.NET Core defaults
        // the path to /Account/Login.
        options.LoginPath = "/Account/Login";
        // If the AccessDeniedPath isn't set, ASP.NET Core defaults
        // the path to /Account/AccessDenied.
        options.AccessDeniedPath = "/Account/AccessDenied";
        options SlidingExpiration = true;
    });

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}

```

这些服务都提供给应用程序通过[依赖关系注入](#)。

通过调用情况下，启用应用程序的标识 `UseAuthentication` 中 `Configure` 方法。`UseAuthentication` 添加了身份验证[中间件](#)向请求管道。

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
        app.UseDatabaseErrorHandler();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseAuthentication();

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

有关进程的应用程序启动的详细信息，请参阅[应用程序启动](#)。

3. 创建一个用户。

启动应用程序，然后单击[注册](#)链接。

如果这是你要执行此操作的第一个时间，你可能需要运行迁移。应用程序会提示您[应用迁移](#)。如果需要请刷新页面。

A database operation failed while processing the request.
`SqlException: Cannot open database "aspnet-identityDemo-e5ecfa96-0c92-4175-a514-21049ad9d8b3" requested by the login. The login failed. Login failed for user 'OSIRIS\steve_000'.`

Applying existing migrations for ApplicationDbContext may resolve this issue
 There are migrations for ApplicationDbContext that have not been applied to the database
 • 000000000000_CreatedIdentitySchema

[Apply Migrations](#)

In Visual Studio, you can use the Package Manager Console to apply pending migrations to the database:
 PM> Update-Database

Alternatively, you can apply pending migrations from a command prompt at your project directory:
 > dotnet ef database update

或者，你可以测试与你的应用不持久的数据库的情况下通过使用内存中数据库的 ASP.NET 核心标识。若要使用的内存中数据库，添加 `Microsoft.EntityFrameworkCore.InMemory` 包到你的应用程序和修改你的应用程序调用 `AddDbContext` 中 `ConfigureServices`，如下所示：

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseInMemoryDatabase(Guid.NewGuid().ToString()));
```

当用户单击[注册](#)链接，`Register` 上调用操作 `AccountController`。`Register` 操作通过调用创建用户 `CreateAsync` 上 `_userManager` 对象（提供给 `AccountController` 通过依赖关系注入）：

```

// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // For more information on how to enable account confirmation and password reset please
            // visit http://go.microsoft.com/fwlink/?LinkID=532713
            // Send an email with this link
            //var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            //var callbackUrl = Url.Action("ConfirmEmail", "Account", new { userId = user.Id, code =
            code }, protocol: HttpContext.Request.Scheme);
            //await _emailSender.SendEmailAsync(model.Email, "Confirm your account",
            //    "Please confirm your account by clicking this link: <a href=\"" + callbackUrl +
            //    "\">link</a>");
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

如果已成功创建用户，用户记录通过调用 `_signInManager.SignInAsync`。

注意：请参阅[帐户确认](#)有关步骤，以防止在注册的即时登录名。

4. 登录。

用户可以通过单击登录链接顶部的站点，或可能的登录页导航它们，如果用户尝试访问要求获得授权的站点的一部分。当用户提交的登录页中上，窗体 `AccountController`Login` 调用操作。

`Login` 操作调用 `PasswordSignInAsync` 上 `_signInManager` 对象（提供给 `AccountController` 通过依赖关系注入）。

```

// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(model.Email,
            model.Password, model.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation(1, "User logged in.");
            return RedirectToLocal(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToAction(nameof(SendCode), new { ReturnUrl = returnUrl, RememberMe =
model.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning(2, "User account locked out.");
            return View("Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return View(model);
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

基于 `Controller` 类会公开 `User` 你可以从控制器方法访问的属性。例如，可以枚举 `User.Claims` 并做出授权决策。有关详细信息，请参阅 [授权](#)。

5. 注销。

单击注销链接调用 `LogOut` 操作。

```

// POST: /Account/LogOut
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LogOut()
{
    await _signInManager.SignOutAsync();
    _logger.LogInformation(4, "User logged out.");
    return RedirectToAction(nameof(HomeController.Index), "Home");
}

```

前面的代码调用上面 `_signInManager.SignOutAsync` 方法。`SignOutAsync` 方法清除在 cookie 中存储的用户的声明。

6. 配置。

标识具有一些可以在应用程序的 startup 类中重写的默认行为。`IdentityOptions` 无需使用的默认行为时配置。

下面的代码设置多个密码强度选项：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity< ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores< ApplicationContext >()
        .AddDefaultTokenProviders();

    services.Configure< IdentityOptions >(options =>
    {
        // Password settings
        options.Password.RequireDigit = true;
        options.Password.RequiredLength = 8;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = true;
        options.Password.RequireLowercase = false;
        options.Password.RequiredUniqueChars = 6;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.AllowedForNewUsers = true;

        // User settings
        options.User.RequireUniqueEmail = true;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
        // If the LoginPath isn't set, ASP.NET Core defaults
        // the path to /Account/Login.
        options.LoginPath = "/Account/Login";
        // If the AccessDeniedPath isn't set, ASP.NET Core defaults
        // the path to /Account/AccessDenied.
        options.AccessDeniedPath = "/Account/AccessDenied";
        options.SlidingExpiration = true;
    });

    // Add application services.
    services.AddTransient< IEmailSender, EmailSender >();

    services.AddMvc();
}
}
```

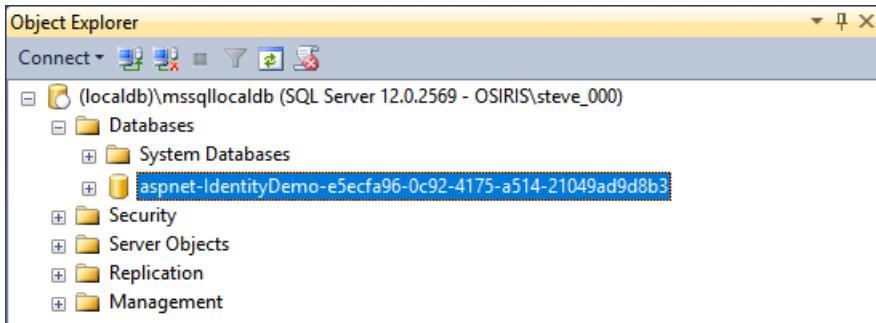
有关如何配置标识的详细信息，请参阅[配置标识](#)。

你还可以配置的数据类型的主键，请参阅[配置标识主键数据类型](#)。

7. 查看数据库。

如果你的应用使用 SQL Server 数据库（默认在 Windows 上以及为 Visual Studio 用户），你可以查看数据库中创建的应用。你可以使用**SQL Server Management Studio**。或者，从 Visual Studio 中，选择视图 > **SQL Server 对象资源管理器**。连接到 **(localdb) \MSSQLLocalDB**。名称匹配的数据库**aspnet-**

<你的项目名称>-<日期字符串> 显示。



展开的数据库并将其表，然后右键单击**dbo。AspNetUsers**表，然后选择查看数据。

8. 确认标识在有效运行

默认值 ASP.NET 核心 Web 应用程序项目模板，用户可以访问应用程序中的任何操作，而无到登录名。若要验证 ASP.NET 标识工作原理，添加 `[Authorize]` 属性设为 `About` 操作 `Home` 控制器。

```
[Authorize]
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";
    return View();
}
```

- [Visual Studio](#)
- [.NET Core CLI](#)

运行项目使用 **Ctrl + F5** 并导航到有关页。仅经过身份验证的用户可以访问有关现在，页，以便 ASP.NET 将你重定向到登录页来登录或注册。

标识组件

标识系统的主引用集是 `Microsoft.AspNetCore.Identity`。此程序包包含 ASP.NET 核心标识接口的核心集，包括 `Microsoft.AspNetCore.Identity.EntityFrameworkCore`。

这些依赖关系需要 ASP.NET Core 应用程序中使用的标识系统：

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore` - 包含要用于实体框架核心标识所需的类型。
- `Microsoft.EntityFrameworkCore.SqlServer` 实体框架的核心是用于类似于 SQL Server 关系数据库的 Microsoft 的推荐使用此数据访问技术。对于测试，你可以使用 `Microsoft.EntityFrameworkCore.InMemory`。
 -
- `Microsoft.AspNetCore.Authentication.Cookies` - 使应用可以使用基于 cookie 的身份验证的中间。

迁移到 ASP.NET 核心标识

有关其他信息和指南迁移你现有的标识存储，请参阅[迁移身份验证和标识](#)。

设置密码强度

请参阅[配置有关设置的最小密码要求的示例](#)。

后续步骤

- [迁移身份验证和标识](#)

- 帐户确认和密码恢复
- 使用 SMS 设置双因素身份验证
- Facebook、Google、和外部提供程序身份验证

配置 ASP.NET 核心标识

2018/5/17 • 8 min to read • [Edit Online](#)

ASP.NET 核心标识使用默认配置设置，例如密码策略、锁定时间和 cookie 设置。应用程序的可重写这些设置 `Startup` 类。

标识选项

`IdentityOptions` 类表示可以用于配置的标识系统的选项。

声明的标识

`IdentityOptions.ClaimsIdentity` 指定 `ClaimsIdentityOptions` 与表中显示的属性。

属性	描述	默认
<code>RoleClaimType</code>	获取或设置用于角色声明的声明类型。	<code>ClaimTypes.Role</code>
<code>SecurityStampClaimType</code>	获取或设置用于安全戳声明的声明类型。	<code>AspNet.Identity.SecurityStamp</code>
<code>UserIdClaimType</code>	获取或设置用于用户标识符声明的声明类型。	<code>ClaimTypes.Nameldentifier</code>
<code>UserNameClaimType</code>	获取或设置用于用户名声明的声明类型。	<code>ClaimTypes.Name</code>

锁定

为一段时间内阻止用户，在给定的失败的访问尝试次数后（默认：5 分钟锁定 5 失败的访问尝试次数后）。成功的身份验证将失败的访问尝试计数重置并重置时钟。

下面的示例显示默认值：

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Lockout settings
    options.Lockout.AllowedForNewUsers = true;
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
})
.AddEntityFrameworkStores<ApplicationContext>()
.AddDefaultTokenProviders();
```

确认 `PasswordSignInAsync` 设置 `lockoutOnFailure` 到 `true`：

```
var result = await _signInManager.PasswordSignInAsync(
    Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);
```

`IdentityOptions.Lockout` 指定 `LockoutOptions` 与表中显示的属性。

属性	描述	默认
AllowedForNewUsers	确定是否新用户会被锁定。	true
DefaultLockoutTimeSpan	时间量锁定用户锁定发生时。	5 分钟
MaxFailedAccessAttempts	失败的访问尝试, 直到用户已被锁定, 如果启用了锁定次数。	5

Password

默认情况下, 标识要求密码包含大写字符、小写字符、数字和非字母数字字符。密码必须至少为六个字符。

`PasswordOptions`可以更改在 `Startup.ConfigureServices`。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

ASP.NET 核心 2.0 增加`RequiredUniqueChars`属性。否则, 选项是 ASP.NET Core 相同 1.x。

```
services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Password settings
    options.Password.RequireDigit = true;
    options.Password.RequiredLength = 8;
    options.Password.RequiredUniqueChars = 2;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
})
.AddEntityFrameworkStores<ApplicationContext>()
.AddDefaultTokenProviders();
```

`IdentityOptions.Password`指定`PasswordOptions`与表中显示的属性。

属性	描述	默认
RequireDigit	需要介于 0-9 中的密码。	true
RequiredLength	密码最小长度。	6
RequiredUniqueChars	仅适用于 ASP.NET 核心 2.0 或更高版本。 需要密码中的非重复字符数。	1
RequireLowercase	需要密码中的小写字符。	true
RequireNonAlphanumeric	需要密码中的非字母数字字符。	true
RequireUppercase	需要密码中的大写字符。	true

登录

```

services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // Signin settings
    options.SignIn.RequireConfirmedEmail = true;
    options.SignIn.RequireConfirmedPhoneNumber = false;
})
.AddEntityFrameworkStores<ApplicationContext>()
.AddDefaultTokenProviders();

```

[IdentityOptions.SignIn](#)指定[SignInOptions](#)与表中显示的属性。

属性	描述	默认
RequireConfirmedEmail	需要登录的确认电子邮件。	false
RequireConfirmedPhoneNumber	需要一个确认的电话号码来登录。	false

标记

[IdentityOptions.Tokens](#)指定[TokenOptions](#)与表中显示的属性。

属性	描述
AuthenticatorTokenProvider	获取或设置 AuthenticatorTokenProvider 用于验证双因素登录使用验证器。
ChangeEmailTokenProvider	获取或设置 ChangeEmailTokenProvider 用于生成电子邮件更改确认电子邮件中使用的令牌。
ChangePhoneNumberTokenProvider	获取或设置 ChangePhoneNumberTokenProvider 用于生成令牌更改电话号码时使用。
EmailConfirmationTokenProvider	获取或设置用于生成在帐户确认电子邮件中使用的令牌的令牌提供程序。
PasswordResetTokenProvider	获取或设置 IUserTwoFactorTokenProvider 用于生成在密码重置电子邮件中使用的令牌。
ProviderMap	用于构造 用户令牌提供程序 的密钥与用作为提供程序的名称。

“用户”

```

services.AddIdentity<ApplicationUser, IdentityRole>(options =>
{
    // User settings
    options.User.RequireUniqueEmail = true;
})
.AddEntityFrameworkStores<ApplicationContext>()
.AddDefaultTokenProviders();

```

[IdentityOptions.User](#)指定[UserOptions](#)与表中显示的属性。

属性	描述	默认
----	----	----

属性	描述	默认
AllowedUserNameCharacters	允许使用的用户名中的字符。	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 .@+
RequireUniqueEmail	要求每个用户具有唯一的电子邮件。	false

Cookie 设置

配置中的应用程序的 cookie `Startup.ConfigureServices` :

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Account/Login";
    // ReturnUrlParameter requires `using Microsoft.AspNetCore.Authentication.Cookies;`
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

`CookieAuthenticationOptions`具有以下属性:

属性	描述
AccessDeniedPath	通知处理程序, 它应更改传出403 禁止访问状态代码转换为302 重定向到给定的路径。 默认值为 <code>/Account/AccessDenied</code> 。
AuthenticationScheme	仅适用于 ASP.NET Core 1.x。 特定的身份验证方案逻辑名称。
AutomaticAuthenticate	仅适用于 ASP.NET Core 1.x。 为 true 时, cookie 身份验证应在每个请求上运行并尝试验证并重新构造它创建的任何序列化的主体。
AutomaticChallenge	仅适用于 ASP.NET Core 1.x。 如果为 true, 则身份验证中间件处理自动挑战。如果为 false, 身份验证中间件仅更改响应时显式由 <code>AuthenticationScheme</code> 。
ClaimsIssuer	获取或设置应将用于创建的任何声明的颁发者 (继承自 <code>AuthenticationSchemeOptions</code>)。
Cookie.Domain	要将与 cookie 相关联的域。

属性	描述
Cookie.Expiration	获取或设置 HTTP cookie (不身份验证 cookie) 的使用期限。通过重写此属性 ExpireTimeSpan 。它不应使用在 CookieAuthentication 的上下文。
Cookie.HttpOnly	指示是否可访问客户端脚本的 cookie。 默认值为 <code>true</code> 。
Cookie.Name	Cookie 的名称。 默认值为 <code>.AspNetCore.Cookies</code> 。
Cookie.Path	Cookie 路径中。
Cookie.SameSite	<code>SameSite</code> 的 cookie 的属性。 默认值是 SameSiteMode.Lax 。
Cookie.SecurePolicy	CookieSecurePolicy 配置。 默认值是 CookieSecurePolicy.SameAsRequest 。
CookieDomain	仅适用于 ASP.NET Core 1.x。 Cookie 提供服务位置的域名。
CookieHttpOnly	仅适用于 ASP.NET Core 1.x。 一个标志，指示该 cookie 应仅服务器访问。 默认值为 <code>true</code> 。
CookiePath	仅适用于 ASP.NET Core 1.x。 用于隔离在相同的主机名上运行的应用程序。
CookieSecure	仅适用于 ASP.NET Core 1.x。 一个标志，指示是否创建的 cookie 应被限制为 HTTPS (<code>CookieSecurePolicy.Always</code>)，HTTP 或 HTTPS (<code>CookieSecurePolicy.None</code>)，或请求的同一个协议 (<code>CookieSecurePolicy.SameAsRequest</code>)。 默认值为 <code>CookieSecurePolicy.SameAsRequest</code> 。
CookieManager	用于从请求中获取 cookie 或将它们设置在响应上的组件。
DataProtectionProvider	如果设置，使用的提供程序通过 CookieAuthenticationHandler 的数据保护。
说明	仅适用于 ASP.NET Core 1.x。 有关提供给应用程序的身份验证类型的其他信息。

属性	描述
事件	该处理程序的提供程序以便为应用程序控件提供在处理发生的位置中的某些点上调用方法。
EventsType	如果设置, 该服务获取类型 <code>Events</code> 实例而不是属性 (继承自 <code>AuthenticationSchemeOptions</code>)。
ExpireTimeSpan	控件用多少时间有效从创建它时保留了 cookie 中存储身份验证票证。 默认值为 14 天。
LoginPath	未授权用户时, 则在重定向到登录到此路径。 默认值为 <code>/Account/Login</code> 。
LogoutPath	当用户已注销时, 则在重定向到此路径。 默认值为 <code>/Account/Logout</code> 。
ReturnUrlParameter	确定附加的中间件的查询字符串参数的名称时 401 未授权状态代码更改为 302 重定向到登录名路径。 默认值为 <code>ReturnUrl</code> 。
SessionStore	要在其中存储跨请求的标识可选容器。
SlidingExpiration	为 true 时, 使用当前 cookie 的多个中途过期窗口新过期时间被颁发一个新的 cookie。 默认值为 <code>true</code> 。
TicketDataFormat	<code>TicketDataFormat</code> 用于保护和取消保护标识和其他属性存储在 cookie 值。

在 ASP.NET 核心中配置 Windows 身份验证

2018/5/17 • 5 min to read • [Edit Online](#)

作者: [Steve Smith](#) 和 [Scott Addie](#)

Windows 身份验证可以配置用于与 IIS 托管的 ASP.NET Core 应用[HTTP.sys](#), 或[WebListener](#)。

什么是 Windows 身份验证？

Windows 身份验证依赖于操作系统的 ASP.NET Core 应用的用户进行身份验证。在使用 Active Directory 域标识或其他 Windows 帐户来标识用户的企业网络上运行你的服务器时，你可以使用 Windows 身份验证。

Windows 身份验证是最适合 intranet 环境中的用户、客户端应用程序和 web 服务器属于同一 Windows 域。

[了解有关 Windows 身份验证和为 IIS 安装。](#)

启用 ASP.NET Core 应用程序中的 Windows 身份验证

Visual Studio Web 应用程序模板可以配置为支持 Windows 身份验证。

使用 Windows 身份验证应用程序模板

在 Visual Studio 中：

1. 创建新的 ASP.NET Core Web 应用程序。
2. 从模板列表中选择 Web 应用程序。
3. 选择**更改身份验证**按钮，然后选择**Windows 身份验证**。

运行应用。用户名已显示在顶部的应用程序的权限。

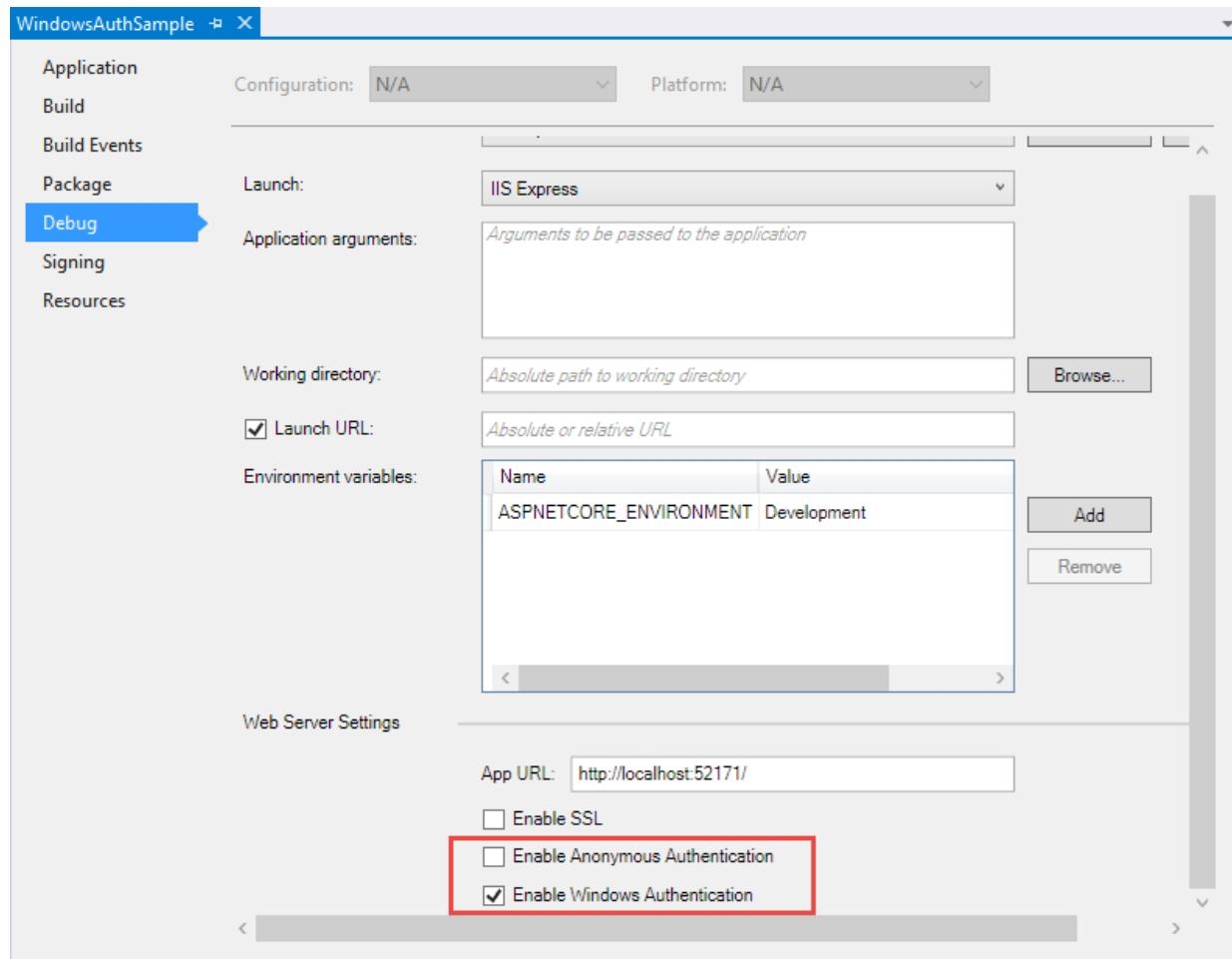
The screenshot shows a Microsoft Edge browser window displaying the home page of a Web Application named "WebApplication6". The URL in the address bar is "localhost:48794". The page title is "WebApplication6". The top navigation bar includes a "Guest" button and links for "Home", "About", and "Contact". A red box highlights the "Hello, OSIRIS\steve_000!" greeting message in the top right corner. Below the header, there's a green section with text about packages and a "Learn More" button. The main content area is titled "Application uses" and lists three items: "Sample pages using ASP.NET Core MVC", "Bower for managing client-side libraries", and "Theming using Bootstrap". A sidebar on the right is titled "How to".

对于使用 IIS Express 的开发工作，该模板提供所有使用 Windows 身份验证所需的配置。以下部分说明如何手

动配置 Windows 身份验证的 ASP.NET Core 应用。

为 Windows 以及匿名身份验证的 visual Studio 设置

Visual Studio 项目属性页面的调试选项卡为 Windows 身份验证和匿名身份验证提供的复选框。



或者，在中配置这两个属性`launchSettings.json`文件：

```
{  
  "iisSettings": {  
    "windowsAuthentication": true,  
    "anonymousAuthentication": false,  
    "iisExpress": {  
      "applicationUrl": "http://localhost:52171/",  
      "sslPort": 0  
    }  
  } // additional options trimmed  
}
```

启用与 IIS 的 Windows 身份验证

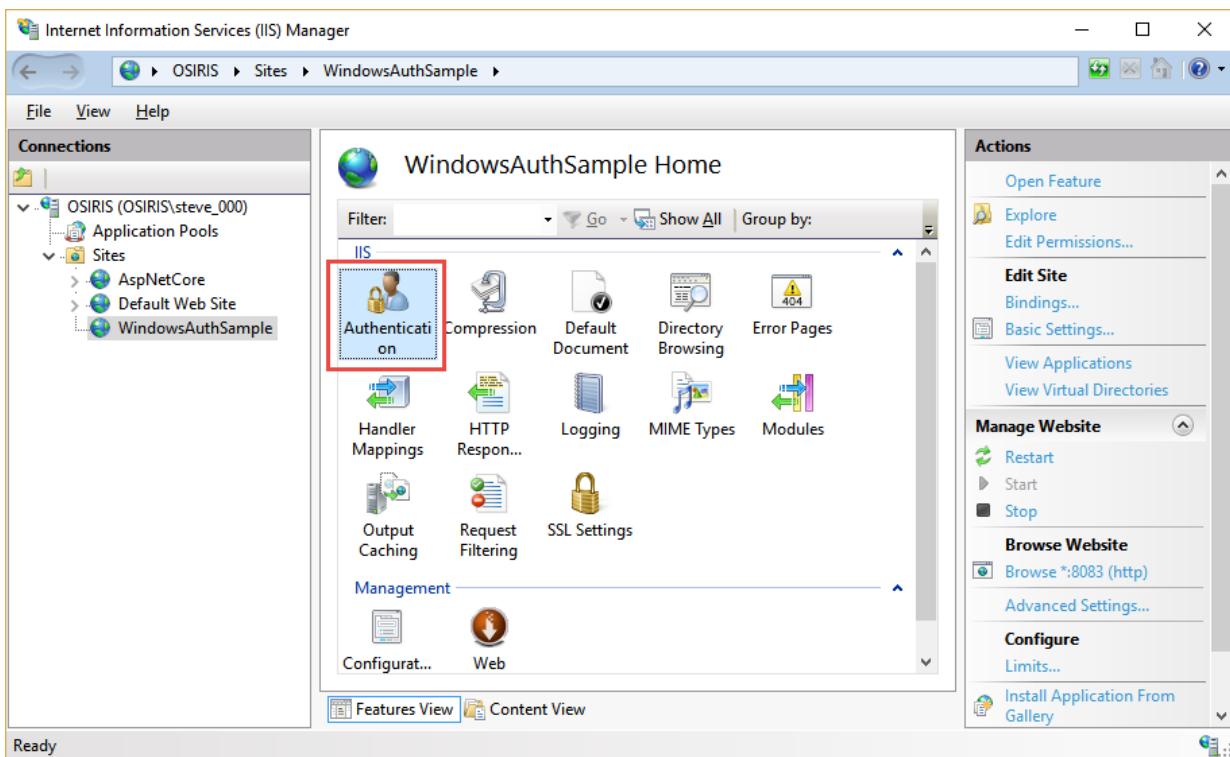
IIS 使用[ASP.NET 核心模块](#)到承载 ASP.NET Core 应用程序。模块流 Windows 身份验证到 IIS 默认情况下。在 IIS 中，不应用配置了 Windows 身份验证。以下部分说明如何使用 IIS 管理器来配置 ASP.NET Core 应用以使用 Windows 身份验证。

创建一个新的 IIS 站点

指定名称和文件夹，并允许它创建新的应用程序池。

自定义身份验证

打开身份验证菜单上的站点。



禁用匿名身份验证并启用 Windows 身份验证。

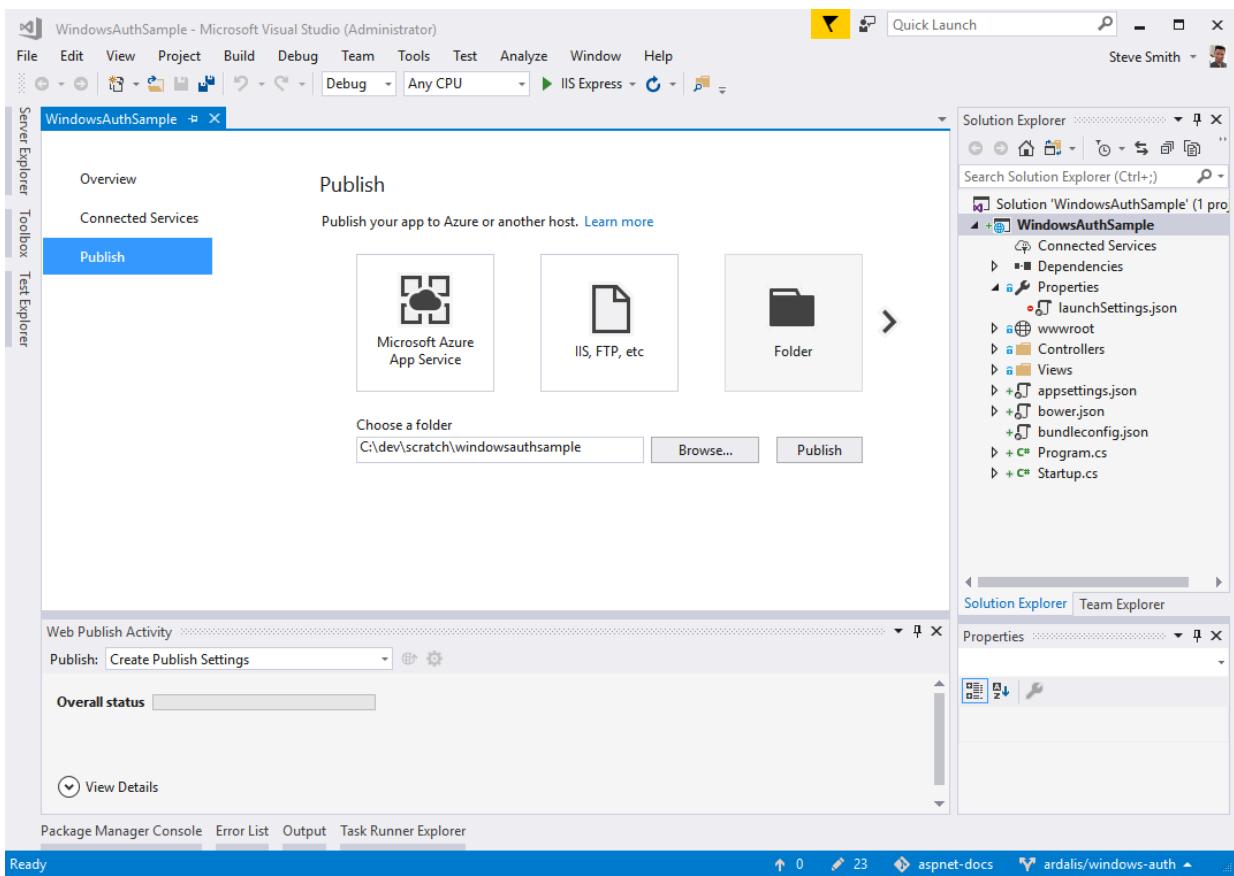
The screenshot shows the 'Authentication' configuration page for the 'WindowsAuthSample' site. A table lists three authentication methods:

Name	Status	Response Type
Anonymous Authentication	Disabled	
ASP.NET Impersonation	Disabled	
Windows Authentication	Enabled	HTTP 401 Challenge

The Actions pane on the right includes 'Enable' and 'Edit...' buttons, as well as a 'Help' link. The status bar at the bottom indicates the configuration file path: 'Configuration: 'localhost' applicationHost.config , <location path="WindowsAuthSample">'.

将你的项目发布到 IIS 站点文件夹

使用 Visual Studio 或.NET 核心 CLI, 将应用发布到目标文件夹。



详细了解[发布到 IIS](#)。

启动应用程序来验证使用 Windows 身份验证。

启用 Windows 身份验证与 HTTP.sys 或 WebListener

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

尽管 Kestrel 不支持 Windows 身份验证, 你可以使用[HTTP.sys](#)以在 Windows 上支持自承载的方案。下面的示例将配置应用程序的 web 主机 HTTP.sys 用于 Windows 身份验证:

```
public class Program
{
    public static void Main(string[] args) =>
        BuildWebHost(args).Run();

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseHttpSys(options =>
            {
                options.Authentication.Schemes =
                    AuthenticationSchemes.NTLM | AuthenticationSchemes.Negotiate;
                options.Authentication.AllowAnonymous = false;
            })
            .Build();
}
```

使用 Windows 身份验证

匿名访问的配置状态确定的方式 `[Authorize]` 和 `[AllowAnonymous]` 特性用于应用程序中。以下两节介绍如何处理的匿名访问权限不允许的和允许配置状态。

不允许匿名访问

当启用了 Windows 身份验证并且禁用了匿名访问, `[Authorize]` 和 `[AllowAnonymous]` 属性产生任何影响。如果 IIS 站点 (或 HTTP.sys 或 WebListener 服务器) 配置为不允许匿名访问, 请求将永远不会到达你的应用程序。为此, `[AllowAnonymous]` 属性不适用。

允许匿名访问

当启用了 Windows 身份验证和匿名访问时, 使用 `[Authorize]` 和 `[AllowAnonymous]` 属性。`[Authorize]` 属性允许你保护的应用程序的部分, 真正需要 Windows 身份验证。`[AllowAnonymous]` 特性重写 `[Authorize]` 属性中允许匿名访问的应用程序的使用情况。请参阅[简单授权](#)对于特性用法的详细信息。

在 ASP.NET Core 2.x, `[Authorize]` 属性要求中的其他配置 `Startup.cs` 质询对于 Windows 身份验证的匿名请求。建议的配置略有根据正在使用的 web 服务器而有所不同。

注意

默认情况下, 缺少授权可访问的页面的用户出现了空的 HTTP 403 响应。[StatusCodePages 中间件](#)可以配置为用户提供更好的“拒绝访问”体验。

IIS

如果使用 IIS, 添加到以下 `ConfigureServices` 方法:

```
// IISDefaults requires the following import:  
// using Microsoft.AspNetCore.Server.IISIntegration;  
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

HTTP.sys

如果使用 HTTP.sys, 添加到以下 `configureServices` 方法:

```
// HttpSysDefaults requires the following import:  
// using Microsoft.AspNetCore.Server.HttpSys;  
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

Impersonation

ASP.NET 核心未实现模拟。使用应用程序池或进程标识的所有请求的应用程序标识使用运行应用。如果你需要显式执行代表用户操作, 使用 `WindowsIdentity.RunImpersonated`。在此上下文中运行的单个操作, 然后关闭上下文。

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState: {user.ImpersonationLevel}\n");

        WindowsIdentity.RunImpersonated(user.AccessToken, () =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\tState: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            context.Response.Body.Write(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

请注意，`RunImpersonated` 不支持异步操作，并且不应可用于复杂的方案。例如，包装整个请求或中间件链不受支持或建议。

在 ASP.NET 核心中配置标识主键数据类型

2018/5/17 • 2 min to read • [Edit Online](#)

ASP.NET 核心标识可以配置用于表示为主键的数据类型。标识使用 `string` 默认的数据类型。你可以重写此行为。

自定义的主键数据类型

1. 创建的自定义实现 `IdentityUser` 类。它表示要用于创建用户对象的类型。在下面的示例中，默认值 `string` 类型将替换 `Guid`。

```
namespace webapptemplate.Models
{
    // Add profile data for application users by adding properties to the ApplicationUser class
    public class ApplicationUser : IdentityUser<Guid>
    {
    }
}
```

2. 创建的自定义实现 `IdentityRole` 类。它表示要用于创建角色对象的类型。在下面的示例中，默认值 `string` 类型将替换 `Guid`。

```
namespace webapptemplate.Models
{
    public class ApplicationRole : IdentityRole<Guid>
    {
    }
}
```

3. 创建一个自定义数据库上下文类。它继承自用于标识的实体框架数据库上下文类。`TUser` 和 `TRole` 自变量引用分别在前一步骤中创建的自定义用户和角色类。`Guid` 为主键定义的数据类型。

```
namespace webapptemplate.Data
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Identity model and override the defaults if needed.
            // For example, you can rename the ASP.NET Identity table names and more.
            // Add your customizations after calling base.OnModelCreating(builder);
        }
    }
}
```

4. 在应用程序的启动类添加身份验证服务时，请注册自定义数据库上下文类。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

`AddEntityFrameworkStores` 方法不接受 `TKey` 自变量，因为它未在 ASP.NET Core 1.x。主键的数据类型推断通过分析 `DbContext` 对象。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    // Add application services.
    services.AddTransient<IEmailSender, EmailSender>();

    services.AddMvc();
}
```

测试更改

在完成配置更改，表示为主键的属性将反映新的数据类型。下面的示例演示如何访问一个 MVC 控制器中的属性。

```
[HttpGet]
[AllowAnonymous]
public async Task<Guid> GetCurrentUserId()
{
    ApplicationUser user = await _userManager.GetUserAsync(HttpContext.User);
    return user.Id; // No need to cast here because user.Id is already a Guid, and not a string
}
```

ASP.NET 核心标识的自定义的存储提供程序

2018/5/14 • 11 min to read • [Edit Online](#)

作者: [Steve Smith](#)

ASP.NET 核心标识是一种可扩展系统，可用于创建自定义存储提供程序并将其连接到你的应用。本主题介绍如何创建 ASP.NET 核心标识的自定义的存储提供程序。它介绍如何创建你自己的存储提供程序的重要概念，但不的分步演练。

[查看或下载 GitHub 中的示例。](#)

介绍

默认情况下，ASP.NET 核心标识系统中使用实体框架核心的 SQL Server 数据库存储用户信息。对于许多应用程序，此方法也适用。但是，你可能希望使用不同的持久性机制或数据架构。例如：

- 你使用[Azure 表存储](#)或其他数据存储。
- 数据库表具有不同的结构。
- 你可能想要使用不同的数据访问方法，如[Dapper](#)。

在每个这种情况下，你可以为你的存储机制编写自定义提供程序并将该提供程序插入您的应用程序。

ASP.NET 核心标识包含在 Visual Studio 中使用“单个用户帐户”选项的项目模板中。

在使用.NET 核心 CLI 时，添加 `-au Individual`：

```
dotnet new mvc -au Individual  
dotnet new webapi -au Individual
```

ASP.NET 核心标识体系结构

ASP.NET 核心标识包含类称为管理器和存储区。[管理器](#)是高级类应用程序开发人员用于执行操作，如创建标识用户。[存储](#)是用于指定如何保持实体，如用户和角色，较低级别类。存储按照[存储库模式](#)和紧密耦合持久性的机制。管理器是相互脱耦存储区，这意味着你可以将持久性机制，而无需更改应用程序代码（除外配置）。

下图显示的 web 应用程序的交互方式具有管理器中，当与数据访问层的存储区交互时。



Identity Manager

Example: `UserManager`, `RoleManager`

Identity Store

Example: `UserStore`, `RoleStore`

Data Access
Layer

Data Source

Example: SQL Server Database, Azure Table Storage

若要创建自定义的存储提供程序，请使用此数据访问层（绿色和灰色框在上图中）创建数据源、数据访问层和交互的存储类。你不需要自定义管理器或与它们（蓝色上面的框）进行交互的应用程序代码。

创建的新实例时 `UserManager` 或 `RoleManager` 提供用户类的类型和作为参数传递存储类的实例。此方法使您可以插入到 ASP.NET 核心的自定义的类。

[重新配置应用程序以使用新存储提供程序](#) 演示如何实例化 `UserManager` 和 `RoleManager` 使用自定义存储。

ASP.NET 核心标识将存储的数据类型

[ASP.NET 核心标识](#) 数据类型进行详细介绍下列各节：

用户

注册用户的网站的用户。`IdentityUser` 可能扩展类型，或将其用作自定义类型的示例。你不必继承来实现自定义标识的存储解决方案的特定类型。

用户声明

一组语句（或[声明](#)）有关的用户的表示用户的标识。可以启用的用户的标识不是可以通过角色实现更大的表达式。

用户登录名

有关外部身份验证提供程序（如 Facebook 或 Microsoft 帐户）的信息在用户日志记录时使用。[示例](#)

角色

你的站点的授权组。包括（如“Admin”或“Employee”）的角色 Id 和角色名称。[示例](#)

数据访问层

本主题假定你熟悉要使用的持久性机制以及如何创建该机制的实体。本主题不提供有关如何创建存储库或数据访问类；详细信息使用 ASP.NET 核心标识时，它提供了有关设计决策的一些建议。

为自定义的存储提供程序设计的数据访问层时，你有大量的自由。只需创建想要使用它在你的应用程序的功能的持久性机制。例如，如果不使用你的应用程序中的角色，你不必创建角色或用户角色关联的存储。你的技术和现有基础结构可能需要的 ASP.NET 核心标识的默认实现不同的结构。你的数据访问层，可以提供要使用的存储实现结构的逻辑。

数据访问层提供逻辑来将数据从 ASP.NET 核心标识保存到数据源。自定义的存储提供程序的数据访问层可能包括以下类用于存储用户和角色信息。

Context 类

封装用于连接到持久性机制和执行查询的信息。多个数据类需要通过依赖关系注入通常提供此类的实例。[示例](#)。

用户存储

存储和检索用户信息（如用户名和密码哈希）。[示例](#)

角色存储

存储和检索角色信息（如角色名称中）。[示例](#)

UserClaims 存储

存储和检索用户声明信息（如的声明类型和值）。[示例](#)

UserLogins 存储

存储和检索用户登录信息（例如外部身份验证提供程序）。[示例](#)

UserRole 存储

存储和检索哪些角色分配给哪些用户。[示例](#)

提示：仅实现你要在应用程序中使用的类。

在数据访问类中，提供的代码来执行持久性机制的数据操作。例如，在自定义提供程序，你可能具有以下代码以创建中的新用户存储类：

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

实现逻辑用于创建用户处于 `_usersTable.CreateAsync` 方法，如下所示。

自定义用户类

时实现存储提供程序，创建一个用户类，这等效于 `IdentityUser` 类。

您的用户类必须包含至少 `Id` 和 `UserName` 属性。

`IdentityUser` 类定义的属性，`UserManager` 调用时执行请求的操作。默认类型 `Id` 属性是一个字符串，但可以继承 `IdentityUser< TKey, TUserClaim, TUserRole, TUserLogin, TUserToken >` 并指定不同的类型。框架需要处理的数据类型转换的存储实现。

自定义用户存储区

创建 `UserStore` 提供用户的所有数据操作的方法的类。此类是等效于 `UserStore` 类。在你 `UserStore` 类中，实现 `IUserStore< TUser >` 和所需的可选接口。你选择的可选接口，以实现基于你的应用程序中提供的功能。

可选接口

- `IUserRoleStore` /dotnet/api/microsoft.aspnetcore.identity.iuserrolestore-1
- `IUserClaimStore` /dotnet/api/microsoft.aspnetcore.identity.iuserclaimstore-1
- `IUserPasswordStore` /dotnet/api/microsoft.aspnetcore.identity.iuserpasswordstore-1
- `IUserSecurityStampStore`
- `IUserEmailStore`
- `IPhoneNumberStore`
- `IQueryableUserStore`
- `IUserLoginStore`
- `IUserTwoFactorStore`
- `IUserLockoutStore`

可选接口继承自 `IUserStore`。你可以看到存储部分实现的示例用户 [此处](#)。

在 `UserStore` 类，你将使用你创建用于执行操作的数据访问类。它们将传递中使用依赖关系注入。例如，在 SQL Server 使用 Dapper 实现，`UserStore` 类具有 `CreateAsync` 使用的实例方法 `DapperUsersTable` 以插入新记录：

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id, user.Email, user.EmailConfirmed,
        user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could not insert user {user.Email}." });
}
```

要实现自定义用户存储时的接口

- **`IUserStore`**

`IUserStore<热熔器>` 接口是在用户存储中必须实现的唯一接口。它定义了用于创建、更新、删除和检索用户的方法。

- **`IUserClaimStore`**

`IUserClaimStore<热熔器>` 接口定义实现以启用用户声明的方法。它包含用于添加、删除和检索用户声明的方法。

- **`IUserLoginStore`**

`IUserLoginStore<热熔器>` 定义实现以启用外部身份验证提供程序的方法。它包含用于添加、删除和检索用户登录名和用于检索用户的登录信息基于方法的方法。

- **`IUserRoleStore`**

`IUserRoleStore<热熔器>` 接口定义实现的用户映射到角色的方法。它包含要添加、删除和检索用户的角色和方法来检查是否将用户分配到角色的方法。

- **`IUserPasswordStore`**

`IUserPasswordStore<热熔器>` 接口定义实现以保持经过哈希处理的密码的方法。它包含用于获取和设置工作经过哈希处理的密码，以及用于指示用户是否已设置密码的方法。

- **`IUserSecurityStampStore`**

`IUserSecurityStampStore<热熔器>` 接口定义的方法实现用于安全戳，该值指示是否已更改用户的帐户信息。当用户更改密码，或添加或删除登录名，将更新此 stamp。它包含用于获取和设置安全戳的方法。

- **`IUserTwoFactorStore`**

`IUserTwoFactorStore<热熔器>` 接口定义实现以支持两个因素身份验证的方法。它包含用于获取和设置是否为

用户启用双重身份验证的方法。

- **IUserPhoneNumberStore**

[IUserPhoneNumberStore<热熔器>](#) 接口定义实现以存储用户电话号码的方法。它包含用于获取和设置的电话号码和是否确认的电话号码的方法。

- **IUserEmailStore**

[IUserEmailStore<热熔器>](#) 接口定义实现以存储用户电子邮件地址的方法。它包含用于获取和设置的电子邮件地址和是否确认电子邮件的方法。

- **IUserLockoutStore**

[IUserLockoutStore<热熔器>](#) 接口定义为存储帐户的锁定信息而实现的方法。它包含用于跟踪未成功的访问和锁定方法。

- **IQueryableUserStore**

[IQueryableUserStore<热熔器>](#) 接口定义成员实现此方法可以提供可查询的用户存储区。

应用程序中实现所需的接口。例如：

```
public class UserStore : IUserStore<IdentityUser>,
    IUserClaimStore<IdentityUser>,
    IUserLoginStore<IdentityUser>,
    IUserRoleStore<IdentityUser>,
    IPasswordStore<IdentityUser>,
    IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim、IdentityUserLogin 和 IdentityUserRole

`Microsoft.AspNet.Identity.EntityFramework` 命名空间包含的实现 [IdentityUserClaim](#)、[IdentityUserLogin](#)、[IdentityUserRole](#) 类。如果你正使用这些功能，你可能想要创建你自己版本的这些类并定义你的应用程序的属性。但是，有时它会更加高效，若要执行基本操作（如添加或删除用户的声明）时不将这些实体加载到内存。相反后，端存储类可以执行这些操作直接在数据源上。例如，`UserStore.GetClaimsAsync` 方法可以调用 `userClaimTable.FindByUserId(user.Id)` 方法上执行查询，它直接表并返回的声明列表。

自定义角色类

在实现角色存储提供程序时，你可以创建自定义角色类型。它不需要实现特定接口，但是它必须具有 `Id`，其中包含通常 `Name` 属性。

以下是一个示例角色类：

```
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

自定义角色存储

你可以创建 `RoleStore` 提供在角色上的所有数据操作的方法的类。此类是等效于 `RoleStore` 类。在 `RoleStore` 类时，实现 `IRoleStore<TRole>` 和（可选）`IQueryableRoleStore<TRole>` 接口。

- **IRoleStore<TRole>**

[IRoleStore](#)接口定义角色存储类中实现的方法。它包含用于创建、更新、删除和检索角色的方法。

- **RoleStore<TRole>**

若要自定义 `RoleStore`，创建一个类以实现 `IRoleStore` 接口。

重新配置应用程序以使用新存储提供程序

一旦你已实现的存储提供程序，你将配置您的应用程序使用它。如果你的应用程序使用默认的提供程序，可将其替换你自定义提供程序上。

1. 删除 `Microsoft.AspNetCore.Identity` NuGet 包。
2. 如果存储提供程序驻留在单独的项目或包，添加对它的引用。
3. 将对所有引用 `Microsoft.AspNetCore.Identity` 使用的命名空间的存储提供程序的语句。
4. 在 `ConfigureServices` 方法，更改 `AddIdentity` 要使用自定义类型的方法。你可以创建你自己的扩展方法实现此目的。请参阅[IdentityServiceCollectionExtensions](#)有关示例。
5. 如果你使用的角色，更新 `RoleManager` 用于你 `RoleStore` 类。
6. 到你的应用的配置更新的连接字符串和凭据。

示例：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add identity types
    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddDefaultTokenProviders();

    // Identity Services
    services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>();
    services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>();
    string connectionString = Configuration.GetConnectionString("DefaultConnection");
    services.AddTransient<SqlConnection>(e => new SqlConnection(connectionString));
    services.AddTransient<DapperUsersTable>();

    // additional configuration
}
```

参考资料

- [ASP.NET 标识的自定义的存储提供程序](#)
- [ASP.NET 核心标识](#)-此存储库包含指向社区维护存储提供程序。

ASP.NET Core 中的 Facebook、Google 和外部提供程序身份验证

2018/4/10 • 3 min to read • [Edit Online](#)

作者: Valeriy Novytskyy 和 Rick Anderson

本教程演示如何生成 ASP.NET Core 2.x 应用，该应用可让用户使用外部身份验证提供程序提供的凭据通过 OAuth 2.0 登录。

以下几节中介绍了 Facebook、Twitter、Google 和 Microsoft 提供程序。第三方程序包中提供了其他提供程序，例如 `AspNet.Security.OAuth.Providers` 和 `AspNet.Security.OpenId.Providers`。

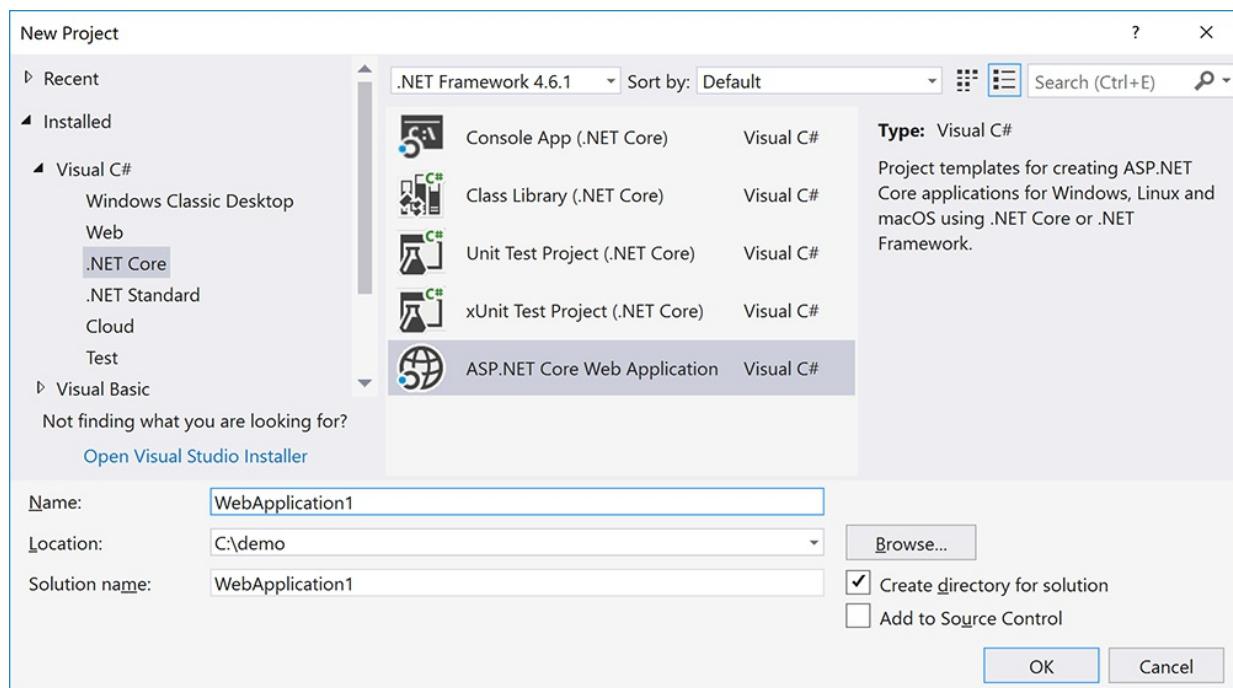


使用户能够使用其当前凭据登录对用户来说十分便利，并且这样做可以将管理登录进程许多复杂操作转移给第三方。有关社交登录如何驱动流量和客户转换的示例，请参阅 [Facebook](#) 和 [Twitter](#) 的案例分析。

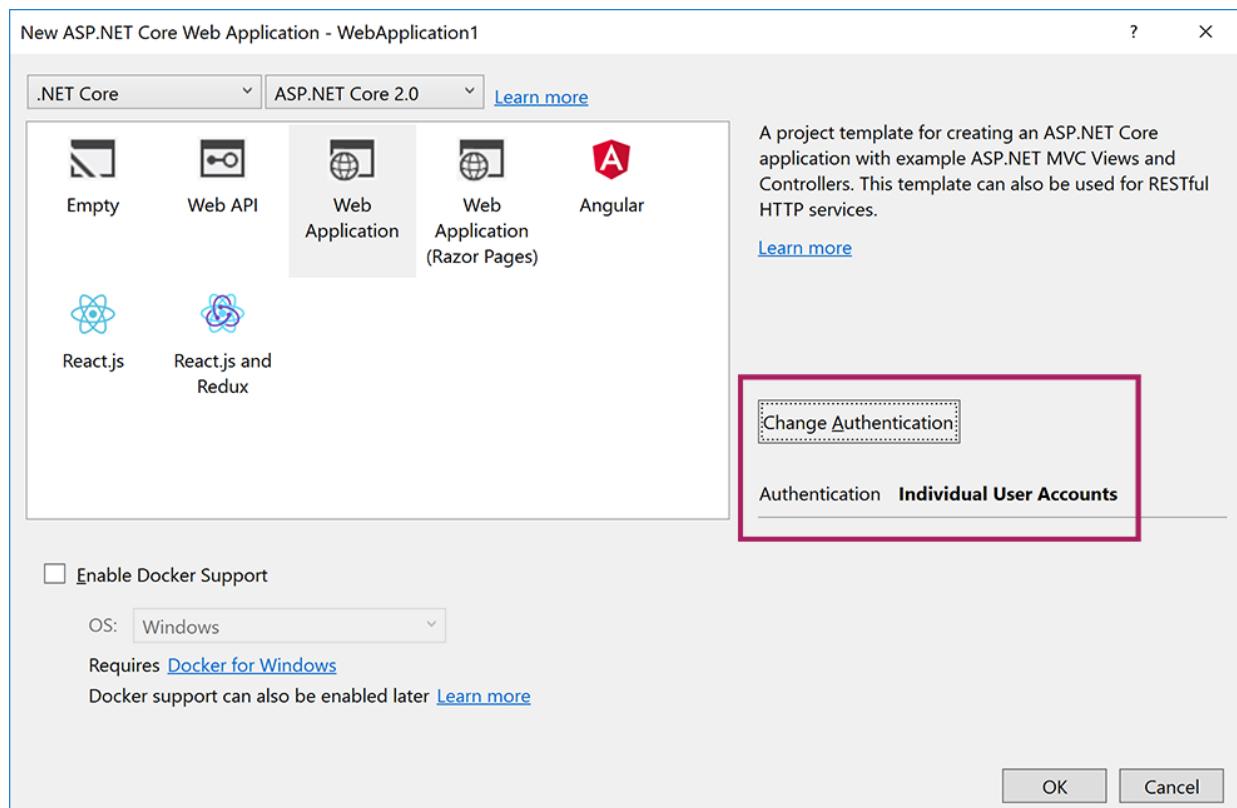
请注意：此处展示的程序包提取了 OAuth 身份验证流的大量复杂操作，但进行故障排除时，可能需要了解相关信息。有许多资源可用，例如，请参阅 [Introduction to OAuth 2](#) (OAuth 2 简介) 或 [Understanding OAuth 2](#) (了解 OAuth 2)。某些问题可通过查看 [ASP.NET Core source code for the provider packages](#) (提供程序包的 ASP.NET Core 源代码) 解决。

创建新的 ASP.NET Core 项目

- 在 Visual Studio 2017 中，从“开始”页创建新项目，或通过“文件”>“新建”>“项目”进行创建。
- 选择“Visual C#”>“.NET Core”分类中提供的“ASP.NET Core Web 应用程序”模板：



- 点击“Web 应用程序”，验证“身份验证”是否设置为“单个用户帐户”：



请注意：本教程适用于可从向导顶部选择的 ASP.NET Core 2.0 SDK 版本。

应用迁移

- 运行应用并选择“登录”链接。
- 选择“以新用户身份注册”链接。
- 输入新帐户的电子邮件地址和密码，再选择“注册”。
- 按照说明操作来应用迁移。

要求 SSL

OAuth 2.0 需要使用 SSL 通过 HTTPS 协议进行身份验证。

请注意：如果如上图所示，在项目向导的“更改身份验证”对话框上选择“单个用户帐户”选项，使用 ASP.NET Core 2.x 的“Web 应用程序”或“Web API”项目模板创建的项目自动配置为启用 SSL 并通过 http URL 启动。

- 按照在 [ASP.NET Core 应用中强制实施 SSL](#) 主题中的步骤进行操作，要求在站点上使用 SSL。

使用 SecretManager 存储登录提供程序分配的令牌

社交登录提供程序在注册过程中分配“应用程序 ID”和“应用程序机密”（确切命名因提供程序而异）。

这些值实际上是应用程序用于访问其 API 的“用户名”和“密码”，它们组成的“机密”可在“机密管理器”的帮助下链接到应用程序配置，而不是直接存储在配置文件中或被硬编码。

按照在 [ASP.NET Core 中进行开发期间安全存储应用机密](#) 主题中的步骤进行操作，以便可以存储以下每个登录提供程序分配的令牌。

应用程序所需的安装登录提供程序

使用以下主题配置应用程序，以使用相应的提供程序：

- [Facebook 相关说明](#)
- [Twitter 相关说明](#)

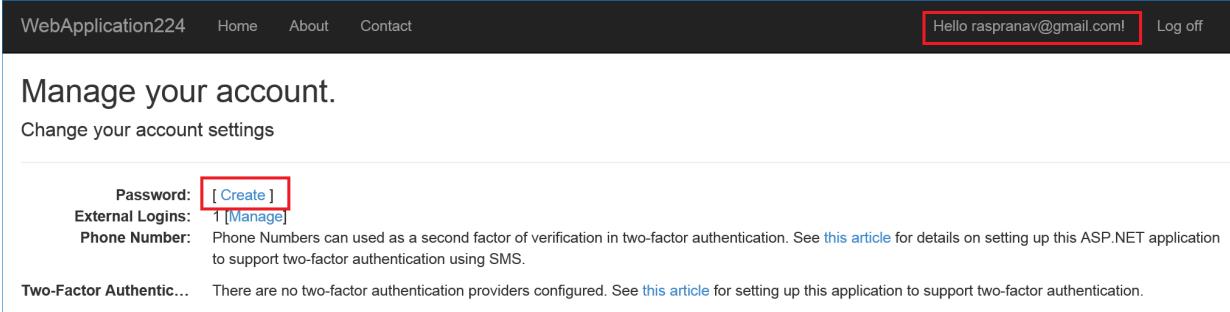
- [Google](#) 相关说明
- [Microsoft](#) 相关说明
- [其他提供程序](#)相关说明

选择性地设置密码

使用外部登录提供程序注册，即表明还没有向应用注册密码。这可让用户无需创建和记住站点密码，但也会使用户依赖外部登录提供程序。如果外部登录提供程序不可用，则无法登录网站。

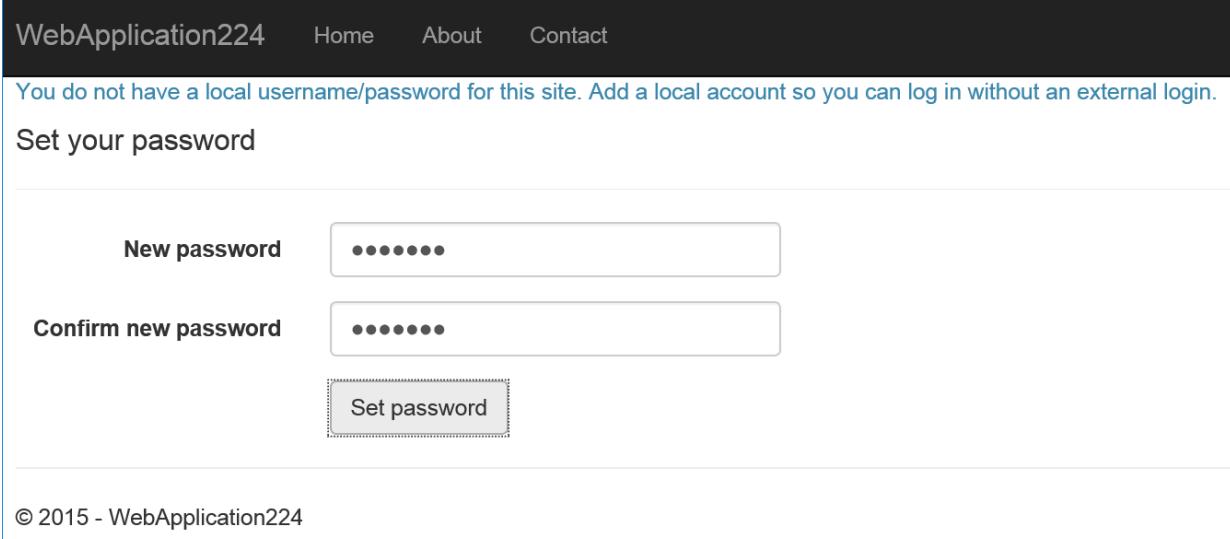
使用外部提供程序在登录过程中设置的电子邮箱创建密码和登录：

- 点击右上角的“Hello”链接，导航到“管理”视图。



The screenshot shows a web application interface titled "WebApplication224". At the top right, there is a red box around the "Hello raspranav@gmail.com!" text and the "Log off" button. Below the header, the page title is "Manage your account." and the sub-section is "Change your account settings". Under the "Password" section, there is a link "[Create]" which is also highlighted with a red box. Other sections include "External Logins" (1 [Manage]), "Phone Number" (with a note about two-factor authentication), and "Two-Factor Authentic..." (with a note about no providers configured).

- 点击“创建”



The screenshot shows a "Set your password" page. At the top, it says "You do not have a local username/password for this site. Add a local account so you can log in without an external login.". Below that, there are two input fields: "New password" and "Confirm new password", both containing six dots to represent masked text. A "Set password" button is located below these fields. At the bottom left, there is a copyright notice: "© 2015 - WebApplication224".

- 设置一个有效密码，可以用此密码和邮箱登录。

后续步骤

- 本文介绍了外部身份验证，并说明了向 ASP.NET Core 应用添加外部登录所需的先决条件。
- 引用特定于提供程序的页，为应用所需的提供程序配置登录。

在 ASP.NET Core Facebook 外部登录安装程序

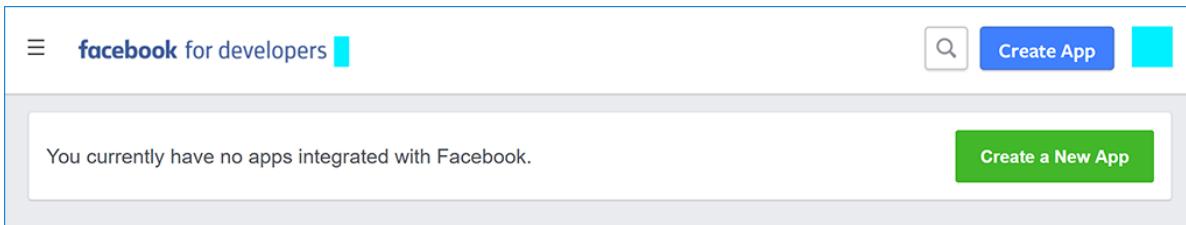
2018/5/17 • 4 min to read • [Edit Online](#)

作者: [Valeriy Novytskyy](#) 和 [Rick Anderson](#)

本教程演示如何使用户可以使用示例 ASP.NET 核心 2.0 项目上创建其 Facebook 帐户登录[上一页](#)。Facebook 身份验证需要[Microsoft.AspNetCore.Authentication.Facebook NuGet 包](#)。我们首先按照创建 Facebook 应用程序 ID[官方步骤](#)。

在 Facebook 中创建应用程序

- 导航到[Facebook 开发人员应用](#)页上，并登录。如果你还没有的 Facebook 帐户，[使用注册 Facebook](#)创建一个的登录页上的链接。
- 点击[添加新的应用程序](#)创建一个新的应用程序 id。右上角的按钮



- 填写表单，然后点击[创建应用程序 ID](#)按钮。

The screenshot shows a 'Create a New App ID' form. It includes fields for 'Display Name' (with placeholder 'The name you want to associate with this App ID') and 'Contact Email' (with placeholder 'Used for important communication about your app'). At the bottom, there's a link 'By proceeding, you agree to the [Facebook Platform Policies](#)' and two buttons: 'Cancel' and 'Create App ID'.

- 上选择一个产品页上，单击[Set Up](#)上[Facebook 登录名](#)卡。

The screenshot shows the SocialLogins dashboard. On the left sidebar, under the 'PRODUCTS' section, the 'Facebook Login' option is selected, indicated by a dark grey background. Other options like 'Account Kit', 'Settings', 'Quickstart', and '+ Add Product' are visible but not selected. The main content area has a heading 'Select a product' and two cards: 'Account Kit' (blue icon with a user symbol) and 'Facebook Login' (blue icon with a cursor symbol). Below each card is a brief description and two buttons: 'Read Docs' and 'Set Up'.

SocialLogins ▾

APP ID: 1477249092352739 | View Analytics

Tools & Support Docs

Dashboard

Settings

Roles

Alerts

App Review

PRODUCTS

+ Add Product

Select a product

Account Kit

Seamless account creation. No more passwords.

Facebook Login

The world's number one social login product.

Read Docs Set Up

- 快速入门向导将启动与选择一个平台作为第一页。现在跳过该向导，通过单击设置左侧菜单中的链接：

The screenshot shows the 'Facebook Login' settings page. The left sidebar has a dark grey background with the following options: 'PRODUCTS' (highlighted), 'Facebook Login' (selected, shown in white text), 'Settings' (in light blue), 'Quickstart', and '+ Add Product'. The main content area is currently empty.

PRODUCTS

Facebook Login

Settings

Quickstart

+ Add Product

- 将向你提供客户端的 OAuth 设置页：

Client OAuth Settings

Yes

Client OAuth Login

Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URIs are allowed with the options below. Disable globally if not used. [?]

Yes

Web OAuth Login

Enables web based OAuth client login for building custom login flows. [?]

No

Force Web OAuth Reauthentication

When on, prompts people to enter their Facebook password in order to log in on the web. [?]

No

Embedded Browser OAuth Login

Enables browser control redirect uri for OAuth client login. [?]

Valid OAuth redirect URIs

https://localhost:44320/signin-facebook

No

Login from Devices

Enables the OAuth client login flow for devices like a smart TV [?]

[Deauthorize](#)

[Discard](#)

[Save Changes](#)

- 输入你的开发 URI 与 /signin-facebook 追加到有效的 OAuth 重定向 Uri 字段 (例如:

https://localhost:44320/signin-facebook)。本教程中稍后配置的 Facebook 身份验证将自动处理请求在 /signin-facebook 要实现的 OAuth 流路由。

- 单击保存更改。
- 单击仪表板左侧导航区域中的链接。

在此页上, 记下你 App ID 和你 App Secret。你将添加到 ASP.NET 核心应用程序下一节中:

SocialLogins

This app is in development mode and can only be used by app admins, developers and testers [?]

API Version [\[?\]](#)

App ID

v2.10

1477249092352739

App Secret

Show

- 部署站点时需要重新访问 Facebook 登录名安装页并注册一个新的公共 URI。

存储 Facebook 应用程序 ID 和应用程序密码

链接敏感设置，例如 Facebook App ID 和 App Secret 到你应用程序配置中使用机密 Manager。对于此教程的目的，命名为令牌 Authentication:Facebook:AppId 和 Authentication:Facebook:AppSecret。

执行以下命令以安全地存储 App ID 和 App Secret 使用密钥管理器：

```
dotnet user-secrets set Authentication:Facebook:AppId <app-id>
dotnet user-secrets set Authentication:Facebook:AppSecret <app-secret>
```

配置 Facebook 身份验证

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

添加中的 Facebook 服务 ConfigureServices 中的方法 Startup.cs 文件：

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

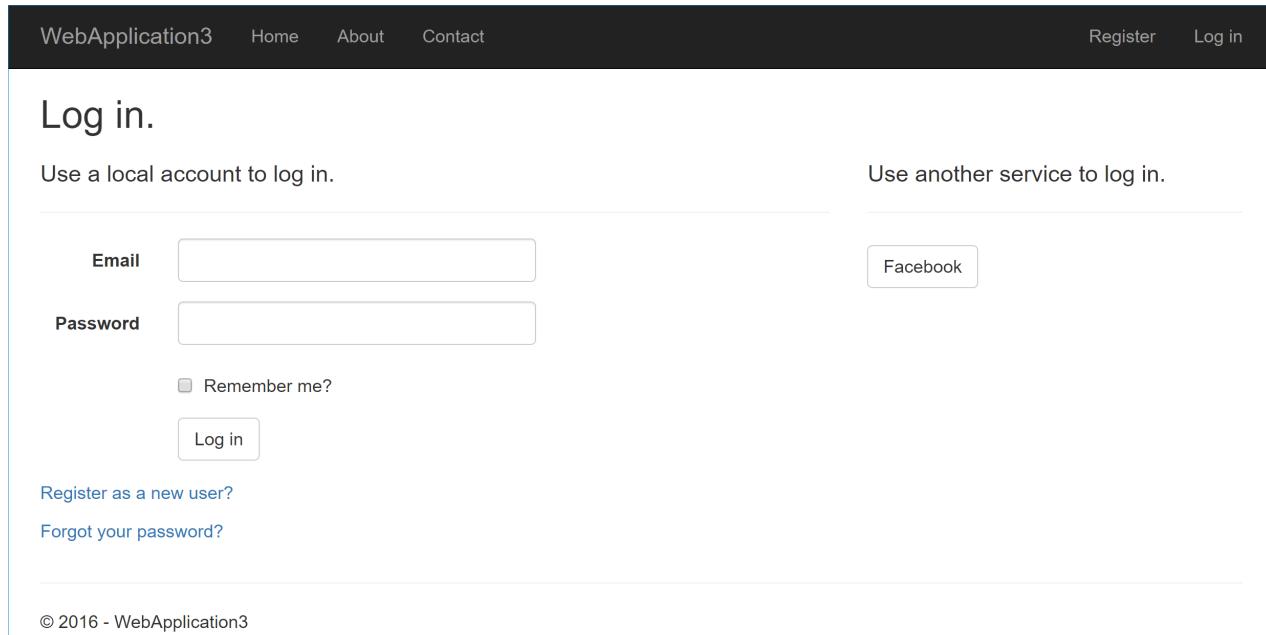
注意：调用 AddIdentity 配置的默认方案设置。AddAuthentication(string defaultScheme) 重载集 DefaultScheme 属性；并且，AddAuthentication(Action<AuthenticationOptions> configureOptions) 重载设置仅显式设置的属性。其中的任一重载应只能调用一次时添加多个身份验证提供程序。对它的后续调用也可能会覆盖任何以前配置的 AuthenticationOptions 属性。

请参阅 [FacebookOptions](#) Facebook 身份验证支持的配置选项的详细信息的 API 参考。配置选项可用于：

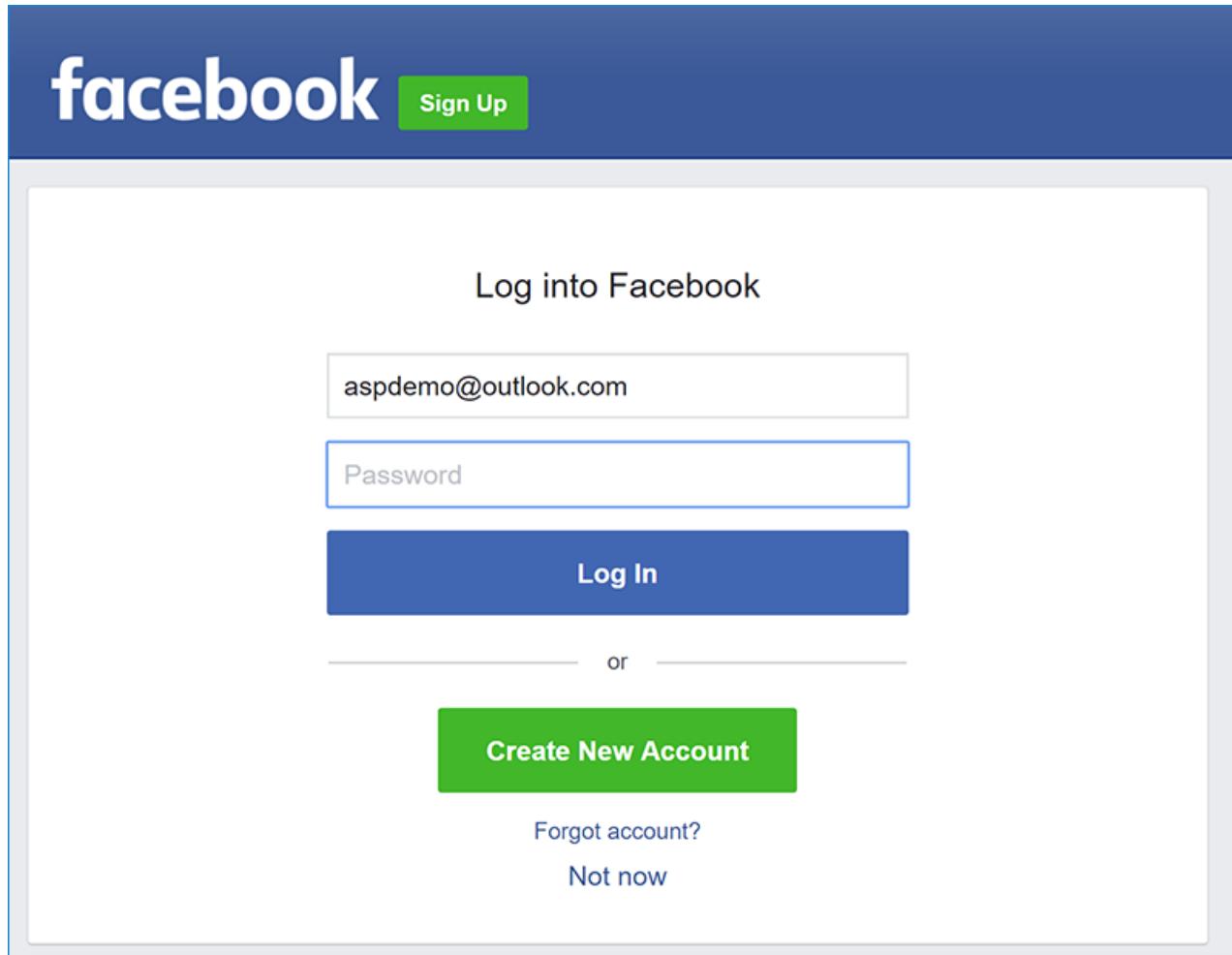
- 请求有关用户的不同信息。
- 添加查询字符串参数，以自定义登录体验。

使用 Facebook 登录

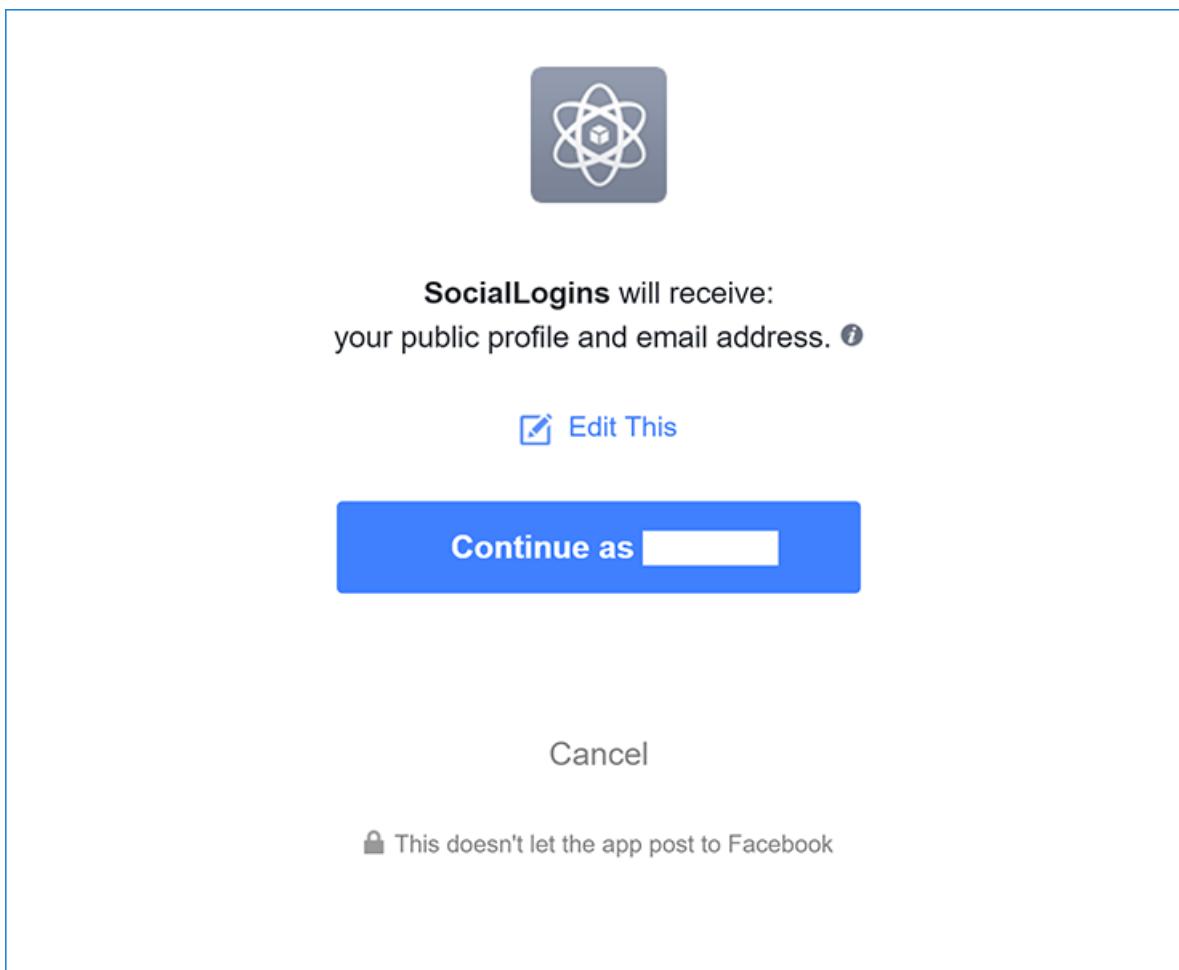
运行你的应用程序，然后单击 **登录**。你看到一个选项以使用 Facebook 登录。



当你单击**Facebook**, 你将重定向到 Facebook 进行身份验证:

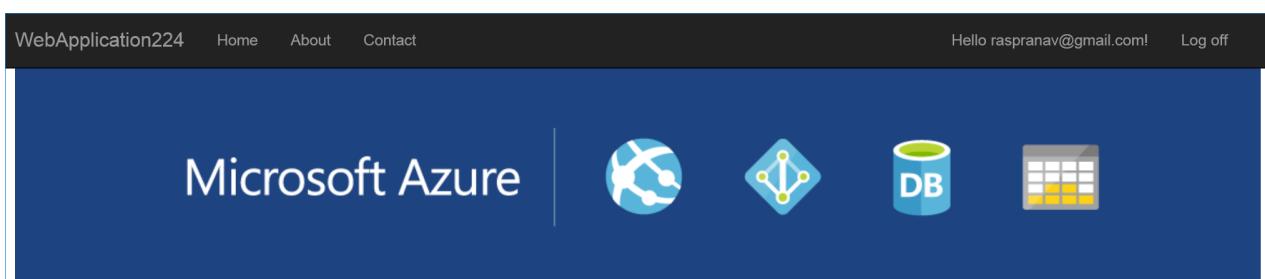


默认情况下, Facebook 身份验证请求公共配置文件和电子邮件地址:



输入你的 Facebook 凭据后你重定向回你的站点，你可以设置你的电子邮件。

现在已在使用你的 Facebook 凭据进行登录：



疑难解答

- **ASP.NET 核心 2.x 仅：**如果标识未通过调用配置 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致 `ArgumentException`：必须提供 `SignInScheme` 选项。在本教程使用的项目模板可确保，这完成的。
- 如果尚未通过应用初始迁移创建站点数据库，则获取处理请求时，数据库操作失败错误。点击 **应用迁移** 创建数据库和刷新可跳过错误。

后续步骤

- 本文介绍了你使用 Facebook 可以进行的验证。你可以遵循类似的方法进行身份验证使用其他提供商上列出[上一页](#)。
- 一旦您的网站发布到 Azure web 应用时，您应重置 `AppSecret` Facebook 开发人员门户中。
- 设置 `Authentication:Facebook:AppId` 和 `Authentication:Facebook:AppSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量中读取项。

Twitter 外部登录名与 ASP.NET 核心的安装程序

2018/5/17 • 4 min to read • [Edit Online](#)

作者: [Valeriy Novytskyy](#) 和 [Rick Anderson](#)

本教程演示如何启用到你的用户[使用 Twitter 帐户登录](#)上使用示例 ASP.NET 核心 2.0 项目创建[上一页](#)。

在 Twitter 中创建应用程序

- 导航到 <https://apps.twitter.com/> 并登录。如果你没有 Twitter 帐户，使用**立即注册** 链接创建一个。登录后，[应用程序管理页](#)所示：

The screenshot shows the Twitter Application Management interface. At the top, there's a navigation bar with the Twitter logo and the text "Application Management". On the right side of the bar is a user profile icon with a dropdown arrow. Below the bar, the main content area has a large heading "Twitter Apps". Underneath the heading, a message says "You don't currently have any Twitter Apps." followed by a "Create New App" button. At the bottom of the page, there are links for "About", "Terms", "Privacy", and "Cookies", along with a copyright notice "© 2017 Twitter, Inc.".

- 点击[创建新的应用程序](#)并填写应用程序名称，说明和公共网站(可以是临时直到你的 URI注册的域名)：

Create an application

Application Details

Name *

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Developer Agreement

Yes, I have read and agree to the [Twitter Developer Agreement](#).

[Create your Twitter application](#)

- 输入你的开发 URI 与 /signin-twitter 追加到有效的 OAuth 重定向 Uri 字段 (例如:

`https://localhost:44320/signin-twitter`)。本教程中稍后配置的 Twitter 身份验证方案将自动处理请求在 /signin-twitter 要实现的 OAuth 流路由。

- 填写表单的其余部分并点击创建 Twitter 应用程序。显示新的应用程序详细信息:

Your application has been created. Please take a moment to review and adjust your application's settings.

SocialLogins

[Test OAuth](#)[Details](#)[Settings](#)[Keys and Access Tokens](#)[Permissions](#)

Social login demo

<https://www.mysite.com/>

Organization

Information about the organization or company associated with your application. This information is optional.

Organization None

Organization website None

Application Settings

Your application's Consumer Key and Secret are used to **authenticate** requests to the Twitter Platform.

Access level Read and write ([modify app permissions](#))[Consumer Key \(API Key\)](#)[\(manage keys and access\)](#)

- 部署站点时你将需要重新访问[应用程序管理页](#)上，并注册一个新的公共 URI。

存储 Twitter ConsumerKey 和 ConsumerSecret

链接敏感设置，例如 Twitter Consumer Key 和 Consumer Secret 到你应用程序配置中使用**机密 Manager**。对于此教程的目的，命名为令牌 Authentication:Twitter:ConsumerKey 和 Authentication:Twitter:ConsumerSecret。

可以在上找到这些标记密钥和访问令牌选项卡后创建新的 Twitter 应用程序：



SocialLogins

[Test OAuth](#)[Details](#)[Settings](#)[Keys and Access Tokens](#)[Permissions](#)

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)

Consumer Secret (API Secret)

Access Level Read and write ([modify app permissions](#))

Owner

Owner ID 890753103795859456

Application Actions

[Regenerate Consumer Key and Secret](#)[Change App Permissions](#)

配置 Twitter 身份验证

在本教程使用的项目模板可确保[Microsoft.AspNetCore.Authentication.Twitter](#)已安装包。

- 若要使用 Visual Studio 2017 安装此包, 请右键单击项目并选择**管理 NuGet 包**。
- 若要使用.NET 核心 CLI 安装, 请在项目目录中执行以下命令:

```
dotnet add package Microsoft.AspNetCore.Authentication.Twitter
```

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

添加中的 Twitter 服务 `ConfigureServices` 中的方法 `Startup.cs` 文件:

```
services.AddIdentity< ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddTwitter(twitterOptions =>
{
    twitterOptions.ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"];
    twitterOptions.ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"];
});
```

注意: 调用 `AddIdentity` 配置的默认方案设置。`AddAuthentication(string defaultScheme)` 重载集 `DefaultScheme` 属性; 并且, `AddAuthentication(Action<AuthenticationOptions> configureOptions)` 重载设置仅显式设置的属性。其中的任一重载应只能调用一次时添加多个身份验证提供程序。对它的后续调用也可能会覆盖任何以前配置

的AuthenticationOptions属性。

请参阅TwitterOptions Twitter 身份验证支持的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同信息。

使用 Twitter 登录

运行你的应用程序，然后单击登录。将显示一个选项以使用 Twitter 登录：

The screenshot shows a login page for a web application named "WebApplication3". At the top, there is a navigation bar with links for "Home", "About", and "Contact", along with "Register" and "Log in" buttons. The main content area has a heading "Log in." and two sections: "Use a local account to log in." and "Use another service to log in.". The local account section contains fields for "Email" and "Password", a "Remember me?" checkbox, and a "Log in" button. The service provider section contains a "Twitter" button. Below these sections are links for "Register as a new user?" and "Forgot your password?". At the bottom of the page, there is a copyright notice: "© 2016 - WebApplication3".

WebApplication3 Home About Contact Register Log in

Log in.

Use a local account to log in.

Email

Password

Remember me?

Log in

Use another service to log in.

Twitter

Register as a new user?

Forgot your password?

© 2016 - WebApplication3

单击Twitter将重定向到 Twitter 进行身份验证：


Sign up for Twitter >

Authorize SocialLogins to use your account?



SocialLogins
www.mysite.com
Social login demo

Remember me · [Forgot password?](#)

Sign In
Cancel

This application will be able to:

- Read Tweets from your timeline.
- See who you follow.

Will not be able to:

- Follow new people.
- Update your profile.
- Post Tweets for you.
- Access your direct messages.
- See your email address.
- See your Twitter password.

在输入你的 Twitter 凭据后，你将重定向回 web 站点，你可以设置你的电子邮件。

现在已在使用你的 Twitter 凭据进行登录：

WebApplication224 Home About Contact Hello raspranav@gmail.com! Log off

Microsoft Azure





疑难解答

- **ASP.NET 核心 2.x 仅**：如果标识未通过调用配置 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致 `ArgumentException`：必须提供 `SignInScheme` 选项。在本教程使用的项目模板可确保，这完成的。
- 如果尚未通过应用初始迁移创建站点数据库，则会出现处理请求时，数据库操作失败错误。点击 **应用迁移** 创建数据库和刷新可跳过错误。

后续步骤

- 本文介绍了你使用 Twitter 可以进行的验证。你可以遵循类似的方法进行身份验证使用其他提供商上列出[上一页](#)。
- 一旦您的网站发布到 Azure web 应用时，您应重置 `ConsumerSecret` Twitter 开发人员门户中。
- 设置 `Authentication:Twitter:ConsumerKey` 和 `Authentication:Twitter:ConsumerSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量中读取项。

在 ASP.NET Core Google 外部登录安装程序

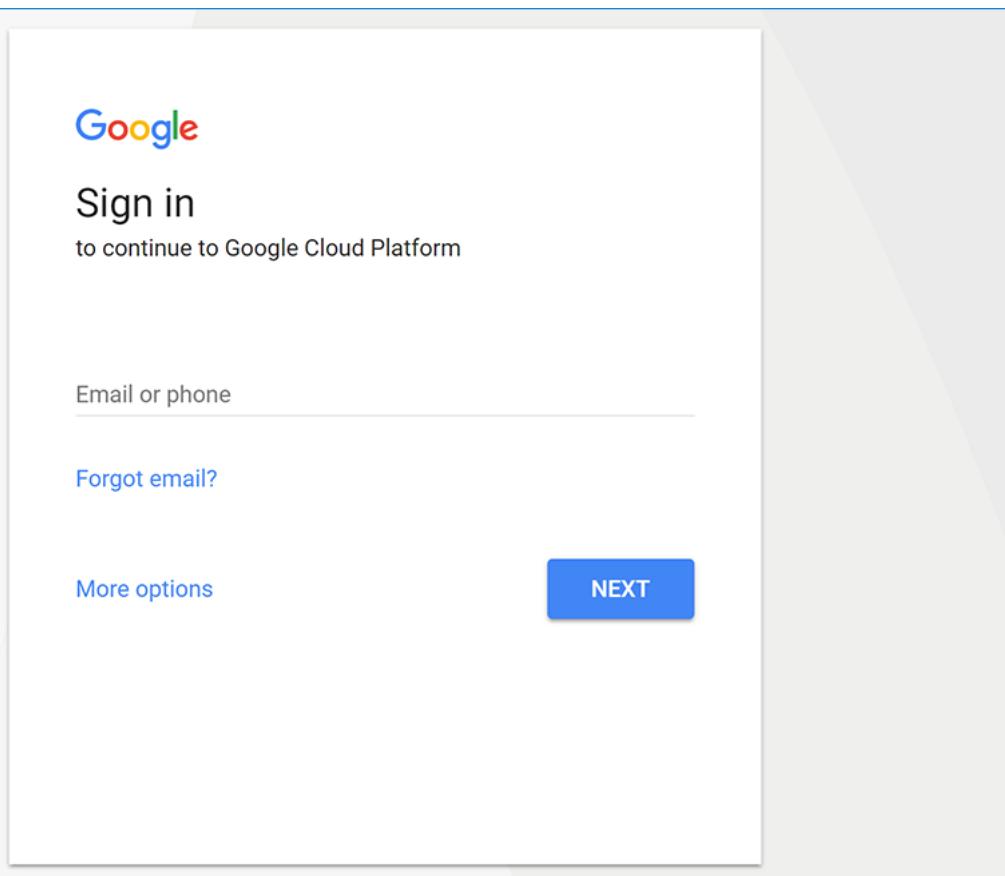
2018/5/17 • 5 min to read • [Edit Online](#)

作者: [Valeriy Novytskyy](#) 和 [Rick Anderson](#)

本教程演示如何使用户可以使用示例 ASP.NET 核心 2.0 项目上创建其 Google + 帐户登录[上一页](#)。我们先执行[官方步骤](#)在 Google API 控制台中创建新的应用程序。

在 Google API 控制台中创建应用程序

- 导航到 <https://console.developers.google.com/projectselector/apis/library> 并登录。如果你还没有 Google 帐户, 使用[更多选项 > 创建帐户](#) 链接创建一个:



- 重定向到**API Manager**库页:

The screenshot shows the Google APIs console interface. On the left, there's a sidebar with three main options: 'Dashboard', 'Library' (which is highlighted with a blue background), and 'Credentials'. The main content area is titled 'Library' and contains a message: 'To view this page, select a project.' Below this message is a blue 'Create' button.

- 点击创建并输入你项目名称:

The screenshot shows the 'New Project' dialog box. It starts with a message: 'You have 12 projects remaining in your quota. [Learn more.](#)' Below this is a 'Project name' field containing 'My Project'. Underneath, it says 'Your project ID will be phrasal-charger-175015' with an 'Edit' link. There's a section for receiving updates: 'Please email me updates regarding feature announcements, performance suggestions, feedback surveys and special offers.' with 'Yes' and 'No' radio buttons. Below that is an agreement statement: 'I agree that my use of any services and related APIs is subject to my compliance with the applicable Terms of Service.' with 'Yes' and 'No' radio buttons. At the bottom are two buttons: a blue 'Create' button and a white 'Cancel' button.

- 接受对话框中之后, 你将重定向回库页允许你选择为新应用程序的功能。查找**Google + API**在列表中, 单击其链接来添加的 API 功能:

Google APIs SocialLogins ▾

API Manager

- Dashboard
- Library**
- Credentials

Library

Google APIs

Google+ API

Back to popular APIs

Name	Description
Google+ API	The Google+ API enables developers to build on top of the Google+ platform.
Google+ Domains API	The Google+ Domains API enables developers to build on top of the Google+ platform for Google Apps

- 显示新添加的 API 的页。点击启用将 Google + 登录功能中添加到你的应用程序：

← Google+ API ➤ ENABLE

About this API

The Google+ API enables developers to build on top of the Google+ platform.

Using credentials with this API

Accessing user data with OAuth 2.0

You can access user data with this API. On the Credentials page, create an OAuth 2.0 client ID. A client ID requests user consent so that your app can access user data. Include that client ID when making your API call to Google. [Learn more](#)

Server-to-server interaction

You can use this API to perform server-to-server interaction, for example between a web application and a Google service. You'll need a service account, which enables app-level authentication. You'll also need a service account key, which is used to authorize your API call to Google. [Learn more](#)

- 启用 API 后, 点击创建凭据配置机密：

Google APIs SocialLogins ▾

API Manager

- Dashboard
- Library**
- Credentials

← Google+ API ➤ DISABLE

⚠ To use this API, you may need credentials. Click "Create credentials" to get started.

[Create credentials](#)

Overview Quotas

- 选择：
 - Google + API**
 - Web 服务器 (例如 node.js, Tomcat), 和
 - 用户数据:

**API** API Manager

Dashboard

Library

Credentials

Credentials

Add credentials to your project

1 Find out what kind of credentials you need

We'll help you set up the correct credentials

If you wish you can skip this step and create an [API key](#), [client ID](#), or [service account](#)

Which API are you using?

Determines what kind of credentials you need.

Google+ API

Where will you be calling the API from?

Determines which settings you'll need to configure.

Web server (e.g. node.js, Tomcat)

What data will you be accessing?

User data

Access data belonging to a Google user, with their permission

Application data

Access data belonging to your own application

What credentials do I need?

2 Get your credentials

Cancel

- 点击我需要什么凭据？ 随即将跳转到应用配置的第二个步骤创建 OAuth 2.0 客户端 ID:

The screenshot shows the Google APIs console interface. On the left, there's a sidebar with 'API Manager' and 'SocialLogins' tabs. Under 'API Manager', 'Dashboard' and 'Library' are listed, while 'Credentials' is selected and highlighted with a blue background. The main content area is titled 'Add credentials to your project'. It has a step-by-step guide: 'Find out what kind of credentials you need' (with a note about calling Google+ API from a web server), followed by 'Create an OAuth 2.0 client ID'. A 'Name' field contains 'Web client 1'. Below it, under 'Restrictions', there are sections for 'Authorized JavaScript origins' (containing 'http://www.example.com') and 'Authorized redirect URIs' (containing 'http://www.example.com/oauth2callback'). At the bottom right is a blue 'Create client ID' button.

- 因为我们将使用只对一个功能（登录），我们可以输入相同创建 Google + 项目名称与我们的项目使用的 OAuth 2.0 客户端 id。
- 输入你的开发 URI 与 `/signin-google` 追加到已授权重定向 Uri 字段（例如：
`https://localhost:44320/signin-google`）。本教程中稍后配置 Google 身份验证将自动处理请求在 `/signin-google` 要实现的 OAuth 流路由。
- 按 tab 键以添加已授权重定向 Uri 条目。
- 点击创建客户端 ID，随即将跳转到第三个步骤中，设置 OAuth 2.0 许可屏幕：

**API** API Manager

Dashboard

Library

Credentials

Credentials

Add credentials to your project

- Find out what kind of credentials you need

Calling Google+ API from a web server

- Create an OAuth 2.0 client ID

Created OAuth client 'Web client 1'

3 Set up the OAuth 2.0 consent screen

Email address

Product name shown to users

 Product name

More customization options

Continue



The consent screen will be shown to users whenever you request access to their private data using your client ID. It will be shown for all applications registered in this project.

<|

- 输入你的面向公众电子邮件地址和产品名称Google + 提示用户登录时显示为你的应用。其他选项下有更多的自定义选项。
- 点击继续继续到最后一个步骤中，下载凭据：

API Manager

Credentials

Add credentials to your project

Find out what kind of credentials you need
Calling Google+ API from a web server

Create an OAuth 2.0 client ID
Created OAuth client 'Web client 1'

Set up the OAuth 2.0 consent screen

Download credentials

Client ID 114882279233-bqsm444d7tcd4gkm3gb46qd32653fjcb.apps.googleusercontent.com

Download I'll do this later

Done Cancel

- 点击下载若要使用应用程序机密，保存 JSON 文件和完成以完成创建新的应用程序。
- 部署站点时你将需要重新访问**Google 控制台**和注册一个新的公共 url。

应用商店 Google ClientID 和 ClientSecret

链接敏感设置，例如 Google Client ID 和 Client Secret 到你应用程序配置中使用**机密 Manager**。对于此教程的目的，命名为令牌 Authentication:Google:ClientId 和 Authentication:Google:ClientSecret。

这些标记的值可以在下一步中下载的 JSON 文件中找到 web.client_id 和 web.client_secret。

配置 Google 身份验证

- ASP.NET Core 2.x
- ASP.NET Core 1.x

添加中的 Google 服务 ConfigureServices 中的方法 Startup.cs 文件：

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId = Configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
});
```

注意：调用 AddIdentity 配置的默认方案设置。AddAuthentication(string defaultScheme) 重载集 DefaultScheme 属性；并且，AddAuthentication(Action<AuthenticationOptions> configureOptions) 重载设置仅显式设置的属性。其中的任一重载应只能调用一次时添加多个身份验证提供程序。对它的后续调用也可能会覆盖任何以前配置的 AuthenticationOptions 属性。

请参阅 [GoogleOptions](#) Google 身份验证支持的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同

信息。

使用 Google 登录

运行你的应用程序，然后单击登录。将显示一个选项以使用 Google 登录：

The screenshot shows a login form for 'WebApplication3'. At the top, there's a navigation bar with links for Home, About, Contact, Register, and Log in. The main area has a heading 'Log in.' followed by two sections: 'Use a local account to log in.' and 'Use another service to log in.'. The local account section contains fields for Email and Password, a 'Remember me?' checkbox, and a 'Log in' button. The service provider section contains a single 'Google' button. Below the form are links for 'Register as a new user?' and 'Forgot your password?'. At the bottom, a copyright notice reads '© 2016 - WebApplication3'.

Use a local account to log in.

Email

Password

Remember me?

Log in

Use another service to log in.

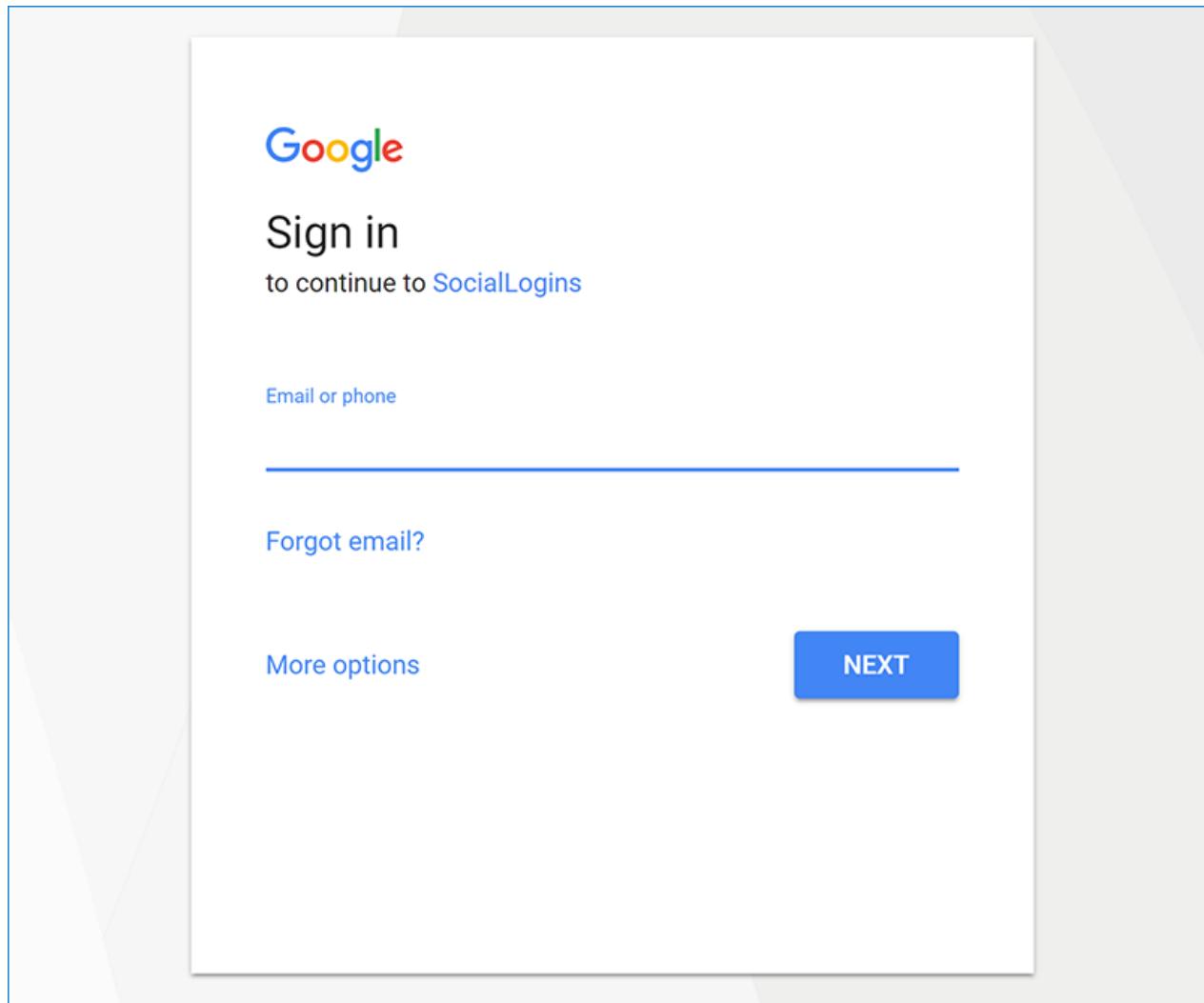
Google

[Register as a new user?](#)

[Forgot your password?](#)

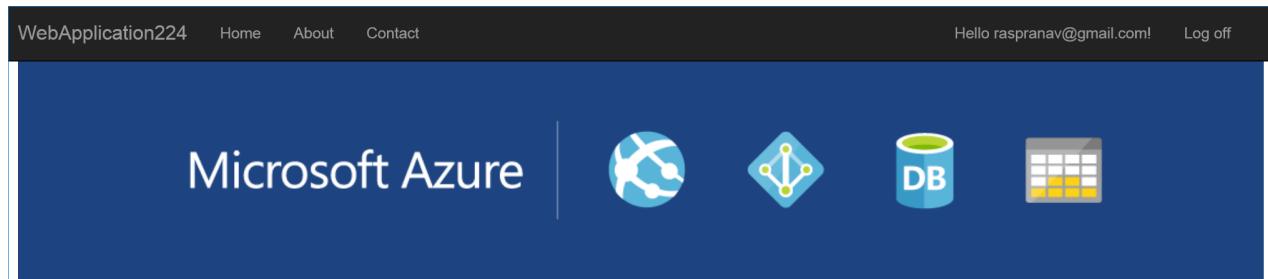
© 2016 - WebApplication3

当你单击 Google 时，你将重定向到 Google 身份验证：



在输入你的 Google 凭据，然后你将重定向回 web 站点，你可以设置你的电子邮件。

现在已在使用你的 Google 凭据进行登录：



疑难解答

- 如果你收到 `403 (Forbidden)` 从你自己的应用，请确保在开发模式（或中断到调试器中相同的错误），运行时的错误页 **Google + API** 已在中启用**API Manager** 库按照列出的步骤[更早版本在此页](#)。如果在登录不起作用，并且不能接收任何错误，切换到开发模式以使问题更易于调试。
- ASP.NET 核心 2.x 仅**：如果标识未通过调用配置 `services.AddIdentity` 中 `ConfigureServices`，尝试进行身份验证将导致 `ArgumentException`: 必须提供 `SignInScheme` 选项。在本教程使用的项目模板可确保，这完成的。
- 如果尚未通过应用初始迁移创建站点数据库，则会出现处理请求时，数据库操作失败错误。点击[应用迁移](#) 创建数据库和刷新可跳过错误。

后续步骤

- 本文介绍了你使用 Google 可以进行的验证。你可以遵循类似的方法进行身份验证使用其他提供商上列出[上一页](#)。
- 一旦您的网站发布到 Azure web 应用时，您应重置 `clientSecret` 在 Google API 控制台中。
- 设置 `Authentication:Google:ClientId` 和 `Authentication:Google:ClientSecret` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量中读取项。

使用 ASP.NET Core 的 Microsoft 帐户外部登录设置

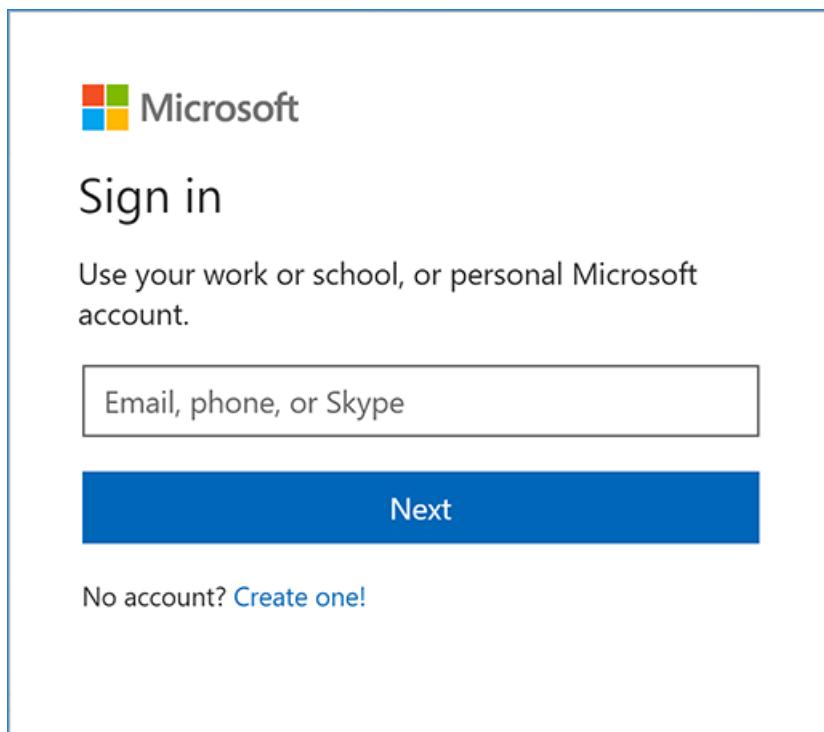
2018/5/17 • 5 min to read • [Edit Online](#)

作者: [Valeriy Novytskyy](#) 和 [Rick Anderson](#)

本教程演示如何使用户可以使用示例 ASP.NET 核心 2.0 项目上创建其 Microsoft 帐户登录[上一页](#)。

在 Microsoft 开发人员门户中创建应用程序

- 导航到 <https://apps.dev.microsoft.com> 和创建或登录到 Microsoft 帐户:



如果你尚没有 Microsoft 帐户, 请点击**[创建一个](#) ! 在登录后你将重定向到我的应用程序**页:

A screenshot of the Microsoft Application Registration Portal. The top navigation bar includes the Microsoft logo, "Application Registration Portal", "Docs", "Feedback", and a user profile icon. The main area is titled "My applications" with a "Learn More" link and a "Add an app" button. Below this, a table lists an application: "Name" is "TestApp" and "App ID / Client Id" is "b7e887c6-23e7-48fa-802a-c528e47cf02a". To the right of the table is a "Delete" button.

- 点击添加应用程序在右上角, 然后输入你应用程序名称和联系人电子邮件:



Register your application

Application Name

SocialLogins

Contact Email

Used for important communications about your application

aspdemo@outlook.com

Guided Setup

Let us help you get started

By proceeding, you agree to the [Microsoft Platform Policies](#)

Create

- 对于此教程的目的, 清除指导安装程序复选框。
- 点击创建继续注册页。提供名称并记下的值**应用程序 Id**, 其中使用即 `ClientId` 教程后面:

[My applications](#) / SocialLogins

SocialLogins Registration

[Click here for help integrating your application with Microsoft.](#)

Properties

Name

SocialLogins

Application Id

4d519acd-6a8c-4df4-a246-fe5ea741c9db

Application Secrets

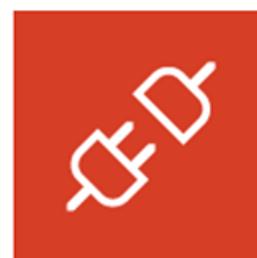
[Generate New Password](#) [Generate New Key Pair](#) [Upload Public Key](#)

Platforms

[Add Platform](#)

- 点击添加平台中平台部分，并选择**Web**平台：

Add Platform



[Cancel](#)

- 在新Web平台部分中，输入与你开发的 URL /signin-microsoft 追加到重定向 Url 字段（例如：`https://localhost:44320/signin-microsoft`）。本教程中稍后配置的 Microsoft 身份验证方案将自动处理请求在 /signin-microsoft 要实现的 OAuth 流路由：

The screenshot shows a 'Platforms' configuration screen. A 'Web' platform is listed with a 'Delete' button. Under 'Redirect URLs', there is a checked checkbox for 'Allow Implicit Flow' and a text input field containing `https://localhost:44320/signin-microsoft`. There is also a 'Logout URL' input field with the placeholder 'e.g. https://myapp.com/end-session'.

- 点击添加 URL 以确保该 URL 已添加。
- 填写应用程序的任何其他设置，如有必要，并点击保存底部的页后，可以将更改保存到应用配置。
- 部署站点时你将需要重新访问注册页，并设置新的公共 URL。

存储 Microsoft 应用程序 Id 和密码

- 请注意 Application Id 上显示注册页。
- 点击生成新密码中应用程序机密部分。此时将显示一个框，其中你可以将复制的应用程序密码：

The dialog box displays the message "New password generated". Below it, a note says "This is the only time when it will be displayed. Please store it securely." A redacted password value "bae" is shown in a text input field, followed by an "Ok" button.

链接敏感设置，例如 Microsoft Application ID 和 Password 到你应用程序配置中使用机密 Manager。对于此教程的目的，命名为令牌 Authentication:Microsoft:ApplicationId 和 Authentication:Microsoft:Password。

配置 Microsoft 帐户身份验证

在本教程使用的项目模板可确保 Microsoft.AspNetCore.Authentication.MicrosoftAccount 已安装包。

- 若要使用 Visual Studio 2017 安装此包, 请右键单击项目并选择管理 NuGet 包。

- 若要使用.NET 核心 CLI 安装, 请在项目目录中执行以下命令:

```
dotnet add package Microsoft.AspNetCore.Authentication.MicrosoftAccount
```

- ASP.NET Core 2.x
- ASP.NET Core 1.x

添加 Microsoft 帐户服务 `ConfigureServices` 中的方法 `Startup.cs` 文件:

```
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
{
    microsoftOptions.ClientId = Configuration["Authentication:Microsoft:ApplicationId"];
    microsoftOptions.ClientSecret = Configuration["Authentication:Microsoft:Password"];
});
```

注意: 调用 `AddIdentity` 配置的默认方案设置。`AddAuthentication(string defaultScheme)` 重载集 `DefaultScheme` 属性; 并且, `AddAuthentication(Action<AuthenticationOptions> configureOptions)` 重载设置仅显式设置的属性。其中的任一重载应只能调用一次时添加多个身份验证提供程序。对它的后续调用也可能会覆盖任何以前配置的 `AuthenticationOptions` 属性。

尽管在 Microsoft 开发人员门户上使用的术语名称这些令牌 `ApplicationId` 和 `Password`, 它们作为公开 `ClientId` 和 `ClientSecret` 到配置 API。

请参阅 [MicrosoftAccountOptions](#) 支持 Microsoft 帐户身份验证的配置选项的详细信息的 API 参考。这可以用于请求有关用户的不同信息。

使用 Microsoft 帐户登录

运行你的应用程序, 然后单击 [登录](#)。将显示一个选项以使用 Microsoft 登录:

The screenshot shows a web application's login page. At the top, there is a navigation bar with links for Home, About, Contact, Register, and Log in. Below the navigation bar, the page title is "Log in.". There are two main sections for logging in: "Use a local account to log in." and "Use another service to log in.". The "Local Account" section contains fields for Email and Password, a "Remember me?" checkbox, and a "Log in" button. The "Service" section contains a "Microsoft" button. At the bottom of the page, there are links for "Register as a new user?", "Forgot your password?", and copyright information: "© 2016 - WebApplication3".

WebApplication3 Home About Contact Register Log in

Log in.

Use a local account to log in.

Email

Password

Remember me?

Log in

Use another service to log in.

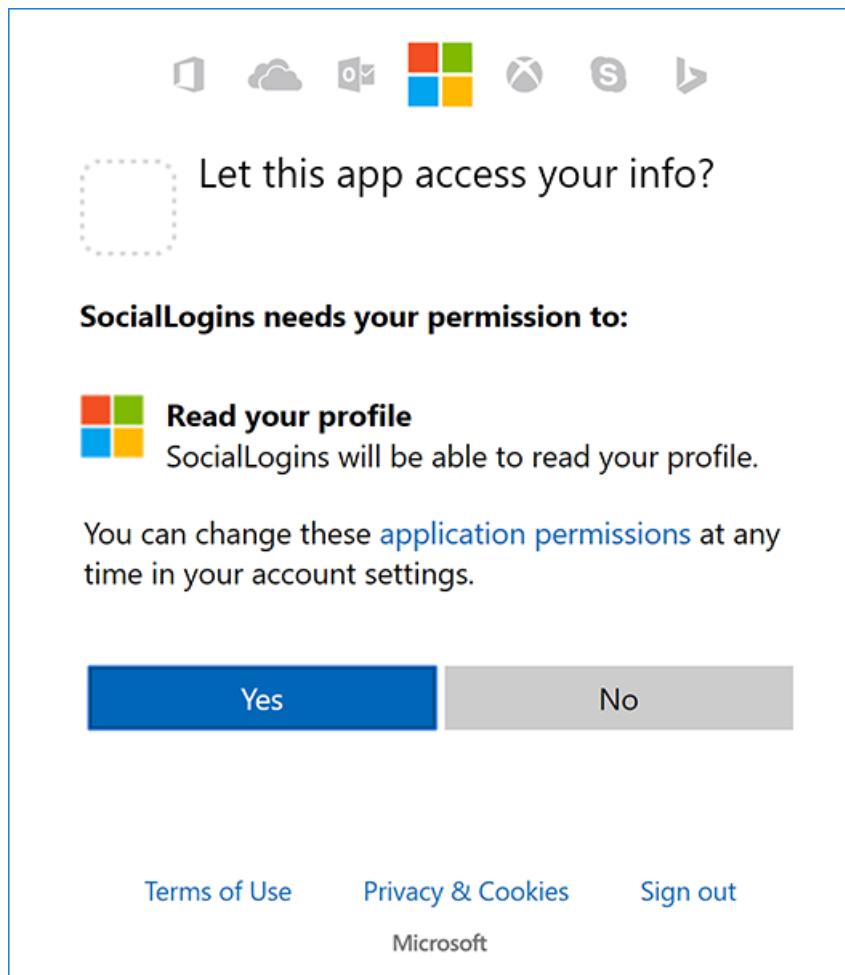
Microsoft

Register as a new user?

Forgot your password?

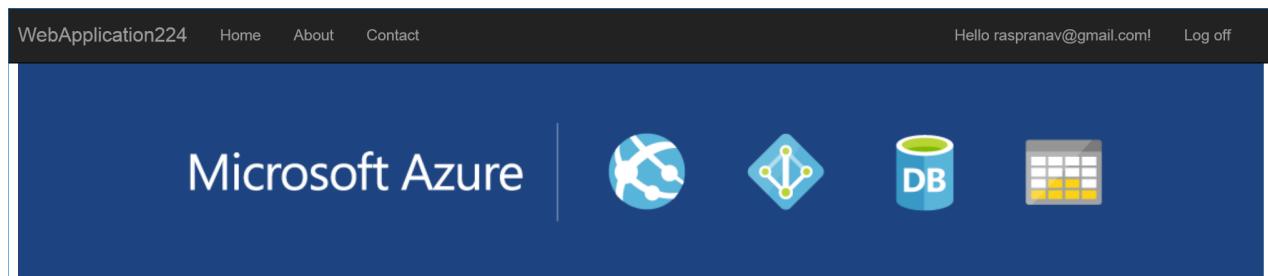
© 2016 - WebApplication3

当你单击 on Microsoft 时, 你将重定向到 Microsoft 进行身份验证。(如果尚未登录), 使用你的 Microsoft 帐户登录后你将会提示您允许此应用访问你的信息:



点击是, 将重定向回 web 站点, 你可以设置你的电子邮件。

现在已在使用您的 Microsoft 凭据进行登录:



疑难解答

- 如果 Microsoft 帐户提供程序将你重定向到登录错误页, 请注意错误标题和说明查询字符串参数紧接 `#` (哈希标记) 的 Uri 中。
尽管错误消息看起来, 以指示 Microsoft 身份验证问题, 但最常见的原因是你的应用程序与任何不匹配的 Uri 重定向 Uri 指定的 Web 平台。
- ASP.NET 核心 2.x 仅:** 如果标识未通过调用配置 `services.AddIdentity` 中 `ConfigureServices`, 尝试进行身份验证将导致 `ArgumentException`: 必须提供 `SignInScheme` 选项。在本教程使用的项目模板可确保, 这完成的。
- 如果尚未通过应用初始迁移创建站点数据库, 则会出现处理请求时, 数据库操作失败错误。点击应用迁移创建数据库和刷新可跳过错误。

后续步骤

- 本文介绍了你与 Microsoft 可以进行的验证。你可以遵循类似的方法进行身份验证使用其他提供商上列出[上一页](#)。
- 一旦您的网站发布到 Azure web 应用时，你应该创建一个新 `Password` Microsoft 开发人员门户中。
- 设置 `Authentication:Microsoft:ApplicationId` 和 `Authentication:Microsoft:Password` 作为在 Azure 门户中的应用程序设置。配置系统设置以从环境变量中读取项。

简短的调查的其他身份验证提供程序

2018/4/10 • 1 min to read • [Edit Online](#)

通过[Rick Anderson](#), [Pranav Rastogi](#), 和[Valeriy Novytsky](#)

此处会设置对于某些其他常见 OAuth 提供程序的说明。第三方 NuGet 包, 例如由维护的[aspnet contrib](#)可以用于补充由 ASP.NET Core 团队实现的身份验证提供程序。

- 设置**LinkedIn**登录: <https://www.linkedin.com/developer/apps>。请参阅[官方步骤](#)。
- 设置**Instagram**登录: <https://www.instagram.com/developer/register/>。请参阅[官方步骤](#)。
- 设置**Reddit**登录: <https://www.reddit.com/login?dest=https%3A%2F%2Fwww.reddit.com%2Fprefs%2Fapps>。请参阅[官方步骤](#)。
- 设置**Github**登录: https://github.com/login?return_to=https%3A%2F%2Fgithub.com%2Fsettings%2Fapplications%2Fnew。请参阅[官方步骤](#)。
- 设置**Yahoo**登录: <https://login.yahoo.com/config/login?src=devnet&.done=http%3A%2F%2Fdeveloper.yahoo.com%2Fapps%2Fcreate%2F>。请参阅[官方步骤](#)。
- 设置**Tumblr**登录: <https://www.tumblr.com/oauth/apps>。请参阅[官方步骤](#)。
- 设置**Pinterest**登录: <https://www.pinterest.com/login/?next=http%3A%2F%2Fdevsite%2Fapps%2F>。请参阅[官方步骤](#)。
- 设置**Pocket**登录: <https://getpocket.com/developer/apps/new>。请参阅[官方步骤](#)。
- 设置**Flickr**登录: <https://www.flickr.com/services/apps/create>。请参阅[官方步骤](#)。
- 设置**Dribbble**登录: <https://dribbble.com/signup>。请参阅[官方步骤](#)。
- 设置**Vimeo**登录: <https://vimeo.com/join>。请参阅[官方步骤](#)。
- 设置**SoundCloud**登录: <https://soundcloud.com/you/apps/new>。请参阅[官方步骤](#)。
- 设置**VK**登录: <https://vk.com/apps?act=manage>。请参阅[官方步骤](#)。

使用 WS 联合身份验证在 ASP.NET 核心中的用户进行身份验证

2018/4/10 • 4 min to read • [Edit Online](#)

本教程演示如何使用户能够使用 WS 联合身份验证提供程序 (如 Active Directory 联合身份验证服务 (ADFS) 登录或 Azure Active Directory (AAD))。它使用 ASP.NET 核心 2.0 示例应用程序中所述 Facebook、Google、和外部提供程序身份验证。

对于 ASP.NET 核心 2.0 应用程序中，WS 联合身份验证的支持由提供 [Microsoft.AspNetCore.Authentication.WsFederation](#)。此组件移植从 [Microsoft.Owin.Security.WsFederation](#) 和共享许多该组件的机制。但是，组件的几个重要方面不同。

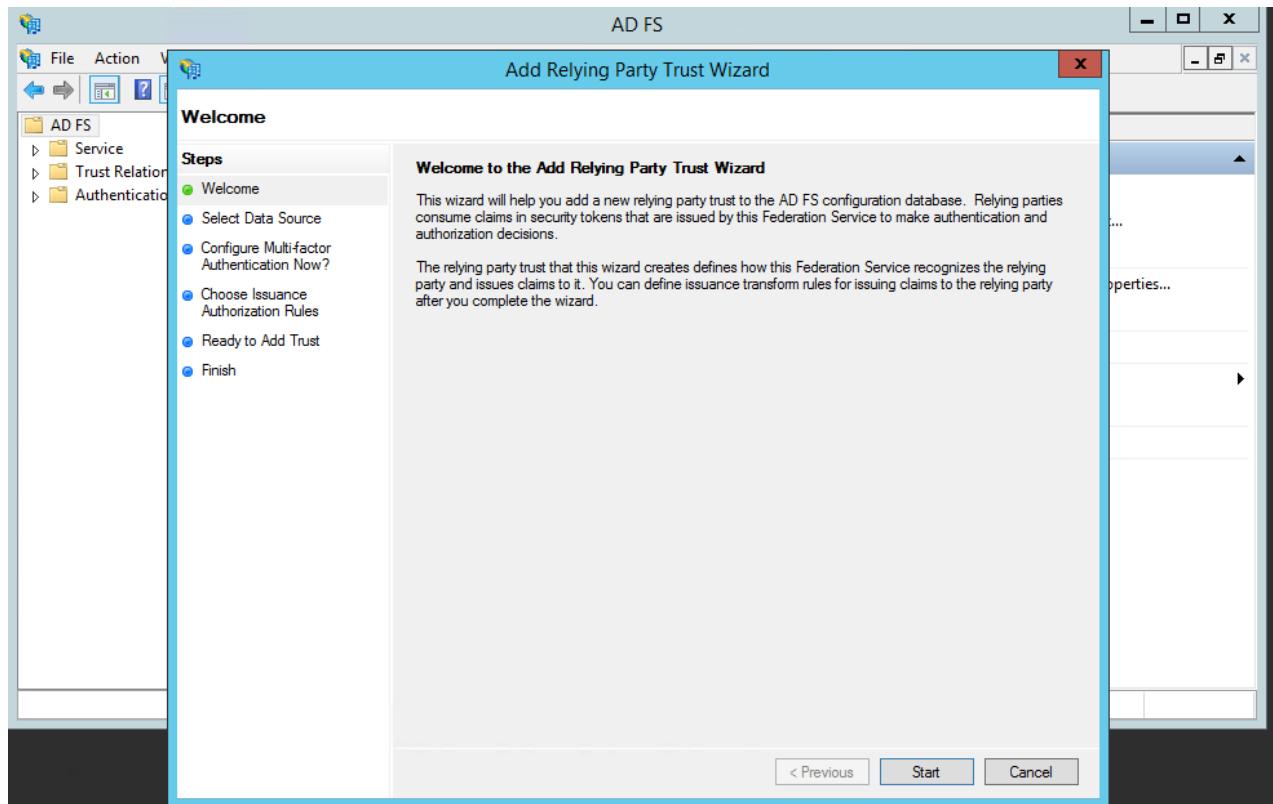
默认情况下，新的中间件：

- 不允许未经请求的登录名。这一功能的 WS 联合身份验证协议容易受到 CSRF 攻击。但是，可以使用启用此 `AllowUnsolicitedLogins` 选项。
- 不会检查消息登录的每个窗体发布请求。只请求在到 `CallbackPath` 登录程序。检查 `CallbackPath` 默认为 `/signin-wsfed` 但可以更改。此路径可以与其他身份验证提供程序共享通过启用 `SkipUnrecognizedRequests` 选项。

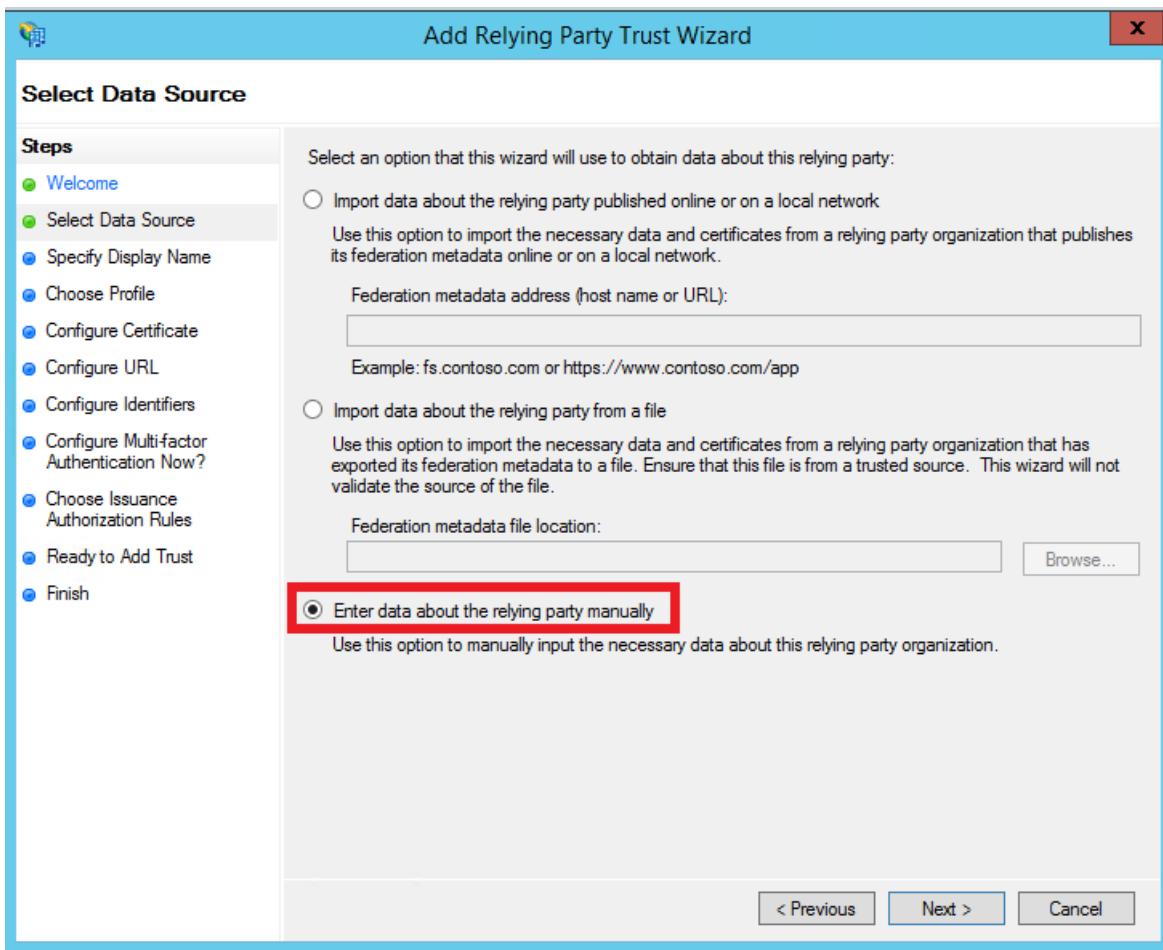
向 Active Directory 注册应用程序

Active Directory 联合身份验证服务

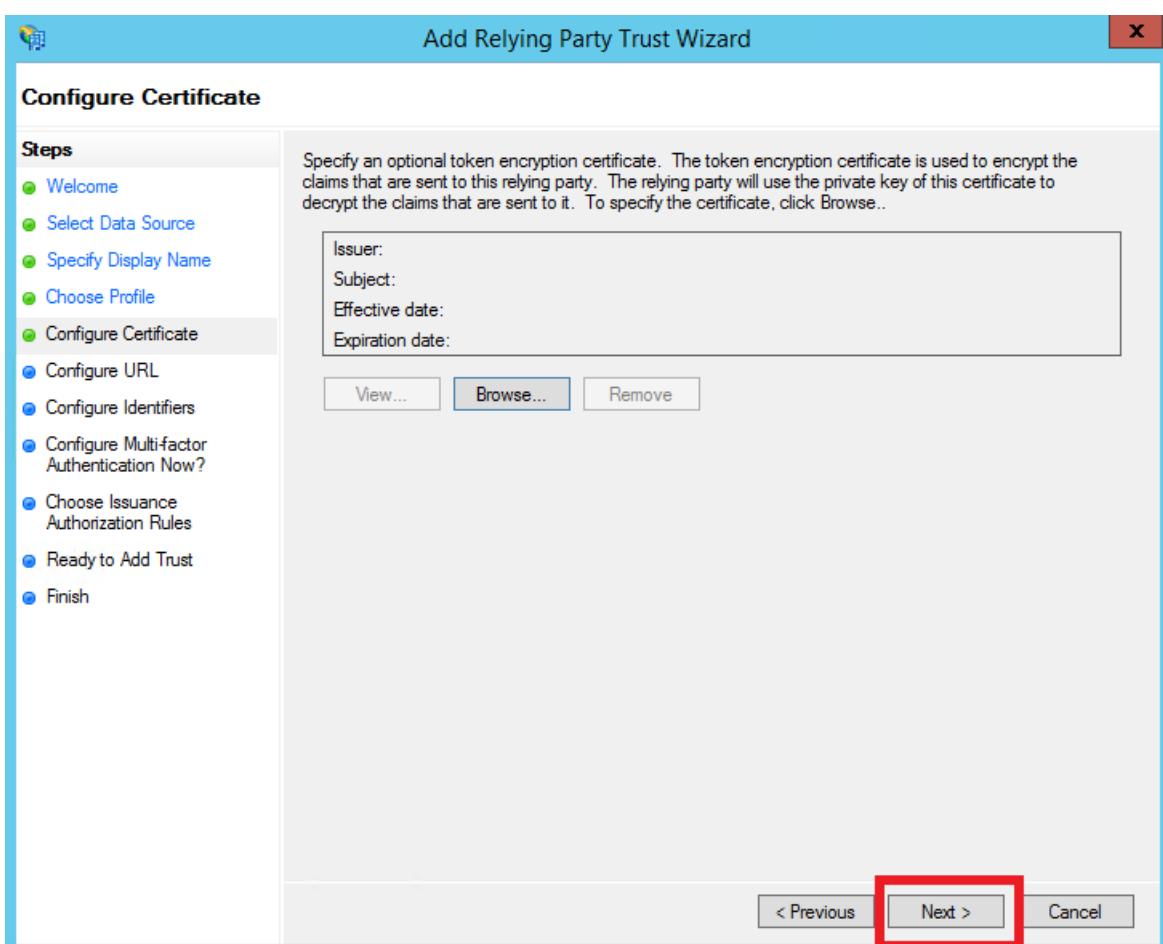
- 打开服务器的添加信赖方信任向导从 ADFS 管理控制台：



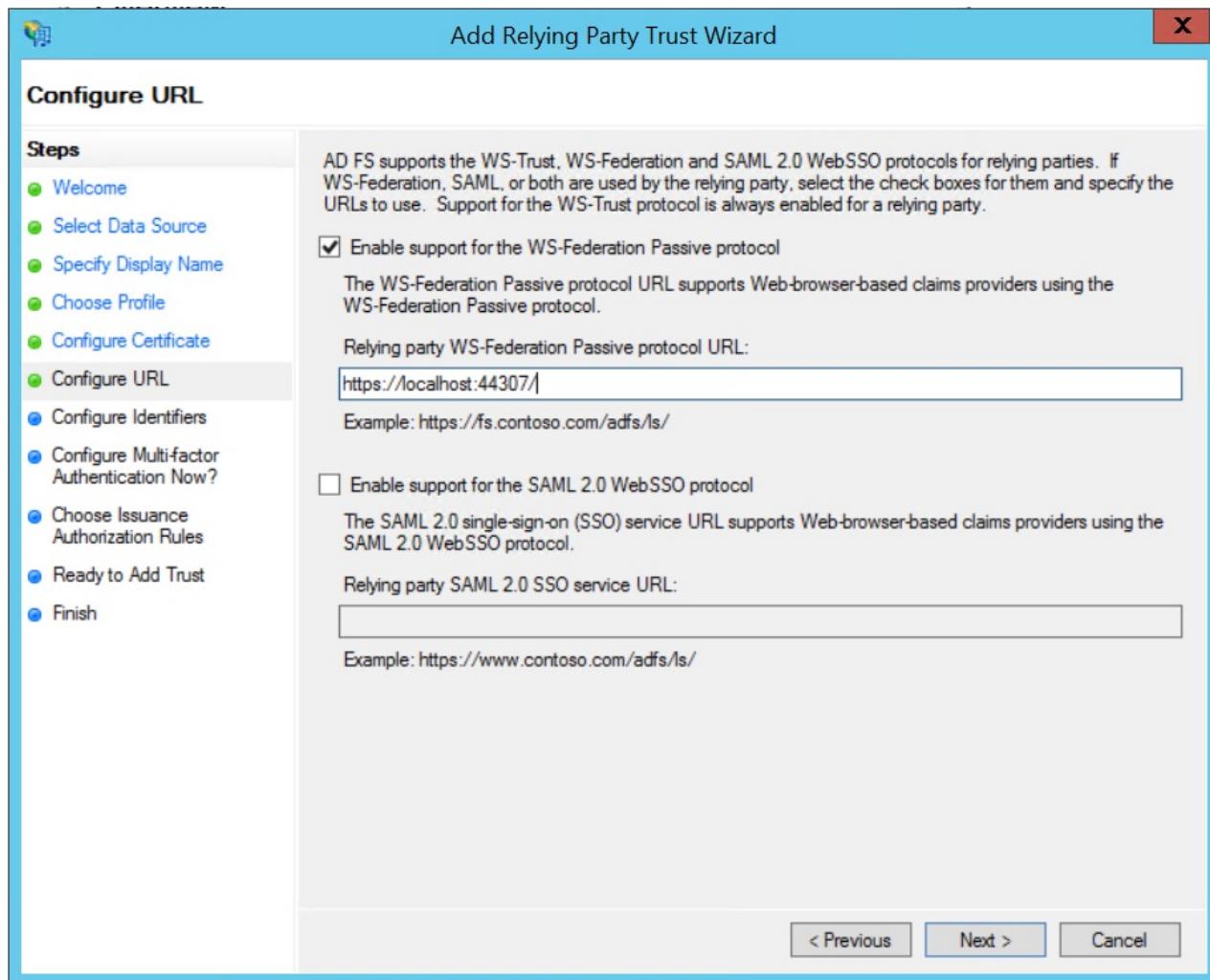
- 选择了手动输入数据：



- 输入信赖方的显示名称。名称并不重要到 ASP.NET 核心应用程序。
- Microsoft.AspNetCore.Authentication.WsFederation 缺少用于令牌加密，因此未配置令牌加密证书的支持：



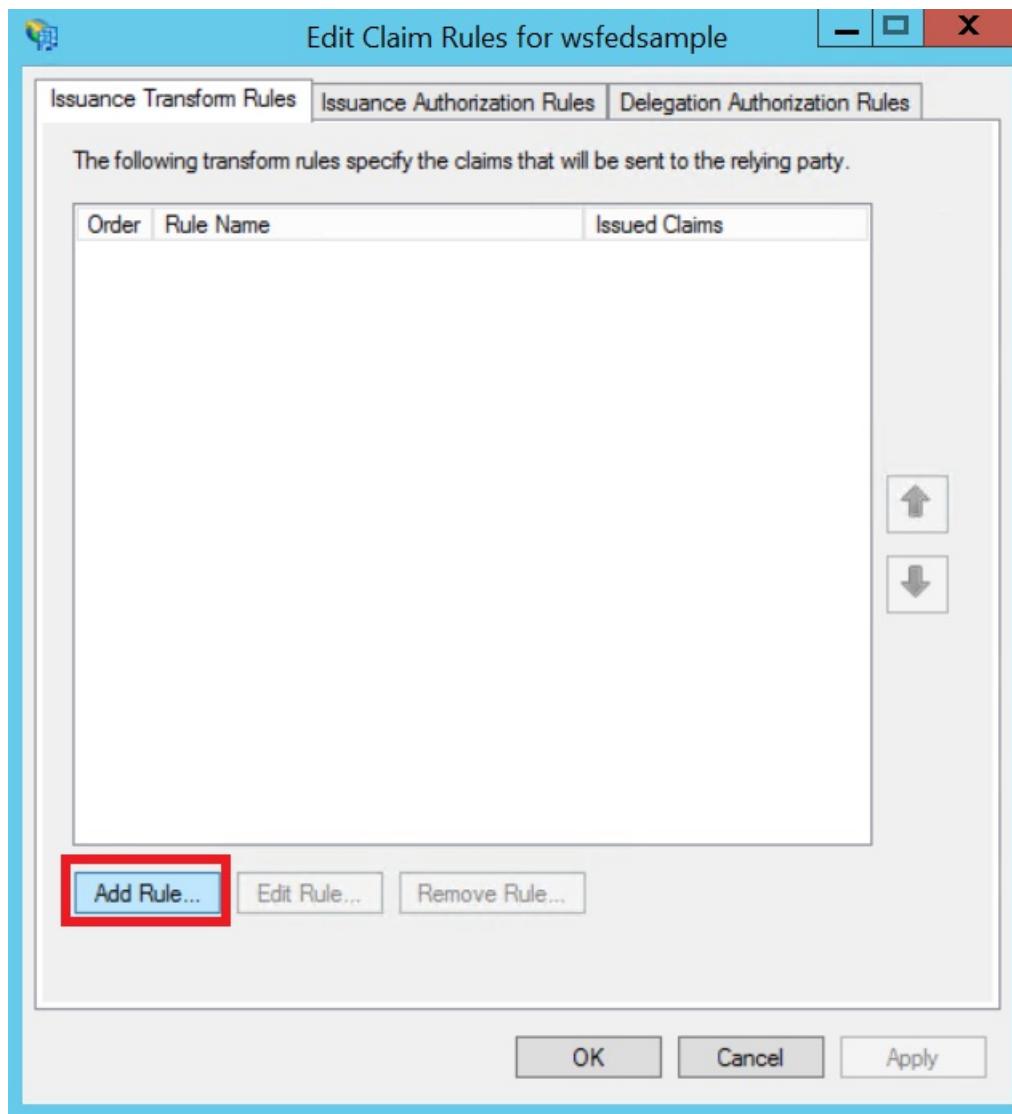
- 启用支持 WS 联合身份验证被动协议，使用应用的 URL。验证端口正确应用：



注意

这必须是 HTTPS URL。在开发期间托管应用程序时，IIS Express 可以提供自签名的证书。Kestrel 需要手动证书配置。请参阅[Kestrel 文档](#)有关详细信息。

- 单击下一步完成该向导的其余部分和关闭末尾。
- ASP.NET 核心标识需要名称 ID 声明。添加一个从编辑声明规则对话框：



- 在添加转换声明规则向导，保留默认以声明方式发送 LDAP 属性选择的模板，然后单击下一步。添加规则映射SAM 帐户名 LDAP 属性到名称 ID 传出声明：

Add Transform Claim Rule Wizard

Configure Rule

Steps

- Choose Rule Type
- Configure Claim Rule

You can configure this rule to send the values of LDAP attributes as claims. Select an attribute store from which to extract LDAP attributes. Specify how the attributes will map to the outgoing claim types that will be issued from the rule.

Claim rule name: send name id

Rule template: Send LDAP Attributes as Claims

Attribute store: Active Directory

Mapping of LDAP attributes to outgoing claim types:

LDAP Attribute (Select or type to add more)	Outgoing Claim Type (Select or type to add more)
SAM-Account-Name	Name ID
*	*

< Previous Finish Cancel

- 单击完成 > 确定中编辑声明规则窗口。

Azure Active Directory

- 导航到 AAD 租户的应用注册边栏选项卡。单击新应用程序注册:

Home > wsfedsample - App registrations

wsfedsample - App registrations

Azure Active Directory

Overview Quick start New application registration Endpoints Troubleshoot

To view and manage your registrations for converged applications, please visit the [Microsoft Application Console](#).

Search by name or AppID My apps

DISPLAY NAME	APPLICATION TYPE	APPLICATION ID
No results.		

MANAGE

- Users
- Groups
- Enterprise applications
- Devices
- App registrations
- Application proxy

- 输入应用程序注册的名称。这并不重要到 ASP.NET 核心应用程序。
- 输入作为侦听的应用程序的 URL 登录 URL:



* Sign-on URL ⓘ

https://localhost:44307

Create

- 单击终结点并记下联合元数据文档URL。这是 WS 联合身份验证中间件的 `MetadataAddress`：

The screenshot shows the Microsoft Application Console with the following details:

- New application registration
- Endpoints (highlighted with a red box)
- Troubleshoot

To view and manage your registrations for converged applications, please visit the Microsoft Application Console.

DISPLAY NAME	APPLICATION TYPE	APPLICATION ID
ws fedsample	Web app / API	adc50fba-9ac5-43cd-9d40-bb7...

FEDERATION METADATA DOCUMENT
https://login.microsoftonline.com/39afe...

WS-FEDERATION SIGN-ON ENDPOINT
https://login.microsoftonline.com/39af...

SAML-P SIGN-ON ENDPOINT

- 导航到新的应用程序注册。单击设置 > 属性并记下应用程序 ID URI。这是 WS 联合身份验证中间件的 `Wtrealm`：

为 ASP.NET 核心标识的外部登录提供程序中添加 WS 联合身份验证

- 添加的依赖项 `Microsoft.AspNetCore.Authentication.WsFederation` 到项目。
- 添加 WS 联合身份验证到 `Configure` 中的方法 `Startup.cs`:

```

services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationContext>()
    .AddDefaultTokenProviders();

services.AddAuthentication()
    .AddWsFederation(options =>
{
    // MetadataAddress represents the Active Directory instance used to authenticate users.
    options.MetadataAddress = "https://<ADFS FQDN or AAD tenant>/FederationMetadata/2007-
    06/FederationMetadata.xml";

    // Wtrealm is the app's identifier in the Active Directory instance.
    // For ADFS, use the relying party's identifier, its WS-Federation Passive protocol URL:
    options.Wtrealm = "https://localhost:44307/";

    // For AAD, use the App ID URI from the app registration's Properties blade:
    options.Wtrealm = "https://wsfedsample.onmicrosoft.com/bf0e7e6d-056e-4e37-b9a6-2c36797b9f01";
});

services.AddMvc()
// ...

```

注意: 调用 `AddIdentity` 配置的默认方案设置。`AddAuthentication(string defaultScheme)` 重载集 `DefaultScheme` 属性; 并且, `AddAuthentication(Action<AuthenticationOptions> configureOptions)` 重载设置仅显式设置的属性。其中的任一重载应只能调用一次时添加多个身份验证提供程序。对它的后续调用也可能会覆盖任何以前配置的 `AuthenticationOptions` 属性。

使用 WS 联合身份验证登录

浏览到应用程序并单击 `登录` nav 标头中的链接。没有用于登录 WsFederation 选项:

Log in

Use a local account to log in.

Email

WsFederation

Password

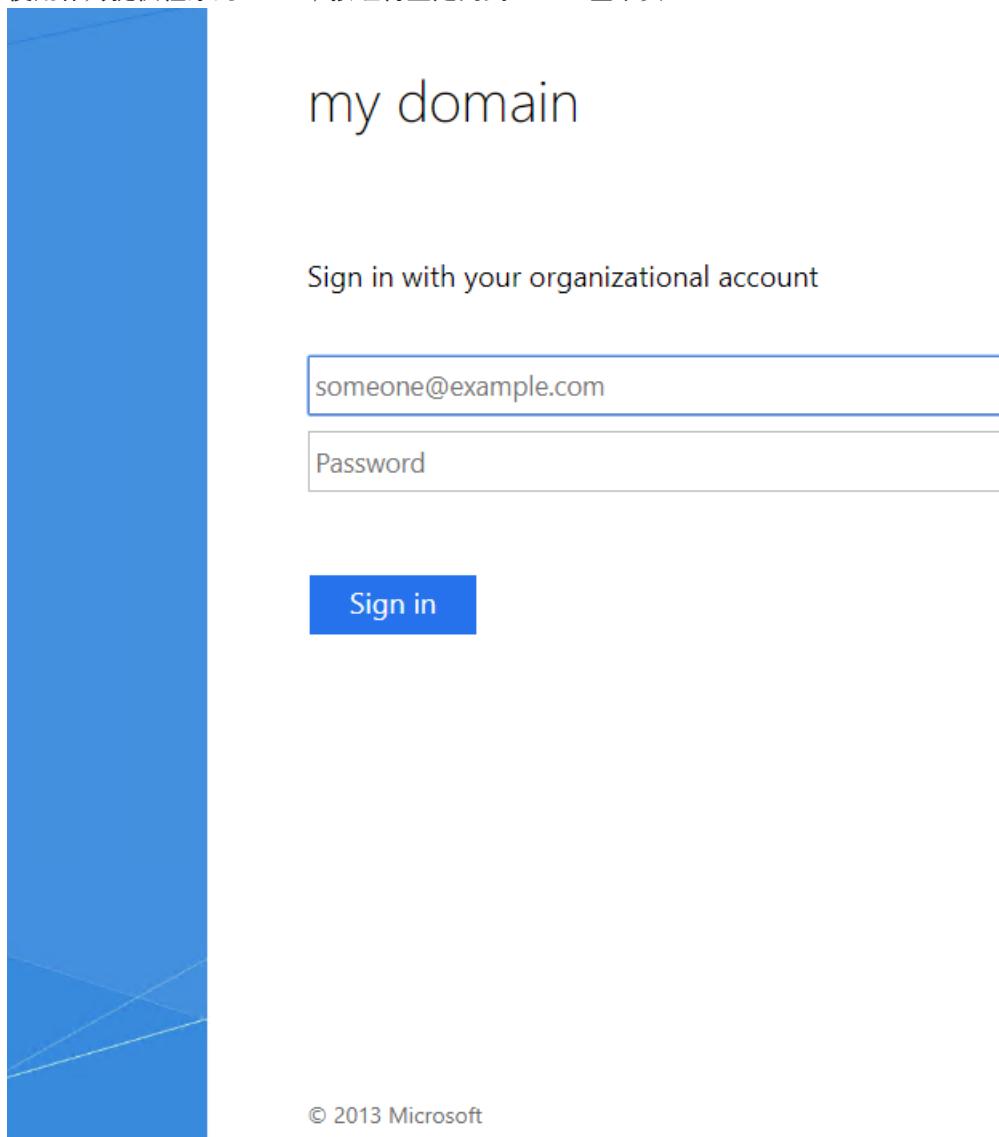
Remember me?

[Forgot your password?](#)

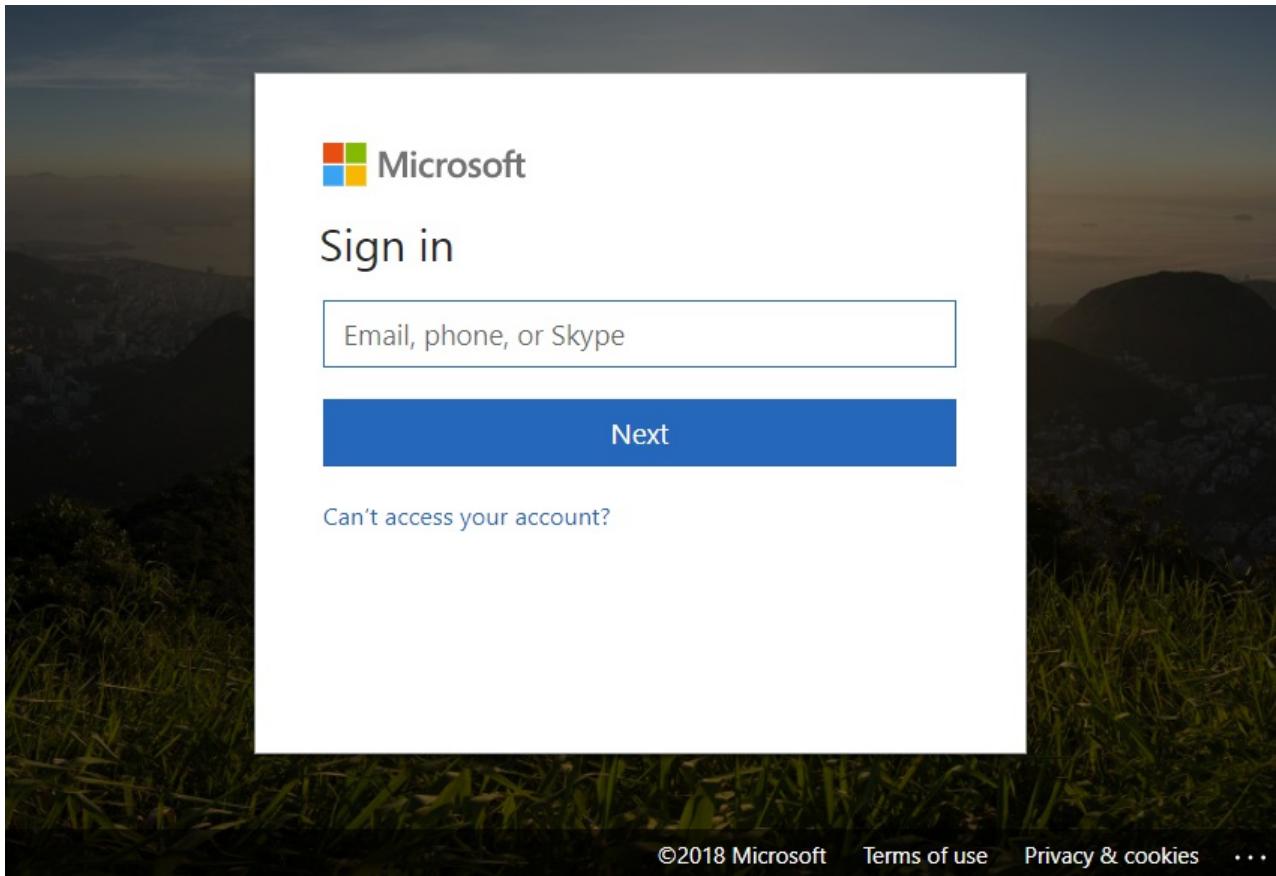
[Register as a new user](#)

Use another service to log in.

使用作为提供程序的 ADFS，按钮将重定向到 ADFS 登录页：



与 Azure Active Directory 作为提供程序，该按钮将重定向到 AAD 登录页：



成功登录的新用户将重定向到应用程序的用户注册页：



Register

Associate your WsFederation account.

You've successfully authenticated with **WsFederation**. Please enter an email address for this site below and click the Register button to finish logging in.

Email

[Register](#)

使用 WS 联合身份验证而无需 ASP.NET 核心标识

没有标识，可以使用 WS 联合身份验证中间件。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme = WsFederationDefaults.AuthenticationScheme;
    })
    .AddWsFederation(options =>
    {
        options.Wtrealm = Configuration["wsfed:realm"];
        options.MetadataAddress = Configuration["wsfed:metadata"];
    })
    .AddCookie();
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
    // ...
}
```

帐户确认和 ASP.NET Core 中的密码恢复

2018/5/17 • 13 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Joe Audette](#)

本教程演示了如何生成具有电子邮件确认及密码重置的 ASP.NET Core 应用。本教程是不开头主题。你应熟悉:

- [ASP.NET Core](#)
- [身份验证](#)
- [帐户确认和密码恢复](#)
- [Entity Framework Core](#)

请参阅[此 PDF 文件](#)的 ASP.NET 核心 MVC 1.1 和 2.x 版本。

系统必备

Install **one** of the following:

- CLI tooling: Windows, Linux, or macOS: [.NET Core SDK 2.0 or later](#)
- IDE/editor tooling
 - Windows: [Visual Studio for Windows](#)
 - **ASP.NET and web development** workload
 - **.NET Core cross-platform development** workload
 - Linux: [Visual Studio Code](#)
 - macOS: [Visual Studio for Mac](#)

使用.NET 核心 CLI 创建新的 ASP.NET Core 项目

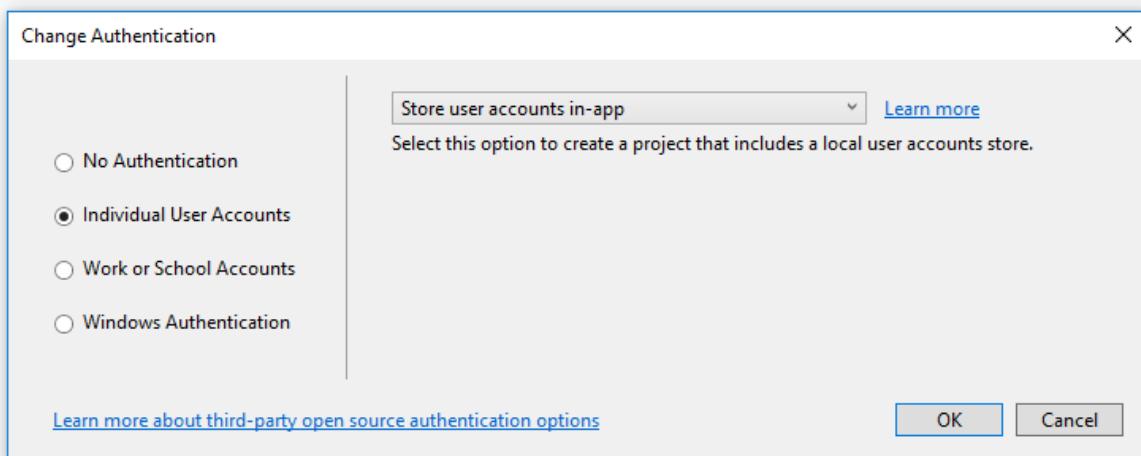
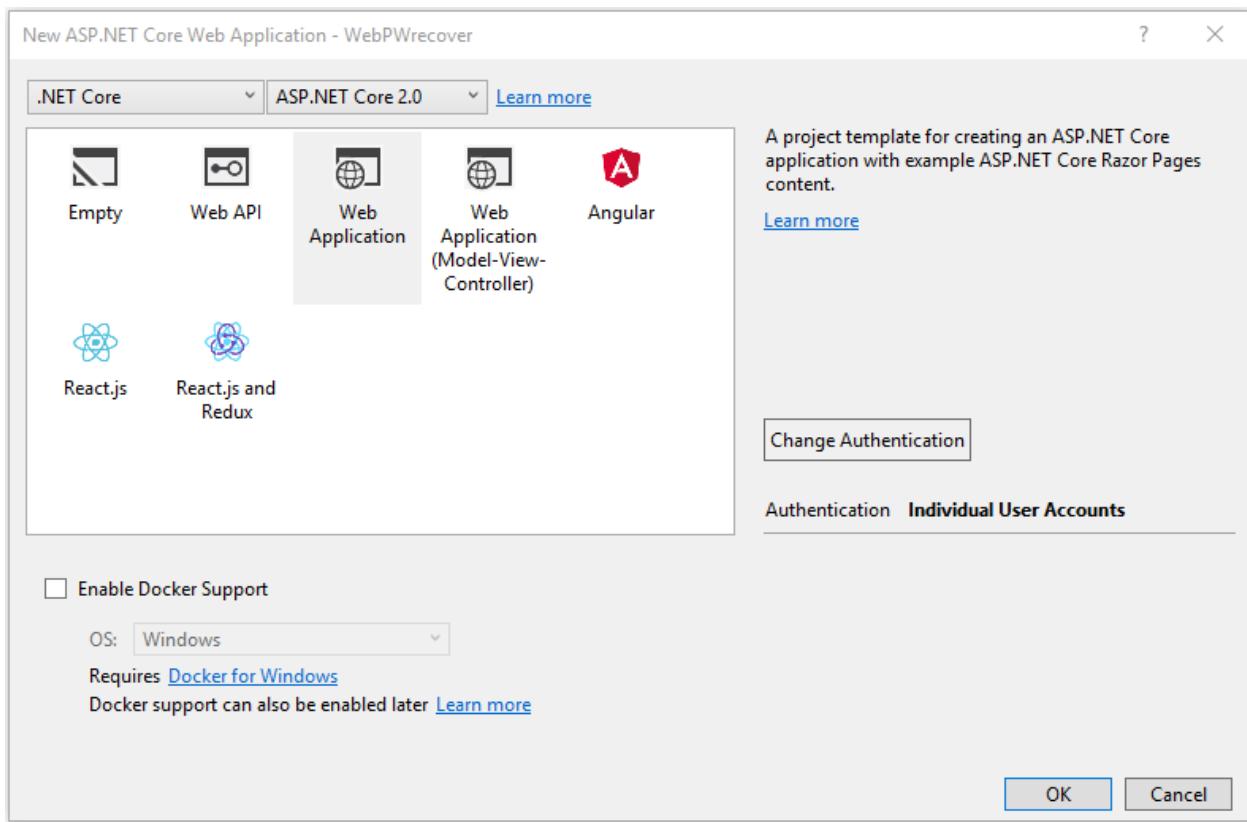
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
dotnet new razor --auth Individual -o WebPWrecover  
cd WebPWrecover
```

- `--auth Individual` 指定的单个用户帐户项目模板。
- 在 Windows 上, 添加 `-uld` 选项。它指定应而不是 SQLite 使用 LocalDB。
- 运行 `new mvc --help` 以获取有关此命令帮助。

或者, 你可以使用 Visual Studio 创建新的 ASP.NET Core 项目:

- 在 Visual Studio 中, 创建一个新**Web 应用程序**项目。
- 选择**ASP.NET Core 2.0**。**.NET 核心**选择在下图中, 但你可以选择**.NET Framework**。
- 选择**更改身份验证**并将设置为单个用户帐户。
- 保留默认值**存储用户帐户在应用程序**。



测试新的用户注册

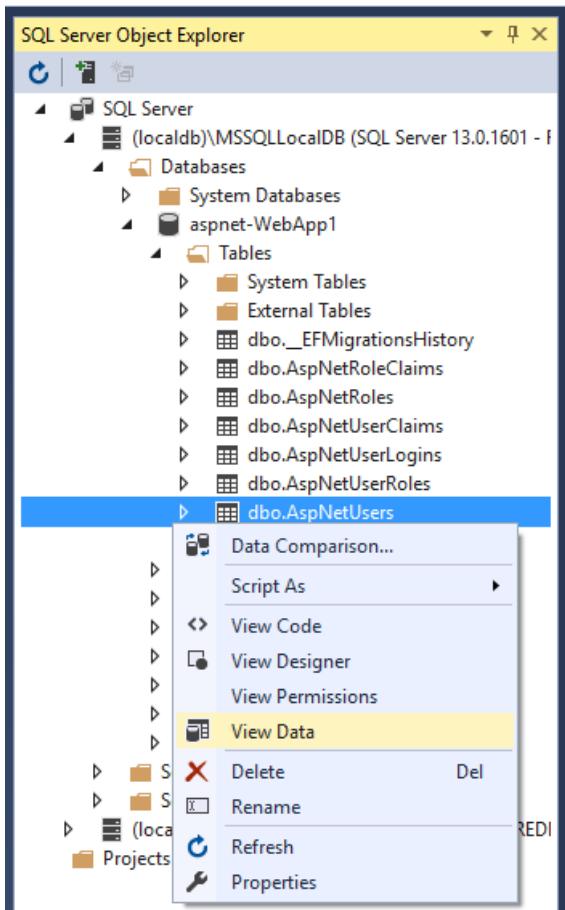
运行应用程序中，选择[注册](#)链接，并注册用户。按照运行实体框架核心迁移的说明。此时，电子邮件的唯一验证是使用[\[EmailAddress\]](#)属性。提交注册之后，你登录到应用程序。更高版本在教程中，这样，直到其电子邮件已验证新的用户无法登录时将更新代码。

查看标识数据库

请参阅[ASP.NET 核心 MVC 项目中使用的 SQLite](#)有关说明如何查看 SQLite 数据库。

Visual studio:

- 从视图菜单上，选择**SQL Server 对象资源管理器**(SSO)。
- 导航到 (**localdb**) **MSSQLLocalDB (SQL Server 13)**。右键单击**dbo**。**AspNetUsers** > 查看数据：



请注意表的 `EmailConfirmed` 字段是 `False`。

你可能想要再次在下一步中使用此电子邮件，当应用程序发送确认电子邮件。右键单击行并选择删除。删除的电子邮件别名便于您在以下步骤。

需要 HTTPS

请参阅[需要 HTTPS](#)。

需要电子邮件确认

它是一种最佳做法以确认新的用户注册的电子邮件。电子邮件确认有助于验证它们不模拟其他人（即，它们尚未注册与其他人的电子邮件）。假设有论坛，并且你想阻止"yli@example.com"发件人将注册为"nolivetto@contoso.com"。而无需电子邮件确认"nolivetto@contoso.com"无法接收来自您的应用程序不需要的电子邮件。假设用户意外注册为"ylo@example.com"和未注意到"yli"的拼写错误。它们将无法使用恢复密码，因为此应用程序没有正确的电子邮件。电子邮件确认从机器人提供有限的保护。电子邮件确认不免受恶意用户的多个电子邮件帐户与提供保护。

你通常想要使不能在有确认电子邮件之前发布到网站的任何数据的新用户。

更新 `ConfigureServices` 需要确认电子邮件：

```

public void ConfigureServices(IServiceCollection services)
{
    // Requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });

    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
    })
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.AddMvc()
        .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizeFolder("/Account/Manage");
        options.Conventions.AuthorizePage("/Account/Logout");
    });
}

```

`config.SignIn.RequireConfirmedEmail = true;` 防止已注册的用户登录，直到其电子邮件进行确认。

配置电子邮件提供商

在本教程中，使用 SendGrid 发送电子邮件。你需要一个 SendGrid 帐户和密钥用于发送电子邮件。你可以使用其他电子邮件提供商。ASP.NET 核心 2.x 包括 `System.Net.Mail`，这允许你从你的应用程序发送电子邮件。我们建议你使用 SendGrid 或另一个电子邮件服务发送电子邮件。SMTP 很难保护并正确设置。

[选项模式](#)用于访问的用户帐户和密钥设置。有关详细信息，请参阅[配置](#)。

创建一个类以提取安全的电子邮件密钥。对于此示例，`AuthMessageSenderOptions` 中创建类`Services/AuthMessageSenderOptions.cs`文件：

```

public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}

```

设置 `SendGridUser` 和 `SendGridKey` 与[机密管理器工具](#)。例如：

```
C:\WebApp1\src\WebApp1>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

在 Windows 上，密钥管理器存储中的键/值对`secrets.json`文件中

`%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` 目录。

内容`secrets.json`文件未加密。`Secrets.json`文件如下所示（`SendGridKey` 值已删除。）

```
{
    "SendGridUser": "RickAndMSFT",
    "SendGridKey": "<key removed>"
}
```

配置启动要使用 `AuthMessageSenderOptions`

添加 `AuthMessageSenderOptions` 到末尾的服务容器 `ConfigureServices` 中的方法 `Startup.cs` 文件：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public void ConfigureServices(IServiceCollection services)
{
    // Requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });

    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
    })
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.AddMvc()
        .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizeFolder("/Account/Manage");
        options.Conventions.AuthorizePage("/Account/Logout");
    });
}

services.AddSingleton<IEmailSender, EmailSender>();

services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

配置 `AuthMessageSender` 类

本教程演示如何添加通过电子邮件通知 [SendGrid](#)，但是你可以发送电子邮件使用 SMTP 和其他机制。

安装 `SendGrid` NuGet 包：

- 从命令行：

```
dotnet add package SendGrid
```

- 从程序包管理器控制台中，输入以下命令：

```
Install-Package SendGrid
```

请参阅 [免费开始使用 SendGrid](#) 注册一个免费的 SendGrid 帐户。

配置了 `SendGrid`

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要配置 `SendGrid`，添加类似于以下的代码 `Services/EmailSender.cs`：

```

using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebPWrecover.Services
{
    public class EmailSender : IEmailSender
    {
        public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            return Execute(Options.SendGridKey, subject, message, email);
        }

        public Task Execute(string apiKey, string subject, string message, string email)
        {
            var client = new SendGridClient(apiKey);
            var msg = new SendGridMessage()
            {
                From = new EmailAddress("Joe@contoso.com", "Joe Smith"),
                Subject = subject,
                PlainTextContent = message,
                HtmlContent = message
            };
            msg.AddTo(new EmailAddress(email));
            return client.SendEmailAsync(msg);
        }
    }
}

```

启用帐户确认和密码恢复

已为帐户确认和密码恢复代码的模板。查找 `OnPostAsync` 中的方法 `Pages/Account/Register.cshtml.cs`。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

可防止新注册的用户自动登录通过注释掉以下行：

```
await _signInManager.SignInAsync(user, isPersistent: false);
```

突出显示的已更改行还会显示完整的方法：

```

public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.EmailConfirmationLink(user.Id, code, Request.Scheme);
            await _emailSender.SendEmailConfirmationAsync(Input.Email, callbackUrl);

            // await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(Url.GetLocalUrl(returnUrl));
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

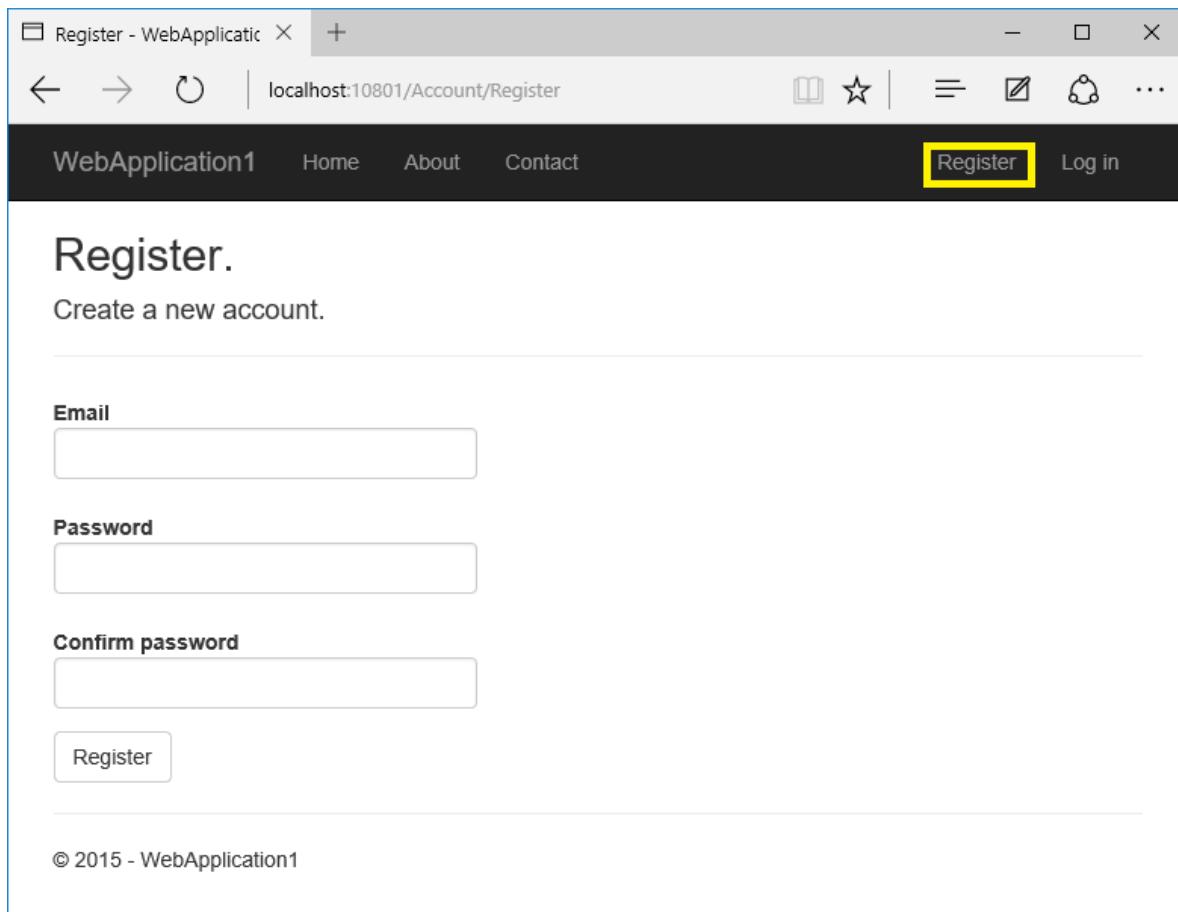
    // If we got this far, something failed, redisplay form
    return Page();
}

```

注册、确认电子邮件，以及重置密码

运行该 web 应用，并测试帐户确认和密码恢复流。

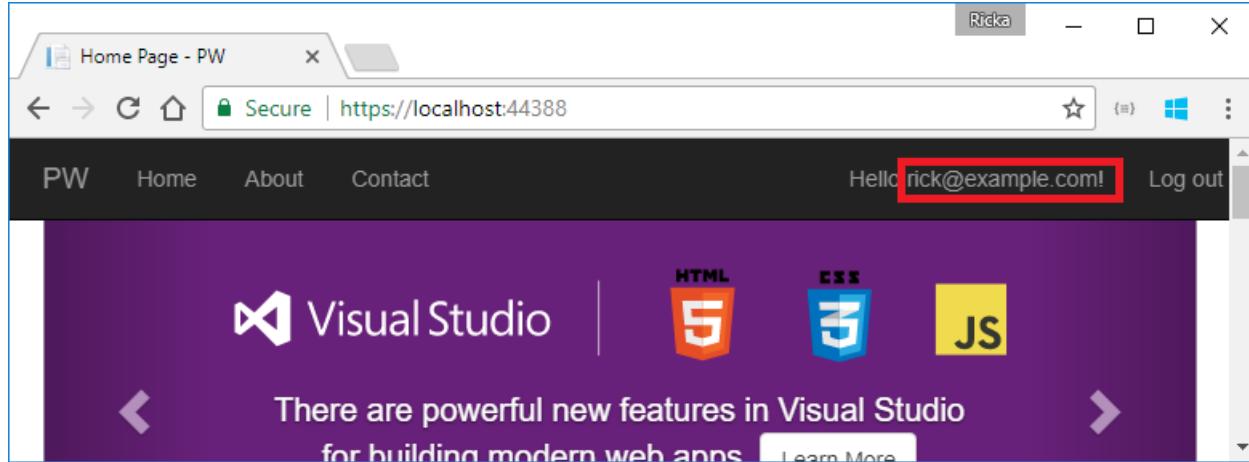
- 运行应用并注册新的用户



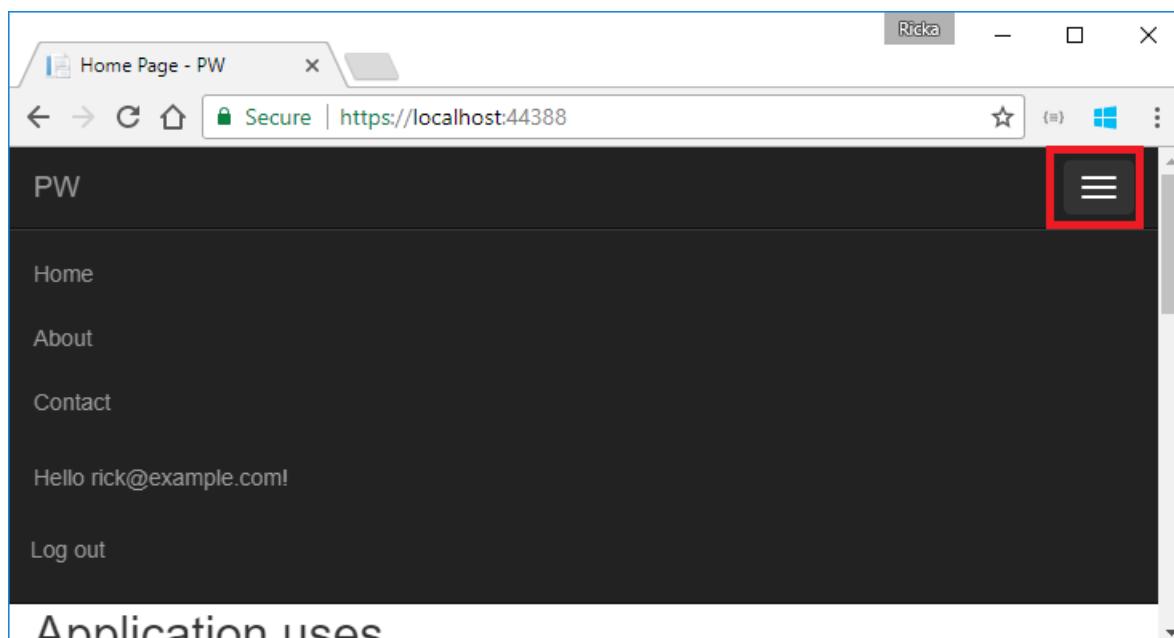
- 检查你的帐户确认链接的电子邮件。请参阅[调试电子邮件](#)如果你不会获得电子邮件。
- 单击链接以确认你的电子邮件。
- 登录你的电子邮件和密码。
- 注销。

查看管理页

在浏览器中选择你的用户名：



你可能需要展开导航栏，以查看用户名。



- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

管理页显示与[配置文件](#)选定选项卡。电子邮件显示一个复选框，表明电子邮件已确认。

The screenshot shows a web browser window titled "Profile - WebPW". The address bar indicates a secure connection to <https://localhost:44319/Manage/Index>. The top navigation bar includes links for "Home", "About", and "Contact", along with a user greeting "Hello rick@example.com!" and a "Log out" link. The main content area has a blue header bar labeled "Profile". Below it, there are links for "Password" and "Two-factor authentication". The "Profile" section is expanded, showing fields for "Username" (set to "rick@example.com"), "Email" (set to "rick@example.com" with a green checkmark icon), and "Phone number" (an empty input field). A "Save" button is located below these fields. At the bottom of the page, a copyright notice reads "© 2017 - WebPW".

测试密码重置

- 如果你要登录，选择**注销**。
- 选择**登录链接并选择忘记了密码？**链接。
- 输入用于注册帐户的电子邮件。
- 发送一封电子邮件包含要重置密码的链接。请检查你的电子邮件，单击链接以重置密码。你的密码已成功重置后，你可以登录你的电子邮件和新密码。

调试电子邮件

如果无法获取电子邮件工作：

- 创建**控制台应用程序发送电子邮件**。
- 查看**电子邮件活动页**。
- 请检查垃圾邮件文件夹。
- 请尝试其他电子邮件提供程序（Microsoft、Yahoo、Gmail 等）上的另一个电子邮件别名
- 尝试发送到不同的电子邮件帐户。

最佳安全方案是不使用生产中测试和开发的机密。如果将应用发布到 Azure，你可以设置 SendGrid 机密与 Azure Web 应用门户中的应用程序设置。配置系统设置以从环境变量中读取项。

合并社交和本地登录帐户

若要完成此部分，必须先启用外部身份验证提供程序。请参阅[Facebook、Google、和外部提供程序身份验证](#)。

你可以通过单击电子邮件链接组合本地和社交帐户。按以下顺序"RickAndMSFT@gmail.com"首先创建为本地登录名；但是，你可以创建帐户作为的社交登录名，然后再添加本地登录名。

The screenshot shows a web browser window titled "Home Page - Web1". The address bar displays "localhost:1234". The top navigation bar includes links for "Web1", "Home", "About", "Contact", and a user greeting "Hello rickandmsft@gmail.com!" which is highlighted with a yellow box. Below the navigation bar, there's a main content area with a blue header containing the text "ASP.NET 5 | Windows | Linux | OSX". A sub-section below it says "Learn how to build ASP.NET apps that can run anywhere." with a "Learn More" button. The main content area is titled "Application uses" and lists three bullet points: "Sample pages using ASP.NET 5 (MVC 6)", "Gulp and Bower for managing client-side resources", and "Theming using Bootstrap". Another section titled "New concepts" is partially visible at the bottom.

单击[管理](#)链接。请注意与此帐户关联 0 外部 (社交登录名)。

The screenshot shows a web browser window titled "Manage your account - Web1". The address bar displays "localhost:1234/Manage". The top navigation bar includes links for "Web1", "Home", "About", "Contact", and a user greeting "Hello rickandmsft@gmail.com!". Below the navigation bar, the main content area is titled "Manage your account." and "Change your account settings". It contains several configuration sections: "Password" with a "[Change]" link, "External Logins" with a red box around the "0 [Manage]" link which is also highlighted with a red box, "Phone Number" with a note about two-factor authentication, and "Two-Factor Authentication" with a note about no providers configured. At the bottom, there's a copyright notice "© 2015 - Web1".

单击另一个登录服务的链接，接受应用程序请求。下图中，在 Facebook 是外部身份验证提供程序：

Manage your external logins.

Registered Logins

Facebook

© 2017 - WebApplication2

已经合并两个帐户。现在可以使用任一帐户登录。你可能希望添加本地帐户，以防其社交登录身份验证服务已关闭，或更有可能已丢失到其社交帐户访问你用户。

启用帐户确认后某个站点具有用户

启用站点上与用户的帐户确认锁定更改所有现有用户。现有用户被锁定，因为他们的帐户不确认。若要解决退出用户锁定，请使用以下方法之一：

- 更新数据库，将标记为正在确认的所有现有用户
- 确认正在退出用户。例如，批处理-发送电子邮件中的标注确认链接。

启用 ASP.NET Core 中的身份验证器应用的 QR 代码生成

2018/4/10 • 2 min to read • [Edit Online](#)

注意：本主题适用于 ASP.NET Core 2.x

ASP.NET 核心附带的单个身份验证的身份验证器应用程序的支持。两个因素身份验证 (2FA) 身份验证器应用，使用基于时间的一次性密码算法 (TOTP)，是推荐的方法为 2FA 行业。2FA 使用 TOTP 优于 SMS 2FA。验证器应用提供哪些用户确认其用户名和密码后，必须输入一个 6 到 8 位代码。通常在智能手机上安装验证器应用。

ASP.NET 核心 web 应用程序模板支持身份验证器，但不提供对 QRCode 生成的支持。QRCode 生成器轻松地 2FA 的安装程序。本文档将指导你完成添加QR 代码生成到 2FA 配置页。

将 QR 代码添加到 2FA 配置页

这些说明使用`qrcode.js`从<https://davidshimjs.github.io/qrcodejs/>存储库。

- 下载`qrcode.js` javascript 库到 `wwwroot\lib` 项目文件夹中的。
- 在`Pages\Account\Manage\EnableAuthenticator.cshtml` (Razor 页)
或`Views\Manage\EnableAuthenticator.cshtml` (MVC)、找到 `Scripts` 文件末尾的部分：

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- 更新 `Scripts` 部分添加到引用 `qrcode.js` 你添加的库和生成的 QR 代码的调用。其外观应，如下所示：

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript">
        new QRCode(document.getElementById("qrCode"),
        {
            text: "@Html.Raw(Model.AuthenticatorUri)",
            width: 150,
            height: 150
        });
    </script>
}
```

- 删除的段落，您链接到这些说明。

运行你的应用，并确保你可以扫描 QR 代码和验证身份验证器证明的代码。

更改 QR 代码中的站点名称

最初创建你的项目时选择的项目名称中获取 QR 代码中的站点名称。你可以通过查找对其进行更改

`GenerateQrCodeUri(string email, string unformattedKey)` 中的方

法`Pages\Account\Manage\EnableAuthenticator.cshtml.cs` (Razor 页) 文件或`Controllers\ManageController.cs` (MVC) 文件。

从模板的默认代码将如下所示：

```
private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenticatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

第二个参数的调用中 `string.Format` 是你的站点名称，取自解决方案名称。它可以更改为任何值，但它必须始终为 URL 编码。

使用不同的 QR 代码库

QR 代码库可以替换你首选的库。HTML 包含 `qrCode` 元素在其中可以通过任何机制将 QR 代码你的库提供。

QR 代码的格式正确 URL 可用于：

- `AuthenticatorUri` 模型的属性。
- `data-url` 中的属性 `qrCodeData` 元素。

TOTP 客户端和服务器时间偏差

TOTP 身份验证依赖于服务器和身份验证器的设备具有精确的时间。仅在 30 秒内，最后一个令牌。如果未 TOTP 2FA 登录名，请检查服务器的时间比准确，并且最好是同步到准确 NTP 服务。

在 ASP.NET Core SMS 的双因素身份验证

2018/5/14 • 6 min to read • [Edit Online](#)

通过[Rick Anderson](#)和[瑞士开发人员](#)

请参阅[ASP.NET Core 中的身份验证器应用启用 QR 代码生成](#)ASP.NET 核心 2.0 及更高版本。

本教程演示如何设置双因素身份验证 (2FA) 使用短信。为提供的说明[twilio](#)和[ASPSMS](#)，但你可以使用任何其他 SMS 提供程序。我们建议你完成[帐户确认和密码恢复](#)之前开始学习本教程。

视图[已完成的示例](#)。[如何下载](#)。

创建新的 ASP.NET Core 项目

创建新的 ASP.NET 核心 web 应用名为 `Web2FA` 与单个用户帐户。按照中的说明[在 ASP.NET Core 应用程序强制实施 SSL 才能设置](#)，并且需要 SSL。

创建 SMS 帐户

创建 SMS 帐户，例如，从[twilio](#)或[ASPSMS](#)。记录身份验证凭据 (twilio: accountSid 和 authToken 的 ASPSMS：用户密钥和密码)。

了解 SMS 提供程序凭据

Twilio:

从你的 Twilio 帐户的仪表板选项卡上，复制帐户 **SID**和**身份验证令牌**。

ASPSMS:

从你的帐户设置，导航到**用户密钥**并将其连同**复制你密码**。

我们将更高版本将存储在密钥中的密钥管理器工具使用这些值 `SMSAccountIdentification` 和 `SMSAccountPassword`。

指定 `SenderId` / 发起方

Twilio:

从数字选项卡，将复制你的 Twilio电话号码。

ASPSMS:

解锁原始发件人菜单上，在解锁一个或多个发送方或选择（不支持的所有网络）的字母数字发起方。

我们将更高版本存储在项的密钥管理器工具使用此值 `SMSAccountFrom`。

SMS 服务提供的凭据

我们将使用[选项模式](#)访问的用户帐户和密钥设置。

- 创建一个类以提取安全 SMS 密钥。对于此示例，`SMSOptions` 中创建类 `Services/SMSOptions.cs` 文件。

```
namespace Web2FA.Services
{
    public class SMSOptions
    {
        public string SMSAccountIdentification { get; set; }
        public string SMSAccountPassword { get; set; }
        public string SMSAccountFrom { get; set; }
    }
}
```

设置 `SMSAccountIdentification`，`SMSAccountPassword` 和 `SMSAccountFrom` 与[机密管理器工具](#)。例如：

```
C:/Web2FA/src/WebApp1>dotnet user-secrets set SMSAccountIdentification 12345
info: Successfully saved SMSAccountIdentification = 12345 to the secret store.
```

- SMS 提供程序添加 NuGet 包。从包管理器控制台 (PMC) 运行：

Twilio:

```
Install-Package Twilio
```

ASPSMS:

```
Install-Package ASPSMS
```

- 中添加代码*Services/MessageServices.cs*文件以启用 SMS。使用 Twilio 或 ASPSMS 部分：

Twilio:

```
using Microsoft.Extensions.Options;
using System.Threading.Tasks;
using Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSOptions Options { get; } // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            // Plug in your SMS service here to send a text message.
            // Your Account SID from twilio.com/console
            var accountSid = Options.SMSAccountIdentification;
            // Your Auth Token from twilio.com/console
            var authToken = Options.SMSAccountPassword;

            TwilioClient.Init(accountSid, authToken);

            return MessageResource.CreateAsync(
                to: new PhoneNumber(number),
                from: new PhoneNumber(Options.SMSAccountFrom),
                body: message);
        }
    }
}
```

ASPSMS:

```

using Microsoft.Extensions.Options;
using System.Threading.Tasks;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSOptions Options { get; } // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            ASPSMS.SMS SMSSender = new ASPSMS.SMS();

            SMSSender.Userkey = Options.SMSAccountIdentification;
            SMSSender.Password = Options.SMSAccountPassword;
            SMSSender.Originator = Options.SMSAccountFrom;

            SMSSender.AddRecipient(number);
            SMSSender.MessageData = message;

            SMSSender.SendTextSMS();

            return Task.FromResult(0);
        }
    }
}

```

配置要使用的启动 `SMSOptions`

添加 `SMSOptions` 对中的服务容器 `ConfigureServices` 中的方法 `Startup.cs`:

```

// Add application services.
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
services.Configure<SMSOptions>(Configuration);
}

```

启用双因素身份验证

打开 `Views/Manage/Index.cshtml` Razor 视图文件并删除注释字符（因此没有标记出 `commented`）。

使用双因素身份验证登录

- 运行应用并注册新的用户

Register - WebApplication1

localhost:44300

Register.

Create a new account.

Email
joe@contoso.com

Password

Confirm password

Register

© 2015 - WebApplication1

- 在你激活的用户名上点击 Index 管理控制器中的操作方法。然后点击的电话号码添加链接。

Manage your account -

localhost:44300/Manage

Hello joe@contoso.com! Log off

Manage your account.

Change your account settings

Password: [Change]
External Logins: 0 [Manage]
Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.
None [Add]
Two-Factor Authentic... Disabled [Enable]

© 2015 - WebApplication1

- 添加一个电话号码，它将接收验证码，并点击发送验证码。

The screenshot shows a web browser window with the title "Add Phone Number - W". The address bar displays "localhost:44300/Manage". The page content is titled "Add Phone Number." and contains the instruction "Add a phone number." Below this is a form field labeled "Phone number" containing the value "206-555-1234". A button labeled "Send verification code" is visible. At the bottom of the page is a copyright notice: "© 2015 - WebApplication1".

- 你将获取验证码短信。输入它，然后点击提交

The screenshot shows a web browser window with the title "Verify Phone Number - We...". The address bar displays "https://localhost:44300/Manage/VerifyPhoneNumber?PhoneNumber=206-555-1234". The page content is titled "Verify Phone Number." and contains the instruction "Add a phone number." Below this is a form field labeled "Code" containing the value "5879". A button labeled "Submit" is visible. At the bottom of the page is a copyright notice: "© 2015 - WebApplication1".

如果你不会获得文本消息，请参阅 twilio 日志页。

- 管理视图显示你的电话号码已成功添加。

Manage your account.

Your phone number was added.

Change your account settings

Password: [Change]

External Logins: 0 [Manage]

Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.

206-555-1234 [Change | Remove]

Two-Factor Authentic... Disabled [Enable](#)

© 2015 - WebApplication1

- 点击启用启用双因素身份验证。

Manage your account.

Your phone number was added.

Change your account settings

Password: [Change]

External Logins: 0 [Manage]

Phone Number: Phone Numbers can be used as a second factor of verification in two-factor authentication. See [this article](#) for details on setting up this ASP.NET application to support two-factor authentication using SMS.

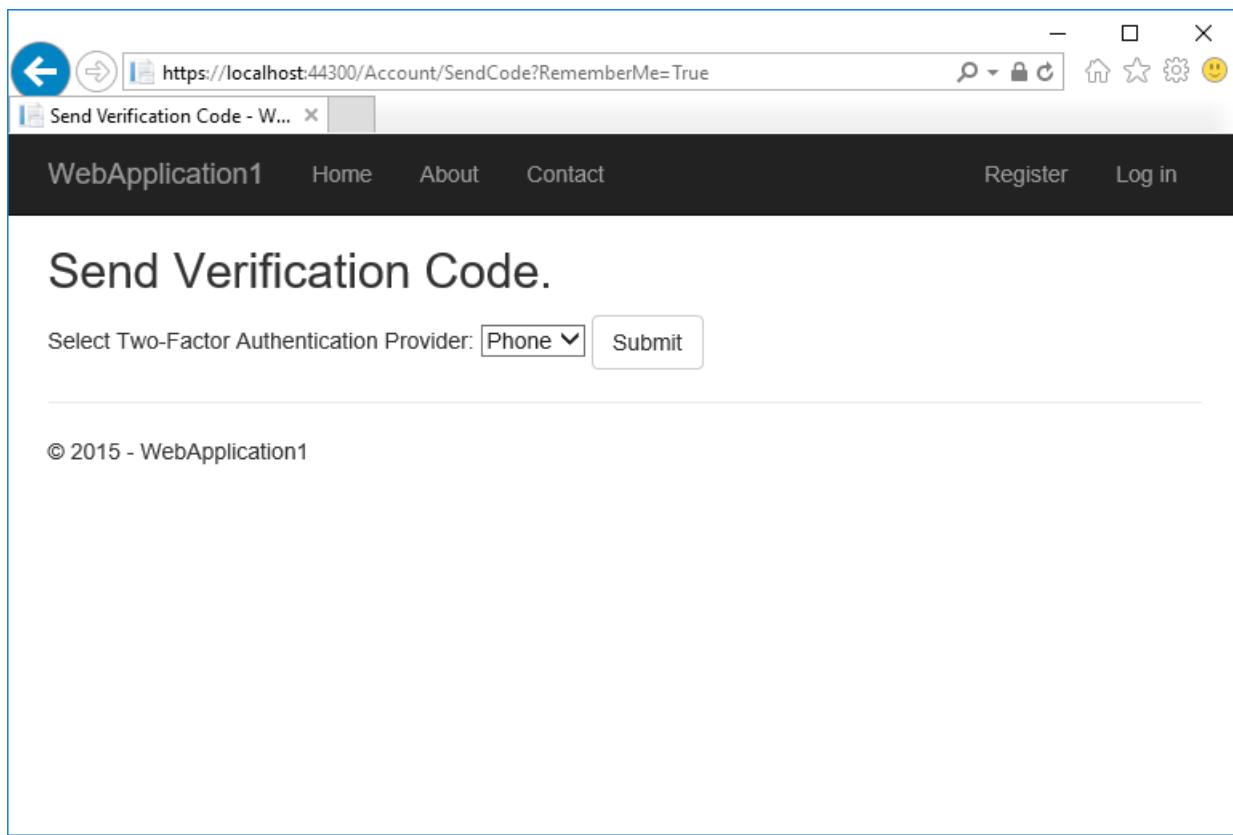
206-555-1234 [Change | Remove]

Two-Factor Authentic... Disabled [Enable](#)

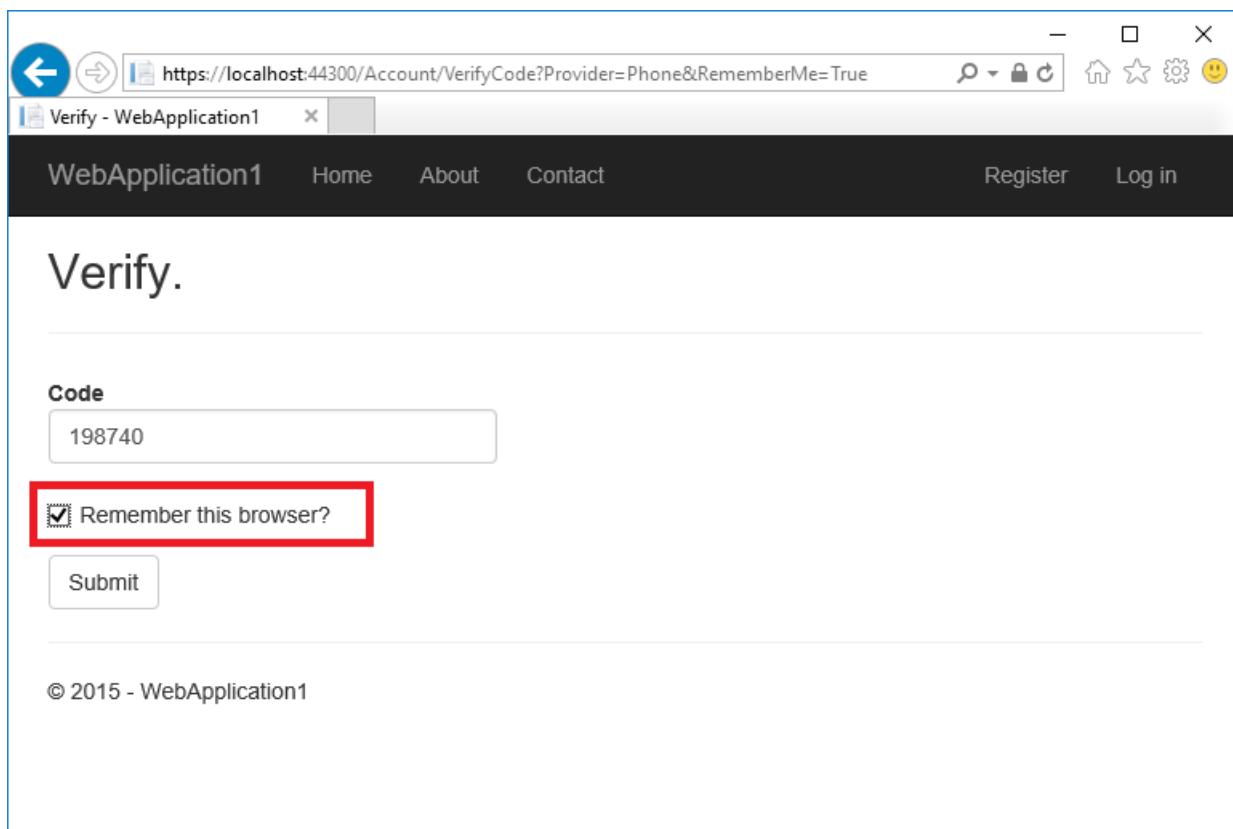
© 2015 - WebApplication1

测试两个双因素身份验证

- 注销。
- 登录。
- 用户帐户已启用双因素身份验证，因此你需要提供身份验证的第二个因素。在本教程中已启用电话验证。内置的模板还可用于将作为第二因素的电子邮件设置。你可以设置其他的第二个因素，如 QR 代码的身份验证。点击提交。



- 输入中的 SMS 消息得到的代码。
- 单击记住此浏览器复选框将免除你无需使用 2FA 时使用相同的设备和浏览器登录。启用 2FA 和单击记住此浏览器将为你提供强 2FA 保护免受恶意用户尝试访问你的帐户，只要它们无权访问你的设备。可以在任何您经常使用的专用设备上执行此操作。通过设置记住此浏览器，设备不经常使用的情况下，从获取 2FA 的附加的安全性以及你可以方便地在无需在你自己的设备通过 2FA 转。



针对暴力破解攻击提供保护的帐户锁定

帐户锁定建议的 2FA 中。一旦用户通过本地帐户或社交帐户登录，则会存储 2FA 在每次失败的尝试。如果达到最

大未成功的访问，则用户锁定（默认：5分钟锁定5失败的访问尝试次数后）。成功的身份验证将失败的访问尝试计数重置并重置时钟。最大失败的访问尝试次数，可以用来设置的锁定时间

MaxFailedAccessAttempts和DefaultLockoutTimeSpan。以下10分钟后访问尝试失败10次配置帐户锁定：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSOptions>(Configuration);
}
```

确认 PasswordSignInAsync 设置 lockoutOnFailure 到 true：

```
var result = await _signInManager.PasswordSignInAsync(
    Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);
```

使用 cookie 而无需 ASP.NET 核心标识的身份验证

2018/5/17 • 18 min to read • [Edit Online](#)

通过[Rick Anderson](#)和[Luke Latham](#)

如你所见在早期的身份验证主题中，[ASP.NET 核心标识](#)完成、功能完备的身份验证提供程序是用于创建和维护登录名。但是，你可能想要使用基于 cookie 的身份验证有时使用您自己的自定义身份验证逻辑。你可以使用基于 cookie 的身份验证作为独立身份验证提供程序，没有 ASP.NET 核心标识。

[查看或下载示例代码 \(如何下载\)](#)

有关从 ASP.NET Core 迁移基于 cookie 的身份验证信息 1.x 到 2.0，请参阅[迁移身份验证和标识到 ASP.NET 核心 2.0 主题 \(基于 Cookie 的身份验证\)](#)。

配置

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

如果你未使用[Microsoft.AspNetCore.All metapackage](#)，安装 2.0 以上版本的[Microsoft.AspNetCore.Authentication.Cookies](#) NuGet 包。

在 `ConfigureServices` 方法，创建具有身份验证中间件服务 `AddAuthentication` 和 `AddCookie` 方法：

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

`AuthenticationScheme` 传递给 `AddAuthentication` 设置应用程序的默认身份验证方案。`AuthenticationScheme` 当有多个实例的 cookie 身份验证，并且希望时非常有用[授权与特定方案](#)。设置 `AuthenticationScheme` 到 `CookieAuthenticationDefaults.AuthenticationScheme` 方案提供的"Cookie"的值。你可以提供任何字符串值，用于区分方案。

在 `Configure` 方法，请使用 `UseAuthentication` 方法来调用设置身份验证中间件 `HttpContext.User` 属性。调用 `UseAuthentication` 方法之前调用 `UseMvcWithDefaultRoute` 或 `UseMvc`：

```
app.UseAuthentication();
```

AddCookie 选项

`CookieAuthenticationOptions` 类用于配置身份验证提供程序选项。

选项	描述
<code>AccessDeniedPath</code>	提供的路径以提供 302 找到 (URL 重定向) 时触发的 <code>HttpContext.ForbidAsync</code> 。默认值为 <code>/Account/AccessDenied</code> 。
<code>ClaimsIssuer</code>	要用于颁发者 颁发者 cookie 身份验证服务创建的任何声明上的属性。

选项	描述
Cookie.Domain	Cookie 提供服务位置的域名。默认情况下，这是请求的主机名。浏览器仅将 cookie 在请求中发送到匹配的主机名。你可能希望调整这在你的域中具有可用于任何主机的 cookie。例如，将 cookie 域设置为 <code>.contoso.com</code> 使其可供 <code>contoso.com</code> 、 <code>www.contoso.com</code> 和 <code>staging.www.contoso.com</code> 。
Cookie.Expiration	获取或设置一个 cookie 的生存期。目前，此选项没有 ops，并且将会过时中 ASP.NET Core 2.1+。使用 <code>ExpireTimeSpan</code> 选项设置 cookie 到期时间。有关详细信息，请参阅 阐明 CookieAuthenticationOptions.Cookie.Expiration 行为 。
Cookie.HttpOnly	一个标志，指示该 cookie 应仅服务器访问。更改此值与 <code>false</code> 允许客户端脚本来访问 cookie 和可能打开你的应用 cookie 被盗。你的应用具备 跨站点脚本 (XSS) 漏洞。默认值为 <code>true</code> 。
Cookie.Name	设置的 cookie 的名称。
Cookie.Path	用于隔离在相同的主机名上运行的应用程序。如果必须在运行的应用程序 <code>/app1</code> 并且想要限制对该应用的 cookie，请将设置 <code>CookiePath</code> 属性 <code>/app1</code> 。通过这样做，cookie 是仅可在上找到对请求 <code>/app1</code> 和其下的任何应用程序。
Cookie.SameSite	该值指示浏览器应允许 cookie 要附加到同一站点的请求 (<code>SameSiteMode.Strict</code>) 或使用安全的 HTTP 方法和同一站点请求的跨站点请求 (<code>SameSiteMode.Lax</code>)。当设置为 <code>SameSiteMode.None</code> ，未设置的 cookie 标头值。请注意， Cookie 策略中间件 可能会覆盖你提供的值。若要支持 OAuth 身份验证，默认值是 <code>SameSiteMode.Lax</code> 。有关详细信息，请参阅 OAuth 身份验证由于 SameSite cookie 策略中断 。
Cookie.SecurePolicy	一个标志，指示是否创建的 cookie 应被限制为 HTTPS (<code>CookieSecurePolicy.Always</code>)，HTTP 或 HTTPS (<code>CookieSecurePolicy.None</code>)，或请求的同一个协议 (<code>CookieSecurePolicy.SameAsRequest</code>)。默认值为 <code>CookieSecurePolicy.SameAsRequest</code> 。
DataProtectionProvider	集 <code>DataProtectionProvider</code> 用于创建默认值 <code>TicketDataFormat</code> 。如果 <code>TicketDataFormat</code> 设置属性， <code>DataProtectionProvider</code> 选项未使用。如果未提供，则使用应用程序的默认数据保护提供程序。
事件	该处理程序在处理中的某些点提供的应用程序控制的提供程序上调用方法。如果 <code>Events</code> 不提供的默认实例提供的方法在调用时不执行任何操作。
EventsType	用作服务类型，以获取 <code>Events</code> 实例而不是属性。

选项	描述
ExpireTimeSpan	<code>TimeSpan</code> 后将存储在 cookie 的身份验证票证到期。 <code>ExpireTimeSpan</code> 将添加到当前时间来创建票证的到期时间。 <code>ExpiredTimeSpan</code> 值始终将验证由服务器加密 AuthTicket 进入。它可能还进入 <code>Set-cookie</code> 标头，但仅当 <code>IsPersistent</code> 设置。若要设置 <code>IsPersistent</code> 到 <code>true</code> ，配置 <code>AuthenticationProperties</code> 传递给 <code>SignInAsync</code> 。默认值 <code>ExpireTimeSpan</code> 为 14 天。
LoginPath	提供的路径以提供 302 找到 (URL 重定向) 时触发的 <code>HttpContext.ChallengeAsync</code> 。当前生成 401 的 URL 添加到 <code>LoginPath</code> 作为查询字符串参数由名为 <code>ReturnUrlParameter</code> 。一次请求 <code>LoginPath</code> 授予新登录标识， <code>ReturnUrlParameter</code> 值用于将浏览器重定向回导致原始的未授权的状态代码的 URL。默认值为 <code>/Account/Login</code> 。
LogoutPath	如果 <code>LogoutPath</code> 然后对该路径的请求重定向到处理程序，提供的值基于 <code>ReturnUrlParameter</code> 。默认值为 <code>/Account/Logout</code> 。
ReturnUrlParameter	确定追加通过处理程序 302 找到 (URL 重定向) 响应的查询字符串参数的名称。 <code>ReturnUrlParameter</code> 请求到达时，将使用 <code>LoginPath</code> 或 <code>LogoutPath</code> 以返回到原始的 URL 的浏览器之后执行的登录或注销操作。默认值为 <code>ReturnUrl</code> 。
SessionStore	用来存储标识跨请求可选容器。使用时，仅一个会话标识符是发送到客户端。 <code>SessionStore</code> 可以用于缓解的大型标识潜在问题。
SlidingExpiration	应动态发出一个标志，指示如果具有更新的到期时间的新 cookie。这可能发生在当前的 cookie 过期时间已超过 50% 过期任何请求。向前移动新的到期日期为当前日期加上 <code>ExpireTimespan</code> 。绝对 cookie 到期时间可以通过使用设置 <code>AuthenticationProperties</code> 类调用时 <code>SignInAsync</code> 。绝对到期时间可通过限制的身份验证 cookie 是有效的时间量来提高您的应用程序的安全性。默认值为 <code>true</code> 。
TicketDataFormat	<code>TicketDataFormat</code> 用于保护和取消保护的标识以及存储在 cookie 值中的其他属性。如果未提供， <code>TicketDataFormat</code> 使用创建 <code>DataProtectionProvider</code> 。
验证	检查的选项为有效的方法。

设置 `CookieAuthenticationOptions` 中的身份验证的服务配置中 `ConfigureServices` 方法：

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

Cookie 策略中间件

[Cookie 策略中间件](#) 启用应用程序中的 cookie 策略功能。将该中间件添加到应用程序处理管道是顺序敏感; 它只影

响在管道中后它注册的组件。

```
app.UseCookiePolicy(cookiePolicyOptions);
```

[CookiePolicyOptions](#) 到 Cookie 策略中间件提供可用于控制到 cookie 处理处理程序的 cookie 处理和挂钩全局特性，在追加或删除 cookie 的时间。

属性	描述
HttpOnly	会影响是否必须为 cookie HttpOnly，该值一个标志，指示是否 cookie 应只允许到服务器可访问。默认值为 <code>HttpOnlyPolicy.None</code> 。
MinimumSameSitePolicy	影响 cookie 的同一点属性（见下文）。默认值为 <code>SameSiteMode.Lax</code> 。此选项仅供 ASP.NET Core 2.0 +。
OnAppendCookie	当追加 cookie 时调用。
OnDeleteCookie	当删除 cookie 时调用。
安全	会影响是否必须安全 cookie。默认值为 <code>CookieSecurePolicy.None</code> 。

MinimumSameSitePolicy (ASP.NET 2.0 + 仅内核)

默认值 `MinimumSameSitePolicy` 值是 `SameSiteMode.Lax` 允许 OAuth2 身份验证。若要严格强制执行的同一点策略 `SameSiteMode.Strict`，将其设置 `MinimumSameSitePolicy`。虽然此设置将中断 OAuth2 和其他跨域身份验证方案，它将提升其他类型的应用程序不依赖于跨域请求处理的 cookie 安全的级别。

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

Cookie 策略中间件设置 `MinimumSameSitePolicy` 可能会影响你的设置的 `Cookie.SameSite` 中 `CookieAuthenticationOptions` 根据下面的矩阵的设置。

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	最终的 COOKIE.SAMESITE 设置
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

创建身份验证 cookie

若要创建一个保存用户信息的 cookie，您必须先构造 [ClaimsPrincipal](#)。序列化用户信息并将其存储在 cookie 中。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

创建`ClaimsIdentity`与任何所需声明s 并调用`SignInAsync`若要在用户登录:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("FullName", user.FullName),
    new Claim(ClaimTypes.Role, "Administrator"),
};

var claimsIdentity = new ClaimsIdentity(
    claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties
{
    //AllowRefresh = <bool>,
    // Refreshing the authentication session should be allowed.

    //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    // The time at which the authentication ticket expires. A
    // value set here overrides the ExpireTimeSpan option of
    // CookieAuthenticationOptions set with AddCookie.

    //IsPersistent = true,
    // Whether the authentication session is persisted across
    // multiple requests. Required when setting the
    // ExpireTimeSpan option of CookieAuthenticationOptions
    // set with AddCookie. Also required when setting
    // ExpiresUtc.

    //IssuedUtc = <DateTimeOffset>,
    // The time at which the authentication ticket was issued.

    //RedirectUri = <string>
    // The full path or absolute URI to be used as an http
    // redirect response value.
};

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);
```

`SignInAsync` 创建一个加密的 cookie, 并将其添加到当前响应。如果没有指定 `AuthenticationScheme`, 则使用默认方案。

事实上, 使用的加密是 ASP.NET Core[数据保护](#)系统。如果您正在承载应用程序在多台计算机、负载平衡应用程序, 或使用 web 场, 则必须配置[数据保护](#)使用同一密钥链和应用程序标识符。

注销

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要注销当前用户, 并删除其 cookie, 调用`SignOutAsync`:

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

如果你未使用 `CookieAuthenticationDefaults.AuthenticationScheme` (或"Cookie") 的方案 (例如, "ContosoCookie"), 提供你在配置身份验证提供程序时使用的方案。否则, 使用的默认方案。

对后端更改的作出反应

一旦创建 cookie, 它将成为标识的单个来源。即使在后端系统中禁用用户, cookie 身份验证系统不了解, 并且用户一直保持登录, 只要其 cookie 的有效期。

`ValidatePrincipal` 中 ASP.NET 核心事件 2.x 或 `ValidateAsync` 中 ASP.NET Core 1.x 可用来截获和重写的 cookie 身份验证方法。这种方法可以降低吊销用户访问应用程序的风险。

Cookie 验证的一种方法取决于跟踪的用户数据库已更改时。如果颁发用户的 cookie 以来, 数据库没有被更改, , 则无需重新进行用户身份验证其 cookie 是否仍有效。若要实现此方案中, 数据库中实现 `IUserRepository` 对于此示例中, 存储 `LastChanged` 值。在数据库中, 更新的任何用户时 `LastChanged` 值设置为当前时间。

为了使 cookie 无效时的数据库更改基于 `LastChanged` 值时, 请创建具有 cookie `LastChanged` 声明包含当前 `LastChanged` 从数据库的值:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要实现的替代 `ValidatePrincipal` 事件, 带有以下签名的类中派生自方法写入 `CookieAuthenticationEvents`:

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

示例如下所示:

```

using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        // Get the database from registered DI services.
        _userRepository = userRepository;
    }

    public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}

```

在中的 cookie 服务注册期间注册的事件实例 `ConfigureServices` 方法。提供有关指定了作用域的服务注册你 `CustomCookieAuthenticationEvents` 类：

```

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
{
    options.EventsType = typeof(CustomCookieAuthenticationEvents);
});

services.AddScoped<CustomCookieAuthenticationEvents>();

```

请考虑在其中更新用户的名称的情况下一决策不会影响任何方式中的安全性。如果你想要非破坏性地更新的用户主体，调用 `context.ReplacePrincipal` 并设置 `context.ShouldRenew` 属性 `true`。

警告

此处介绍的方法上的每个请求触发。这可能导致应用程序较大的性能损失。

永久 cookie

你可能想要在浏览器会话之间保持的 cookie。仅应使用显式用户有一个"记住我"复选框上登录名或类似机制同意的情况下启用此持久性。

下面的代码段将创建的标识和能够通过浏览器闭包后保留下来的相应 cookie。遵循任何以前配置的滑动过期设置。如果 cookie 过期浏览器处于关闭状态时，浏览器将它重新启动后清除 cookie。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
});
```

[AuthenticationProperties](#)类驻留在[Microsoft.AspNetCore.Authentication](#)命名空间。

绝对 cookie 到期时间

你可以设置与绝对过期时间[ExpiresUtc](#)。你还必须设置[IsPersistent](#)；否则为[ExpiresUtc](#)将被忽略，并创建一个单会话 cookie。当[ExpiresUtc](#)上设置[SignInAsync](#)，它将重写的值[ExpireTimeSpan](#)选项[CookieAuthenticationOptions](#)，如果设置。

下面的代码段将创建的标识和持续时间为 20 分钟的相应 cookie。这将忽略任何以前配置的滑动过期设置。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
});
```

请参阅

- [身份验证 2.0 更改 / 迁移公告](#)
- [使用方案限制标识](#)
- [基于声明的授权](#)
- [基于策略的角色检查](#)

Azure Active Directory 与 ASP.NET Core

2018/4/10 • 1 min to read • [Edit Online](#)

- [Integrating Azure AD Into an ASP.NET Core Web App](#)(将 Azure AD 集成到 ASP.NET Core Web 应用中)
- [Calling a ASP.NET Core Web API From a WPF Application Using Azure AD](#)(从使用 Azure AD 的 WPF 应用程序调用 ASP.NET Core Web API)
- [Calling a Web API in an ASP.NET Core Web Application Using Azure AD](#)(在使用 Azure AD 的 ASP.NET Core Web 应用程序中调用 Web API)
- [带有 Azure AD B2C 的 ASP.NET Core Web API](#)

Azure Active Directory B2C ASP.NET 核心中使用云身份验证

2018/3/15 • 6 min to read • [Edit Online](#)

作者:Cam Soper

Azure Active Directory B2C (Azure AD B2C) 是为 web 和移动应用的云标识管理解决方案。服务提供用于在云中和本地托管的应用的身份验证。身份验证类型包括个人帐户，社交网络帐户，以及联合企业帐户。此外，Azure AD B2C 可以提供以最小配置多因素身份验证。

提示

Azure Active Directory (Azure AD) Azure AD B2C 包括单独的产品。Azure AD 租户表示是一个组织，而 Azure AD B2C 租户表示标识要用于信赖方应用程序的集合。若要了解详细信息，请参阅[Azure AD B2C: 常见问题 \(FAQ\)](#)。

在本教程中，学习如何：

- 创建 Azure Active Directory B2C 租户
- 在 Azure AD B2C 注册某个应用程序
- 使用 Visual Studio 创建 ASP.NET 核心 web 应用配置为使用 Azure AD B2C 租户进行身份验证
- 配置控制 Azure AD B2C 租户的行为的策略

系统必备

对于本演练具备以下条件：

- Microsoft Azure 订阅
- Visual Studio 2017 (任意版本)

创建 Azure Active Directory B2C 租户

创建 Azure Active Directory B2C 租户[文档中所述](#)。出现提示时，将租户与 Azure 订阅相关联是可选的出于本教程。

在 Azure AD B2C 中注册应用程序

新创建的 Azure AD B2C 租户中注册你的应用使用[文档中的步骤](#)下注册 web 应用部分。在停止创建 web 应用程序客户端密钥部分。出于本教程中，无需客户端密钥。

使用以下值：

设置	“值”	说明
名称	<应用程序名称>	输入名称描述你的应用对使用者的应用。
包括 web 应用程序/web API	是	
允许隐式流	是	

设置	“值”	说明
回复 URL	<input type="text" value="https://localhost:44300"/>	答复 Url 是 Azure AD B2C 其中返回您的应用程序请求的任何令牌的终结点。Visual Studio 提供要使用的答复 URL。现在, 输入 <input type="text" value="https://localhost:44300"/> 完成窗体。
应用程序 ID URI	将保留为空	无需进行本教程。
包括本机客户端	否	

警告

如果设置非 localhost 答复 URL, 请注意[允许的答复 URL 列表中的约束](#)。

注册应用后, 显示的租户中的应用列表。选择刚刚注册的应用。选择**复制**右侧的图标**应用程序 ID**字段可将它复制到剪贴板。

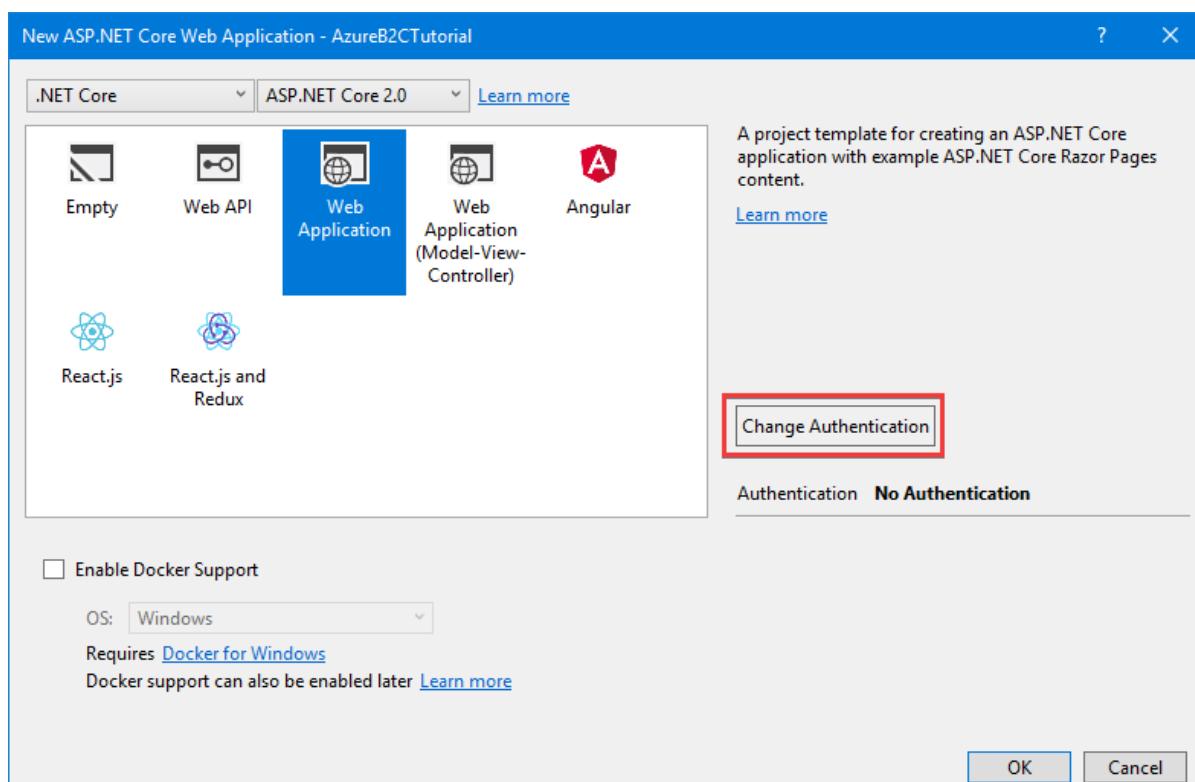
执行任何操作的详细信息在此时, 可以在 Azure AD B2C 租户中配置, 但使浏览器窗口保持打开状态。创建 ASP.NET Core 应用后, 没有更多的配置。

在 Visual Studio 2017 年 1 中创建 ASP.NET Core 应用

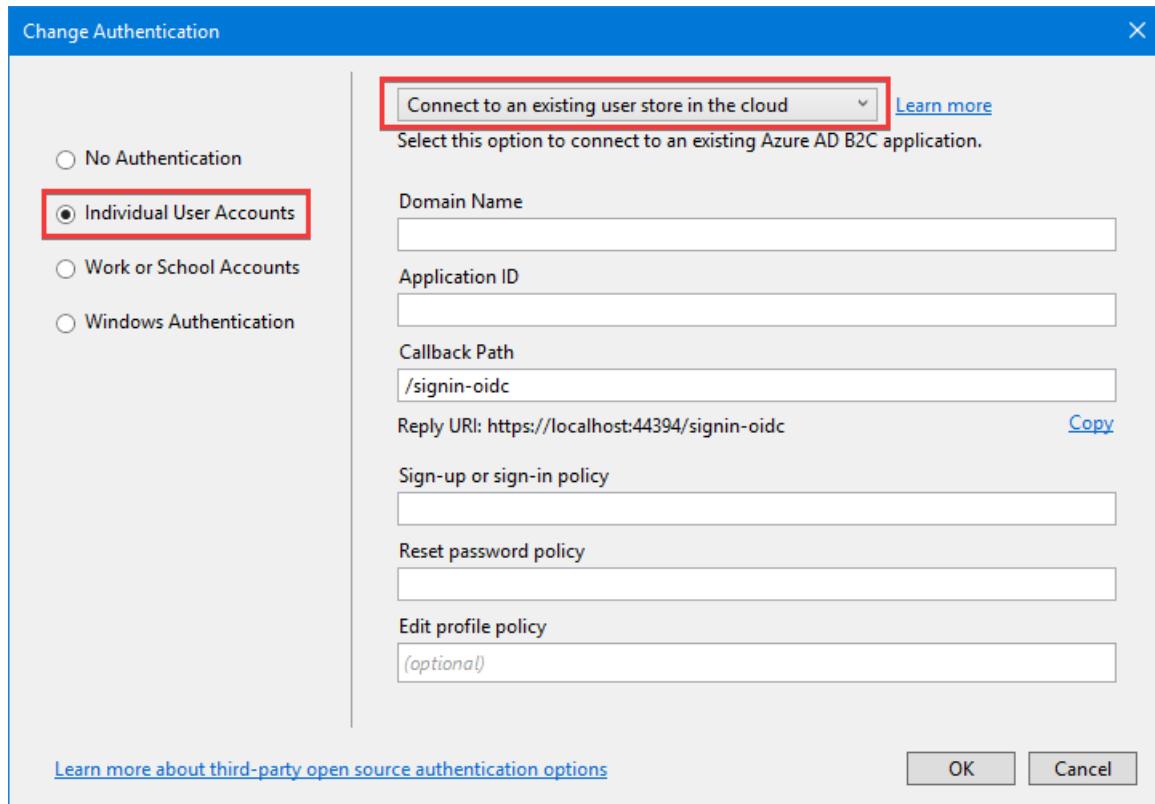
Visual Studio Web 应用程序模板可以配置为使用 Azure AD B2C 租户进行身份验证。

在 Visual Studio 中:

1. 创建新的 ASP.NET Core Web 应用程序。
2. 选择**Web 应用程序**从模板列表中。
3. 选择**更改身份验证**按钮。



4. 在更改身份验证对话框中，选择单个用户帐户，然后选择连接到现有用户存储在云中下拉列表中。



5. 完成窗体具有以下值：

设置	“值”
域名	<B2C 租户的域名>
应用程序 ID	<粘贴剪贴板中的应用程序 ID>
回调路径	<使用默认值>
注册或登录策略	B2C_1_SiUpIn
重置密码策略	B2C_1_SSPr
编辑配置文件策略	<将保留为空>

选择 **复制** 旁边链接 **答复 URI** 将答复 URI 复制到剪贴板。选择 **确定** 关闭更改身份验证对话框。选择 **确定** 创建 web 应用。

完成 B2C 应用程序注册

返回到 B2C 应用程序属性仍处于打开状态的浏览器窗口。更改临时 **答复 URL** 从 Visual Studio 指定更早版本的值复制。选择 **保存在窗口的顶部**。

提示

如果未复制的答复 URL，在 web 项目属性中，使用调试选项卡的 SSL 地址并追加 **CallbackPath** 值从 `appsettings.json`。

配置策略

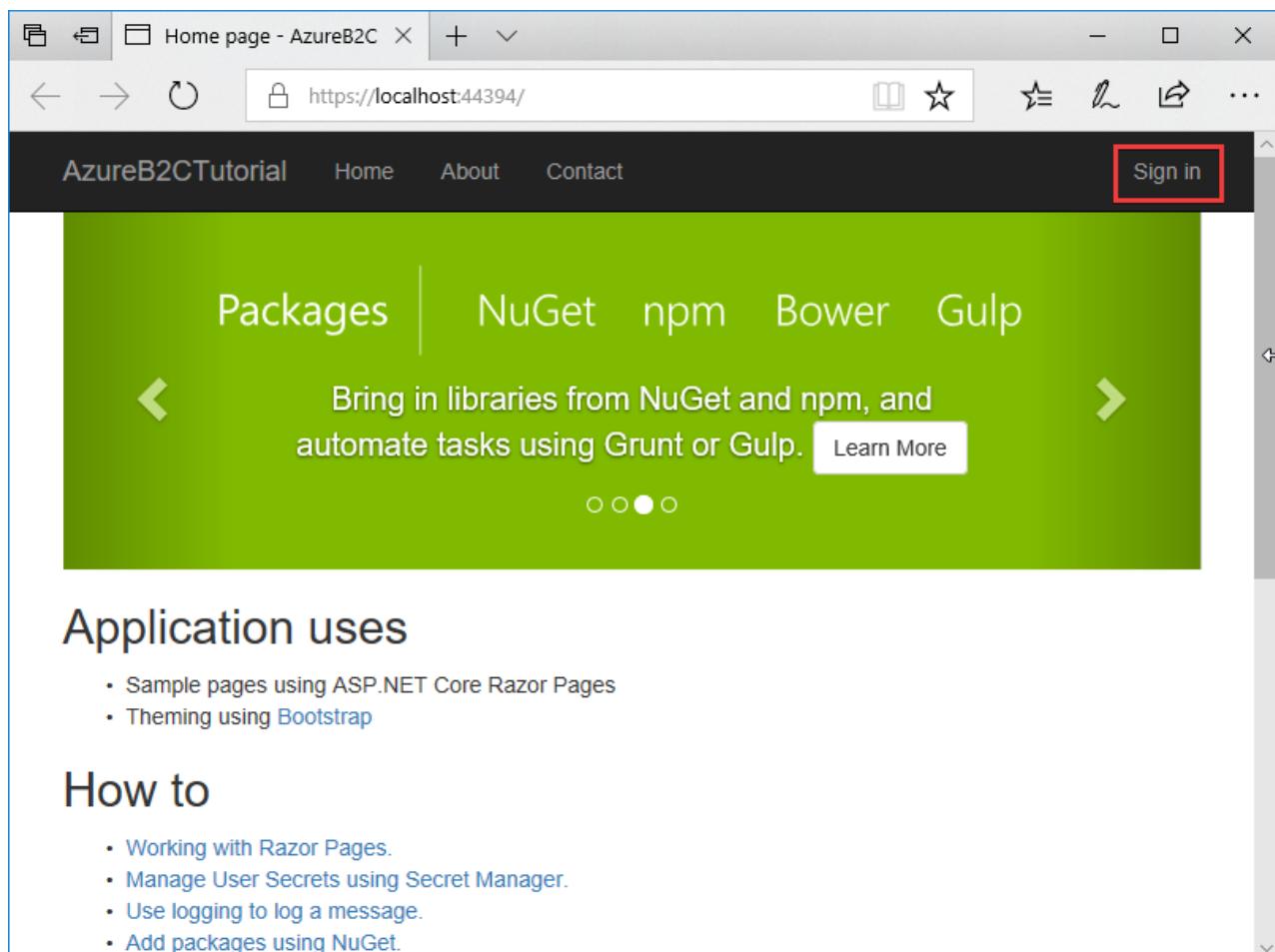
使用到的 Azure AD B2C 文档中的步骤[创建注册或登录策略](#)，然后创建密码重置策略。使用的文档中提供的示例值标识提供程序，注册属性，和应用程序声明。使用立即运行是可选的按钮以测试这些策略，如文档中所述。

警告

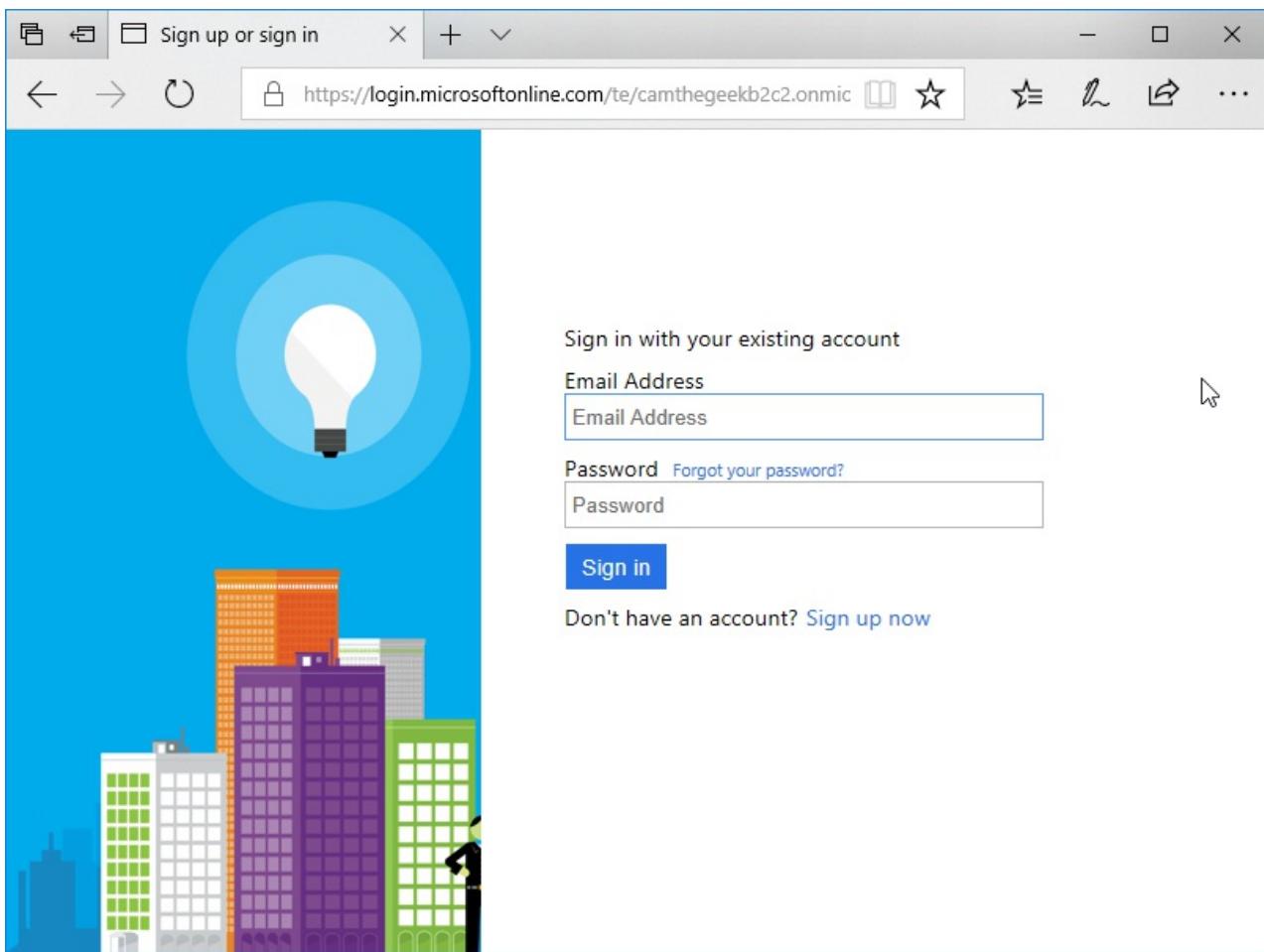
确保策略名称严格按说明在文档中，并与这些策略中使用[更改身份验证](#)Visual Studio 中的对话框。策略名称可以验证在 `appsettings.json`。

运行应用

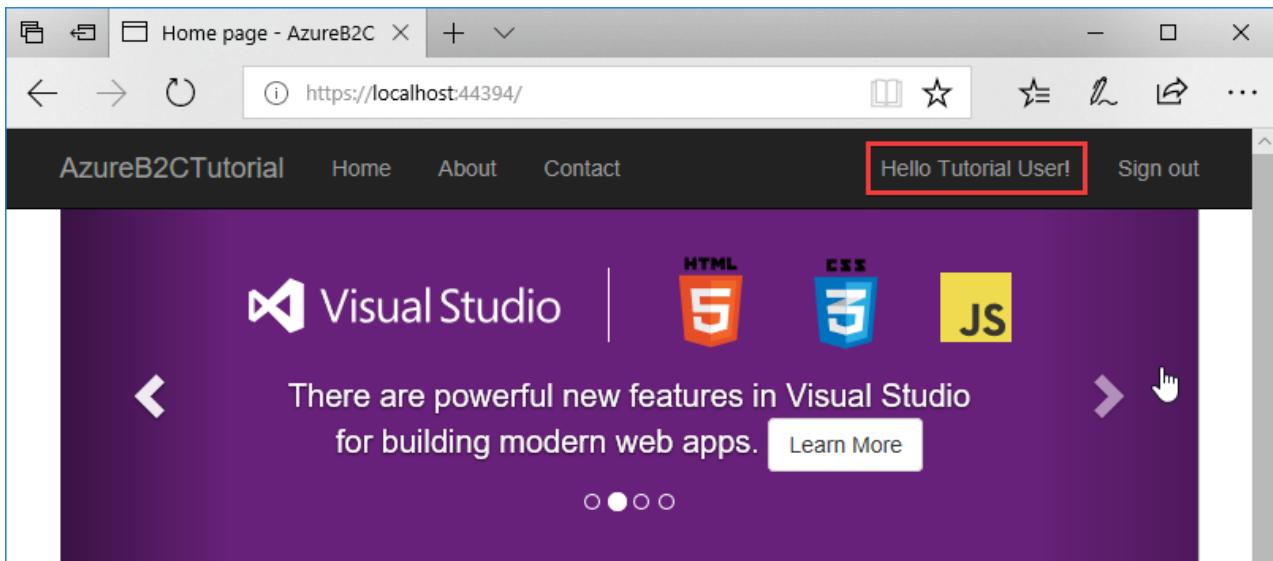
在 Visual Studio 中，按 F5 生成并运行应用程序。Web 应用启动后，选择登录。



浏览器将重定向到 Azure AD B2C 租户。使用现有帐户登录（如果已创建一个测试策略）或选择[立即注册](#)创建新帐户。[**忘记了密码？**](#)链接用于重置忘记了的密码。



已成功登录后，浏览器将重定向到 web 应用。



Application uses

- Sample pages using ASP.NET Core Razor Pages
- Theming using Bootstrap

How to

- Working with Razor Pages.
- Manage User Secrets using Secret Manager.
- Use logging to log a message.
- Add packages using NuGet.

后续步骤

在本教程中，你将了解：

- 创建 Azure Active Directory B2C 租户
- 在 Azure AD B2C 注册某个应用程序
- 使用 Visual Studio 创建 ASP.NET 核心 Web 应用程序配置为使用 Azure AD B2C 租户进行身份验证
- 配置控制 Azure AD B2C 租户的行为的策略

现在，ASP.NET Core 应用程序配置为使用 Azure AD B2C 进行身份验证，[Authorize 属性](#)可用来保护你的应用。在继续开发你的应用到学习：

- [自定义 Azure AD B2C 用户界面。](#)
- [配置密码复杂性要求。](#)
- [启用多因素身份验证。](#)
- 配置其他标识提供程序，例如[Microsoft](#), [Facebook](#), [Google](#), [Amazon](#), [Twitter](#), 和其他人。
- [使用 Azure AD Graph API](#)从 Azure AD B2C 租户中检索其他用户信息，例如组成员身份。
- [保护 web API 使用 Azure AD B2C 的 ASP.NET Core。](#)
- [从使用 Azure AD B2C 的.NET web 应用程序调用.NET web API。](#)

中的 web API 与 Azure Active Directory B2C 中 ASP.NET 核心云身份验证

2018/4/10 • 10 min to read • [Edit Online](#)

作者 : [Cam Soper](#)

Azure Active Directory B2C (Azure AD B2C) 是为 web 和移动应用的云标识管理解决方案。服务提供用于在云中和本地托管的应用的身份验证。身份验证类型包括个人帐户，社交网络帐户，以及联合企业帐户。此外，Azure AD B2C 可以提供以最小配置多因素身份验证。

提示

Azure Active Directory (Azure AD) 和 Azure AD B2C 包括单独的产品。Azure AD 租户表示是一个组织，而 Azure AD B2C 租户表示标识要用于信赖方应用程序的集合。若要了解详细信息，请参阅[Azure AD B2C: 常见问题 \(FAQ\)](#)。

由于 web API 具有没有用户界面，因此它们无法将用户重定向到 Azure AD B2C 等安全令牌服务。相反，API 是具有已用户进行身份验证与 Azure AD B2C 将调用应用程序中传递的持有者令牌。API 然后验证该令牌而无需直接用户交互。

在本教程中，学习如何：

- 创建 Azure Active Directory B2C 租户。
- 在 Azure AD B2C 中注册 Web API。
- 使用 Visual Studio 创建 Web API 配置为使用 Azure AD B2C 租户进行身份验证。
- 配置策略控制的 Azure AD B2C 租户的行为。
- 使用 Postman 以模拟 web 应用将其提供一个登录对话框中，检索令牌，并使用它来对 web API 发出请求。

系统必备

对于本演练具备以下条件：

- [Microsoft Azure 订阅](#)
- [Visual Studio 2017 \(任意版本\)](#)
- [Postman](#)

创建 Azure Active Directory B2C 租户

创建 Azure AD B2C 租户[文档中所述](#)。出现提示时，将租户与 Azure 订阅相关联是可选的出于本教程。

配置注册或登录策略

使用到的 Azure AD B2C 文档中的步骤[创建注册或登录策略](#)。命名策略时**SignIn**。使用的文档中提供的示例值标识提供程序，注册属性，和应用程序声明。使用立即运行是可选的按钮以测试策略，如文档中所述。

在 Azure AD B2C 注册 API

新创建的 Azure AD B2C 租户中注册你的 API 使用[文档中的步骤下注册 web API](#)部分。

使用以下值：

设置	值	说明
名称	<API 名称>	输入名称描述你的应用对使用者的应用。
包括 web 应用程序/web API	是	
允许隐式流	是	
回复 URL	https://localhost	答复 Url 是 Azure AD B2C 其中返回您的应用程序请求的任何令牌的终结点。
应用程序 ID URI	api	URI 不需要解析为物理地址。它只需是唯一的。
包括本机客户端	否	

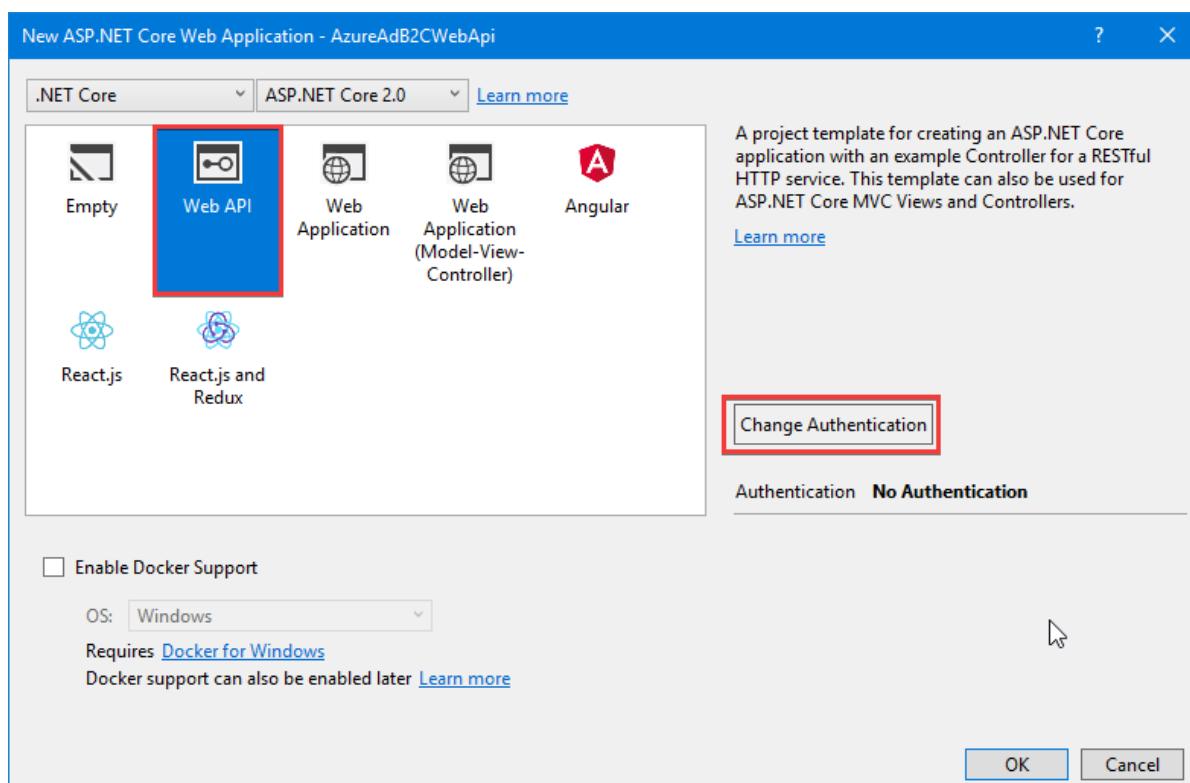
注册 API 后，将显示在租户中的应用和 API 的列表。选择刚刚注册的 API。选择 **复制** 右侧的图标 **应用程序 ID** 字段可将它复制到剪贴板。选择 **发布作用域** 和验证默认 *user_impersonation* 作用域是存在。

在 Visual Studio 2017 年 1 中创建 ASP.NET Core 应用

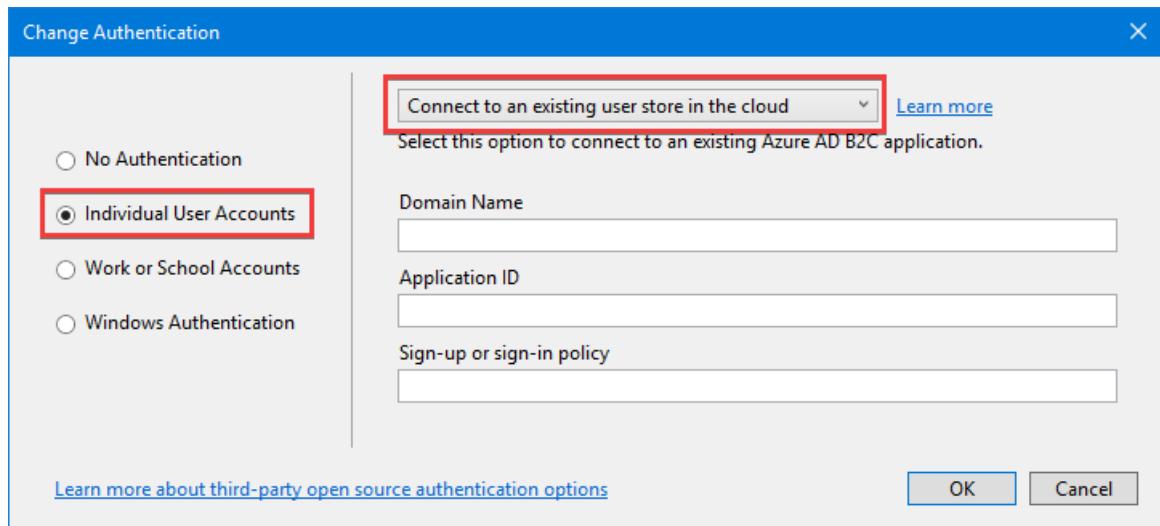
Visual Studio Web 应用程序模板可以配置为使用 Azure AD B2C 租户进行身份验证。

在 Visual Studio 中：

1. 创建新的 ASP.NET Core Web 应用程序。
2. 选择 **Web API** 从模板列表中。
3. 选择 **更改身份验证** 按钮。



4. 在更改身份验证对话框中，选择单个用户帐户，然后选择连接到现有用户存储在云中下拉列表中。



5. 完成窗体具有以下值：

设置	值
域名	<B2C 租户的域名>
应用程序 ID	<粘贴剪贴板中的应用程序 ID>
注册或登录策略	B2C_1_SignUpIn

选择确定关闭更改身份验证对话框。选择确定创建 web 应用。

Visual Studio 使用名为的控制器中创建 web API `ValuesController.cs` 返回 GET 请求的硬编码值。类用修饰 `Authorize` 属性，因此所有请求都需要身份验证。

运行 web API

在 Visual Studio 中，运行 API。Visual Studio 将启动浏览器指向的 API 根 URL。请注意地址栏中中的 URL 并将在后台运行的 API。

注意

由于没有为根 URL 定义无控制器，浏览器将显示 404（找不到页）错误。这是预期行为。

使用 Postman 来获取令牌和测试 API

Postman 是用于测试的工具 web API。对于本教程，Postman 模拟的 web 应用程序访问 web API 代表该用户。

为 web 应用注册 Postman

由于 Postman 模拟的 web 应用程序可以从 Azure AD B2C 租户获取令牌，则必须注册在租户中为 web 应用。注册使用 Postman 文档中的步骤下注册 web 应用部分。在停止创建 web 应用程序客户端密钥部分。出于本教程中，无需客户端密钥。

使用以下值：

设置	值	说明
名称	Postman	

设置	值	说明
包括 web 应用程序/web API	是	
允许隐式流	是	
回复 URL	<code>https://getpostman.com/postman</code>	
应用程序 ID URI	<将保留为空>	无需进行本教程。
包括本机客户端	否	

新注册的 web 应用程序需要访问 web API 代表该用户的权限。

1. 选择**Postman**的应用，然后选择列表中**API** 访问从左侧菜单。
2. 选择**+ 添加**。
3. 在选择 **API**下拉列表中，选择 web API 的名称。
4. 在选择作用域下拉列表中，确保选中所有作用域。
5. 选择**确定**。

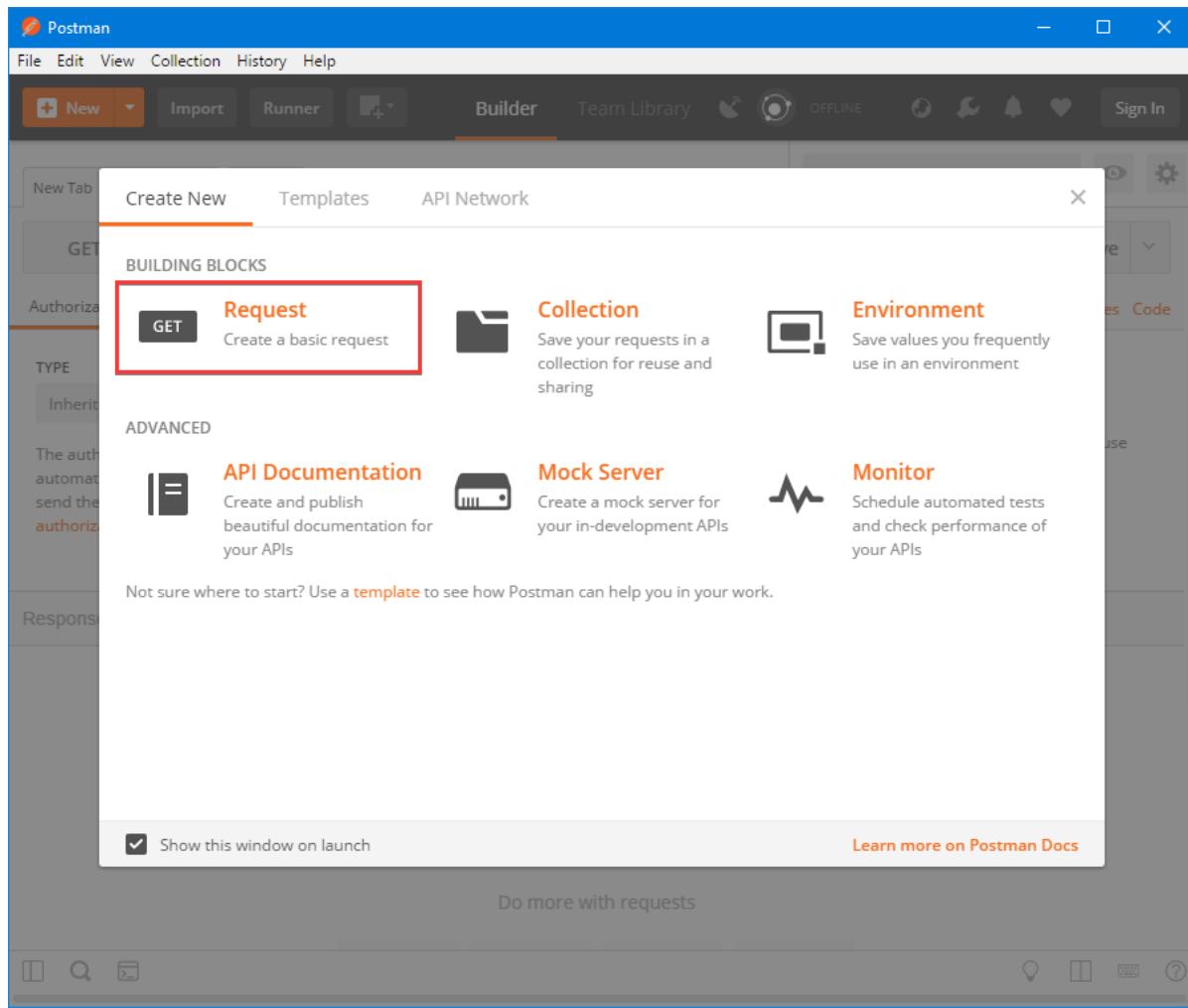
请注意 Postman 应用的应用程序 ID，因为它需获取的持有者令牌。

创建 Postman 请求

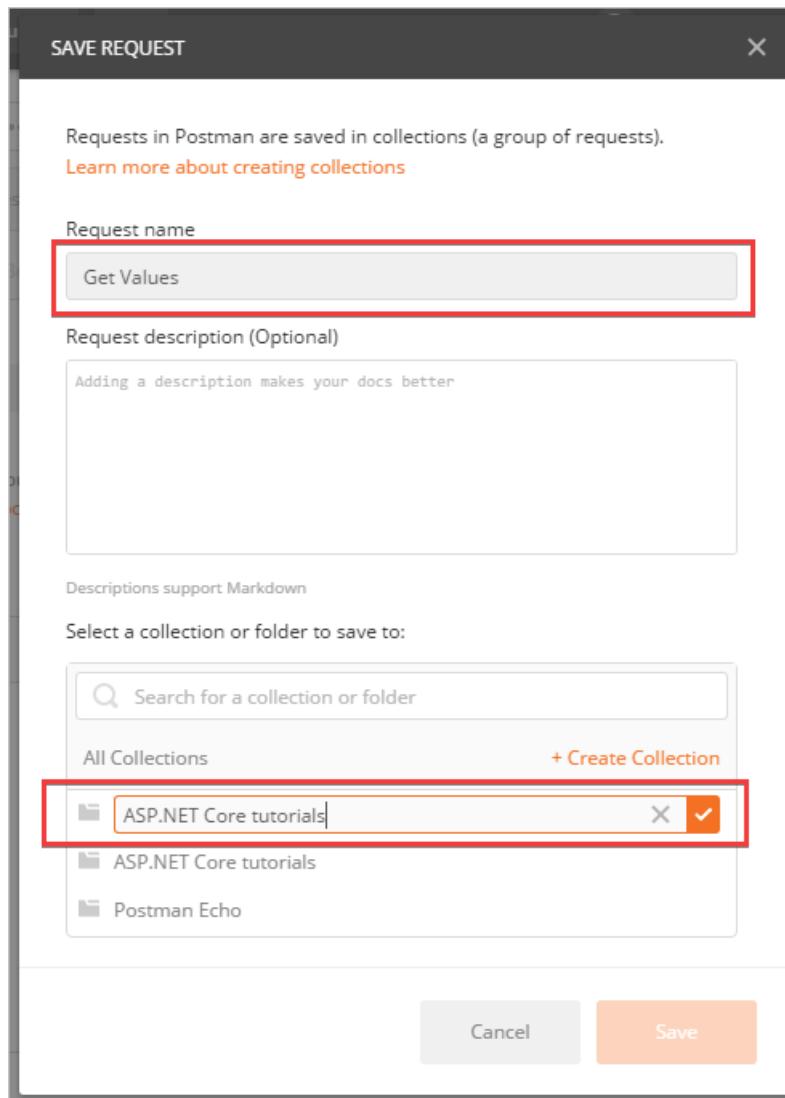
启动 Postman。默认情况下，显示 Postman新建时启动的对话框。如果未显示对话框中，选择**+ 新建**在左上角的按钮。

从新建对话框：

1. 选择**请求**。



2. 输入获取值中请求名称框。
3. 选择**+ 创建集合**以创建用于存储请求新的集合。命名集合ASP.NET Core 教程, 然后选择复选标记。

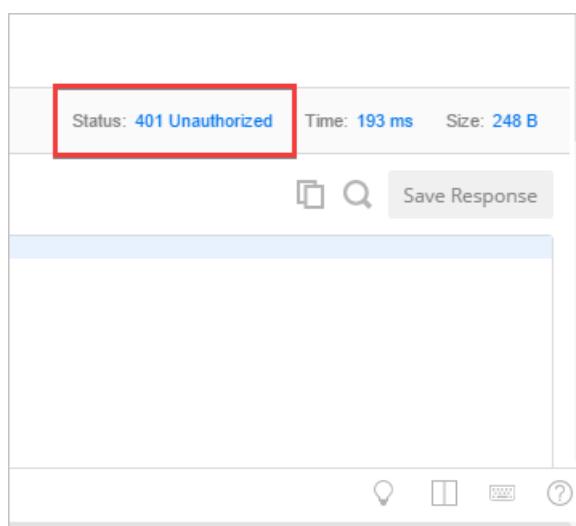


4. 选择将保存到 ASP.NET 核心教程按钮。

测试无需身份验证 web API

若要验证 web API 需要身份验证，首先请无身份验证的请求。

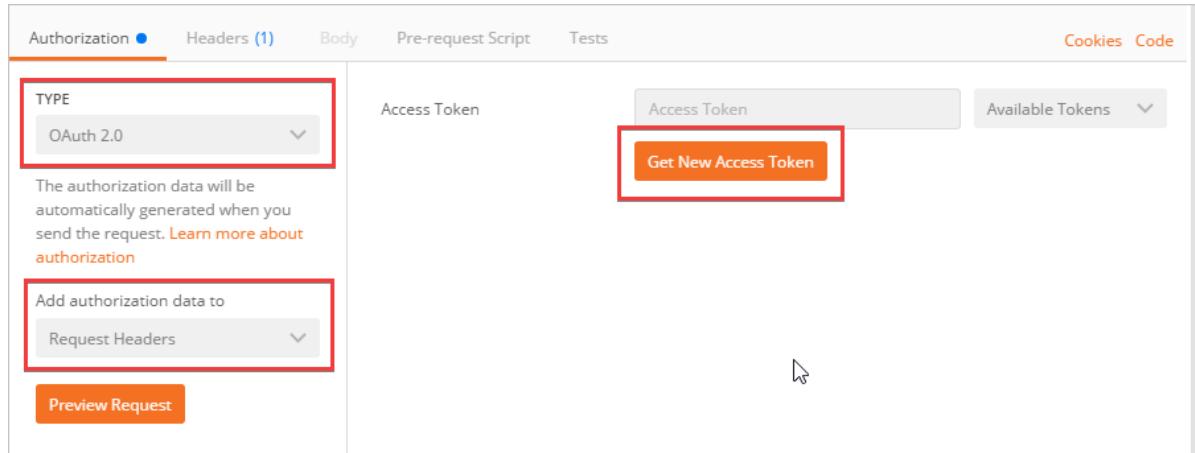
1. 在输入请求 URL 框中，输入的 URL `valuesController`。URL 是显示在浏览器中使用的相同 `api/` 值追加。示例将 `https://localhost:44375/api/values`。
2. 选择发送按钮。
3. 请注意响应的状态是 `401 未授权`。



获取的持有者令牌

要对 web API 进行身份验证的请求，持有者令牌是必需的。Postman 使得容易登录到 Azure AD B2C 租户并获取的令牌。

1. 上授权选项卡上，在类型下拉列表中，选择**OAuth 2.0**。在授权将数据添加到下拉列表中，选择请求标头。选择获取新访问令牌。



2. 完成获取新访问令牌，如下所示的对话框：

设置	值	说明
标记名称	<标记名称>	输入该令牌的描述性名称。
授予类型构建的	隐式	
回调 URL	https://getpostman.com/postman	
身份验证 URL	<a href="https://login.microsoftonline.com/<租户域名>/oauth2/v2.0/authorize?client_id=B2C_1_SiUpIn">https://login.microsoftonline.com/<租户域名>/oauth2/v2.0/authorize?client_id=B2C_1_SiUpIn	
客户端 ID	<输入 Postman 应用应用程序 ID>	
客户端密钥	<将保留为空>	
范围	<a href="https://<tenant domain name>/<api>/user_impersonation openid offline_access">https://<tenant domain name>/<api>/user_impersonation openid offline_access	替换<租户域名>与租户的域名。替换<api>替换为 Web API 项目名称。你还可以使用应用程序 id。用于 URL 的模式是： <a href="https://<tenant>.onmicrosoft.com/<app_name_or_id>/<scope 名称>">https://<tenant>.onmicrosoft.com/<app_name_or_id>/<scope 名称> 。
客户端身份验证	在正文中发送客户端凭据	

3. 选择令牌请求按钮。
4. Postman 打开包含 Azure AD B2C 租户的登录对话框中的新窗口中。使用现有帐户登录（如果已创建一个测试策略）或选择立即注册创建新帐户。**忘记了密码？**链接用于重置忘记了的密码。
5. 已成功登录后，该窗口将关闭与管理访问令牌此时将显示对话框。向下的滚动到底部并选择使用令牌按钮。

The screenshot shows the Postman interface with the following details:

- Method:** GET
- URL:** https://localhost:44375/api/values
- Collection:** MANAGE ACCESS TOKENS
- Item:** Azure AD B2C Tutorial Te...
- token_type:** Bearer
- expires_in:** 3600
- Use Token** button (highlighted with a red box)

测试身份验证与 web API

选择发送按钮以重新发送请求。此时，响应状态是200 确定JSON 负载且在响应上可见正文选项卡。

The screenshot shows the Postman interface with the following details:

- Status:** 200 OK
- Time:** 111 ms
- Size:** 287 B
- Body:**
 - Pretty
 - Raw
 - Preview
 - JSON (highlighted with a red box)
- Response Body:** ["value1", "value2"]

后续步骤

在本教程中，你将了解：

- 创建 Azure Active Directory B2C 租户。
- 在 Azure AD B2C 中注册 Web API。
- 使用 Visual Studio 创建 Web API 配置为使用 Azure AD B2C 租户进行身份验证。
- 配置策略控制的 Azure AD B2C 租户的行为。
- 使用 Postman 以模拟 web 应用将其提供一个登录对话框中，检索令牌，并使用它来对 web API 发出请求。

在继续学习到开发你的 API：

- [保护 web 应用使用 Azure AD B2C 的 ASP.NET Core。](#)
- [从使用 Azure AD B2C 的.NET web 应用程序调用.NET web API。](#)
- [自定义 Azure AD B2C 用户界面。](#)
- [配置密码复杂性要求。](#)
- [启用多因素身份验证。](#)
- [配置其他标识提供程序，例如Microsoft, Facebook, Google, Amazon, Twitter，和其他人。](#)
- [使用 Azure AD Graph API 从 Azure AD B2C 租户中检索其他用户信息，例如组成员身份。](#)

基于 ASP.NET 核心项目使用单个用户帐户创建项目

2018/4/10 • 1 min to read • [Edit Online](#)

ASP.NET 核心标识包含在 Visual Studio 中使用“单个用户帐户”选项的项目模板中。

中使用的.NET 核心 CLI 中可用的身份验证模板 `-au Individual`：

```
dotnet new mvc -au Individual  
dotnet new webapi -au Individual  
dotnet new razor -au Individual
```

以下文章演示了如何使用 ASP.NET Core 使用单个用户帐户的模板中生成的代码：

- [使用 SMS 设置双因素身份验证](#)
- [ASP.NET Core 中的帐户确认和密码恢复](#)
- [使用受授权的用户数据创建 ASP.NET Core 应用](#)

ASP.NET Core 中的授权

2018/5/14 • 1 min to read • [Edit Online](#)

- [介绍](#)
- [通过授权保护的用户数据创建应用](#)
- [Razor 页面授权](#)
- [简单授权](#)
- [基于角色的授权](#)
- [基于声明的授权](#)
- [基于策略的授权](#)
- [自定义授权策略提供程序](#)
- [要求处理程序中的依赖关系注入](#)
- [基于资源的授权](#)
- [基于视图的授权](#)
- [特定方案授权](#)

在 ASP.NET Core 中授权简介

2018/4/27 • 1 min to read • [Edit Online](#)

授权是指确定进程用户是能够执行。例如，管理用户可以创建文档库、将文档添加、编辑文档，以及删除它们。使用库的非管理用户仅有权读取文档。

授权是正交和独立于身份验证。但是，授权要求的身份验证机制。身份验证是有助于确定用户是谁的过程。身份验证可能会创建一个或多个标识为当前用户。

授权类型

ASP.NET 核心授权提供一个简单、声明性[角色](#)以及丰富[基于策略](#)的模型。要求，以表示授权，并且处理程序评估针对需求的用户的声明。命令性检查可以基于简单的策略或策略的评估用户标识和该用户尝试访问资源的属性。

命名空间

授权组件，包括 `AuthorizeAttribute` 和 `AllowAnonymousAttribute` 中找不到属性，`Microsoft.AspNetCore.Authorization` 命名空间。

请查阅上的文档[简单授权](#)。

使用受授权的用户数据创建 ASP.NET Core 应用

2018/5/4 • 21 min to read • [Edit Online](#)

作者: [Rick Anderson](#) 和 [Joe Audette](#)

本教程演示如何使用受授权的用户数据创建 ASP.NET 核心 web 应用。它显示的身份验证的（注册）的用户的联系人列表已创建。有三个安全组：

- **注册用户**可以查看所有已批准的数据并可以编辑/删除其自己的数据。
- **管理器**可以批准或拒绝联系人数据。仅批准的联系人是对用户可见。
- **管理员**可以批准/拒绝和编辑/删除任何数据。

在下图中，用户 Rick (`rick@example.com`) 登录。Rick 只能查看被批准的联系人和编辑/删除/新建他联系人的链接。仅最新的记录，创建由 Rick，显示编辑和删除链接。其他用户不会看到的最新记录，直到经理或管理员的状态更改为“已批准”。

The screenshot shows a browser window for the URL `https://localhost:44380/Contacts`. The title bar says "Index - ContactManager". The top navigation bar includes "ContactManager", "Home", "About", "Contact", "Hello rick@example.com!", and "Log off". A yellow box highlights the greeting message. The main content area is titled "Create New" and displays a table of contact data:

Address	City	Email	Name	State	Zip	Status	
5678 1st Ave W	Redmond	thorsten@example.com	Thorsten Weinrich	WA	10999	Approved	Details
9012 State St	Redmond	yuhong@example.com	Yuhong Li	WA	10999	Approved	Details
3456 Maple St	Redmond	jon@example.com	Jon Orton	WA	10999	Approved	Details
123 N 456 E	GF	rick@example.com	Rick Anderson	MT	59405	Submitted	Edit Details Delete

A red box highlights the "Edit", "Details", and "Delete" links for the last contact entry. At the bottom left, it says "© 2017 - ContactManager".

在下图中，`manager@contoso.com` 进行签名和管理员角色中：

Address	City	Email	Name	State	Zip	Status	
1234 Main St	Redmond	debra@example.com	Debra Garcia	WA	10999	Rejected	Details
5678 1st Ave W	Redmond	thorsten@example.com	Thorsten Weinrich	WA	10999	Approved	Details
9012 State st	Redmond	yuhong@example.com	Yuhong Li	WA	10999	Approved	Details
3456 Maple St	Redmond	jon@example.com	Jon Orton	WA	10999	Approved	Details
7890 2nd Ave E	Redmond	diliana@example.com	Diliana Alexieva-Bosseva	WA	10999	Submitted	Details
123 N 456 E	GF	rick@example.com	Rick Anderson	MT	59405	Approved	Details

© 2016 - ContactManager

下图显示了管理器的联系人的详细信息视图：

Name	Rick Anderson
Email	rick@example.com
Address	123 N 456 E
City	GF
State	MT
Zip	59405
Status	Submitted

[Approve](#) [Reject](#)

[Edit](#) | [Back to List](#)

© 2016 - ContactManager

批准和拒绝按钮仅显示经理和管理员。

在下图中，`admin@contoso.com` 进行签名和管理员角色中：

Address	City	Email	Name	State	Zip	Status	
1234 Main St	Redmond	debra@example.com	Debra Garcia	WA	10999	Rejected	Edit Details Delete
5678 1st Ave W	Redmond	thorsten@example.com	Thorsten Weinrich	WA	10999	Approved	Edit Details Delete
9012 State st	Redmond	yuhong@example.com	Yuhong Li	WA	10999	Approved	Edit Details Delete
3456 Maple St	Redmond	jon@example.com	Jon Orton	WA	10999	Approved	Edit Details Delete

管理员具有所有权限。她可以读取、编辑或删除任何联系人，并更改联系人的状态。

应用程序由基架以下 Contact 模型：

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

该示例包含以下授权处理器：

- `ContactIsOwnerAuthorizationHandler`：可确保用户只能编辑其数据。
- `ContactManagerAuthorizationHandler`：允许管理器批准或拒绝联系人。
- `ContactAdministratorsAuthorizationHandler`：允许管理员批准或拒绝联系人，还可以编辑/删除联系人。

系统必备

本教程被高级。你应熟悉：

- `ASP.NET Core`
- 身份验证
- 帐户确认和密码恢复

- 授权
- Entity Framework Core

请参阅此 [PDF 文件](#) ASP.NET 核心 MVC 版本。本教程的 ASP.NET 核心 1.1 版本是[这](#)文件夹。ASP.NET 核心示例是在 1.1 [示例](#)。

初学者和已完成应用程序

[下载完成应用](#)。[测试](#)已完成的应用程序使你熟悉其安全功能。

初学者应用

[下载初学者应用](#)。

运行应用，点击**ContactManager**链接，并验证是否可以创建、编辑和删除联系人。

保护用户数据

下列各节具有所有主要的步骤以创建安全的用户数据应用。你可能会发现已完成的项目是指很有帮助。

将向用户的联系人数据

使用 ASP.NET 标识用户 ID，以确保用户可以编辑其数据，而非其他用户数据。添加 `OwnerID` 和 `ContactStatus` 到 `Contact` 模型：

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string OwnerID { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` 是从用户的 ID `AspNetUser` 表中标识数据库。`Status` 字段确定是否可由常规用户查看联系人。

创建一个新迁移并更新数据库：

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

需要 HTTPS 和经过身份验证的用户

添加 `IHostingEnvironment` 到 `Startup`：

```

public class Startup
{
    public Startup(IConfiguration configuration, IHostingEnvironment env)
    {
        Configuration = configuration;
        Environment = env;
    }

    public IConfiguration Configuration { get; }
    private IHostingEnvironment Environment { get; }

```

在 `ConfigureServices` 方法 `Startup.cs` 文件中，添加 `RequireHttpsAttribute` 授权筛选器：

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity< ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores< ApplicationContext >()
        .AddDefaultTokenProviders();

    var skipHTTPS = Configuration.GetValue< bool >("LocalTest:skipHTTPS");
    // requires using Microsoft.AspNetCore.Mvc;
    services.Configure< MvcOptions >(options =>
    {
        // Set LocalTest:skipHTTPS to true to skip SSL requirement in
        // debug mode. This is useful when not using Visual Studio.
        if (Environment.IsDevelopment() && !skipHTTPS)
        {
            options.Filters.Add(new RequireHttpsAttribute());
        }
    });
}

```

如果你使用 Visual Studio，则启用 HTTPS。

若要将 HTTP 请求重定向到 HTTPS，请参阅[URL 重写中间件](#)。如果你使用 Visual Studio Code 或不包括为支持 HTTPS 的测试证书的本地平台上测试：

设置 `"LocalTest:skipSSL": true` 中 `appsettings.Development.json` 文件。

要求经过身份验证的用户

设置默认身份验证策略以要求用户进行身份验证。你可以选择不在与 Razor 页、控制器或操作方法级别的身份验证 `[AllowAnonymous]` 属性。设置默认身份验证策略以要求用户进行身份验证保护新添加的 Razor 页和控制器。默认情况下所需的身份验证为比依赖于新控制器和 Razor 页，以包含更安全 `[Authorize]` 属性。

与所有用户进行身份验证，要求 `AuthorizeFolder` 和 `AuthorizePage` 调用不是必需的。

更新 `ConfigureServices` 有以下更改：

- 注释掉 `AuthorizeFolder` 和 `AuthorizePage`。
- 设置默认身份验证策略以要求用户进行身份验证。

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    var skipHTTPS = Configuration.GetValue<bool>("LocalTest:skipHTTPS");
    // requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
        // Set LocalTest:skipHTTPS to true to skip SSL requirement in
        // debug mode. This is useful when not using Visual Studio.
        if (Environment.IsDevelopment() && !skipHTTPS)
        {
            options.Filters.Add(new RequireHttpsAttribute());
        }
    });
}

services.AddMvc();
//.AddRazorPagesOptions(options =>
//{
//    options.Conventions.AuthorizeFolder("/Account/Manage");
//    options.Conventions.AuthorizePage("/Account/Logout");
//});

services.AddSingleton<IEmailSender, EmailSender>();

// requires: using Microsoft.AspNetCore.Authorization;
//           using Microsoft.AspNetCore.Mvc.Authorization;
services.AddMvc(config =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    config.Filters.Add(new AuthorizeFilter(policy));
});

```

添加[AllowAnonymous](#)到索引中，因此匿名用户可以获取有关站点的信息，然后它们注册的有关，和联系人页面。

```

// requires using Microsoft.AspNetCore.Mvc.RazorPages;
[AllowAnonymous]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
}

```

添加 [\[AllowAnonymous\]](#) 到 [LoginModel](#) 和 [RegisterModel](#)。

配置测试帐户

[SeedData](#) 类将创建两个帐户：管理员和管理员。使用[机密管理器工具](#)设置这些帐户的密码。从项目目录设置密码（目录包含 *Program.cs*）：

```
dotnet user-secrets set SeedUserPW <PW>
```

更新 [Main](#) 使用测试密码：

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = BuildWebHost(args);

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var context = services.GetRequiredService<ApplicationContext>();
            context.Database.Migrate();

            // requires using Microsoft.Extensions.Configuration;
            var config = host.Services.GetRequiredService< IConfiguration>();
            // Set password with the Secret Manager tool.
            // dotnet user-secrets set SeedUserPW <pw>

            var testUserPw = config["SeedUserPW"];

            try
            {
                SeedData.Initialize(services, testUserPw).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService< ILogger<Program>>();
                logger.LogError(ex, "An error occurred while seeding the database.");
                throw ex;
            }
        }

        host.Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

创建测试帐户和更新联系人

更新 `Initialize` 中的方法 `SeedData` 类，以创建测试帐户：

```

public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
{
    using (var context = new ApplicationDbContext(
        serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
    {
        // For sample purposes we are seeding 2 users both with the same password.
        // The password is set with the following command:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw, "admin@contoso.com");
        await EnsureRole(serviceProvider, adminID, Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var uid = await EnsureUser(serviceProvider, testUserPw, "manager@contoso.com");
        await EnsureRole(serviceProvider, uid, Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}

private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
                                              string testUserPw, string UserName)
{
    var userManager = serviceProvider.GetService<UserManager< ApplicationUser >>();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new ApplicationUser { UserName = UserName };
        await userManager.CreateAsync(user, testUserPw);
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider,
                                                       string uid, string role)
{
    IdentityResult IR = null;
    var roleManager = serviceProvider.GetService<RoleManager< IdentityRole >>();

    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager< ApplicationUser >>();

    var user = await userManager.FindByIdAsync(uid);

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}

```

添加管理员的用户 ID 和 `ContactStatus` 向联系人。先创建一个"已提交"和一个"已拒绝"的联系人。将用户 ID 和状态添加到所有联系人。只能有一个联系人是所示：

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return; // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
    );
}
```

创建所有者、管理器中，和管理员授权处理程序

创建 `ContactIsOwnerAuthorizationHandler` 类授权文件夹。`ContactIsOwnerAuthorizationHandler` 验证对资源进行操作的用户拥有的资源。

```

using System.Threading.Tasks;
using ContactManager.Data;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        UserManager< ApplicationUser > _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager< ApplicationUser >
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                OperationAuthorizationRequirement requirement,
                Contact resource)
        {
            if (context.User == null || resource == null)
            {
                // Return Task.FromResult(0) if targeting a version of
                // .NET Framework older than 4.6:
                return Task.CompletedTask;
            }

            // If we're not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}

```

`ContactIsOwnerAuthorizationHandler` 调用[上下文。成功](#)当前经过身份验证的用户是否联系人的所有者。授权处理程序通常：

- 返回`context.Succeed`满足的要求。
- 返回`Task.CompletedTask`当不满足要求。`Task.CompletedTask`既不成功或失败—它允许运行其他授权处理程序。

如果你需要显式失败，返回[上下文。失败](#)。

该应用让联系人所有者到编辑/删除/创建其自己的数据。`ContactIsOwnerAuthorizationHandler`不需要请检查在要求参数中传递的操作。

创建管理器授权处理程序

创建 `ContactManagerAuthorizationHandler` 类授权文件夹。`ContactManagerAuthorizationHandler` 验证用户对资源进行操作是一个管理器。只有经理可以批准或拒绝内容更改 (新的或已更改的)。

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                OperationAuthorizationRequirement requirement,
                Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }
        }

        return Task.CompletedTask;
    }
}
```

创建一个管理员授权处理程序

创建 `ContactAdministratorsAuthorizationHandler` 类授权文件夹。`ContactAdministratorsAuthorizationHandler` 验证对资源进行操作的用户是管理员。管理员可以执行所有操作。

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
        : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            OperationAuthorizationRequirement requirement,
            Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

注册的授权处理程序

必须为注册服务使用实体框架核心[依赖关系注入](#)使用[AddScoped](#)。[ContactIsOwnerAuthorizationHandler](#) 使用[ASP.NET Core标识](#)，这基于实体框架核心。注册服务集合的处理程序，因此要对其可供[ContactsController](#) 通过[依赖关系注入](#)。将以下代码添加到末尾[ConfigureServices](#)：

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>()
        .AddDefaultTokenProviders();

    var skipHTTPS = Configuration.GetValue<bool>("LocalTest:skipHTTPS");
    // requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
        // Set LocalTest:skipHTTPS to true to skip SSL requirement in
        // debug mode. This is useful when not using Visual Studio.
        if (Environment.IsDevelopment() && !skipHTTPS)
        {
            options.Filters.Add(new RequireHttpsAttribute());
        }
    });
}

services.AddMvc();
//.AddRazorPagesOptions(options =>
//{
//    options.Conventions.AuthorizeFolder("/Account/Manage");
//    options.Conventions.AuthorizePage("/Account/Logout");
//});

services.AddSingleton<IEmailSender, EmailSender>();

// requires: using Microsoft.AspNetCore.Authorization;
//           using Microsoft.AspNetCore.Mvc.Authorization;
services.AddMvc(config =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    config.Filters.Add(new AuthorizeFilter(policy));
});

// Authorization handlers.
services.AddScoped<IAuthorizationHandler,
    ContactIsOwnerAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
    ContactAdministratorsAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
    ContactManagerAuthorizationHandler>();
}

```

`ContactAdministratorsAuthorizationHandler` 和 `ContactManagerAuthorizationHandler` 添加为单一实例。它们是单一实例，因为它们不使用 EF 和所需的所有信息都，请参阅 `Context` 参数 `HandleRequirementAsync` 方法。

支持授权

在本部分中，你将更新 Razor 的网页，并添加操作要求类。

查看联系人的操作要求类

查看 `ContactOperations` 类。此类包含要求应用程序支持：

```

using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
            new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
            new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
            new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
            new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
            new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
            new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
            "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}

```

创建用于 Razor 页面的基本类

创建一个包含在联系人 Razor 页中所使用的服务的基本类。基的类将该初始化代码放在一个位置：

```

using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<ApplicationUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<ApplicationUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}

```

前面的代码：

- 将添加 `IAuthorizationService` 服务访问的授权处理程序。
- 添加标识 `UserManager` 服务。
- 添加 `ApplicationContext`。

更新 CreateModel

更新创建页模型构造函数以使用 `DI_BasePageModel` 基类：

```
public class CreateModel : DI_BasePageModel
{
    public CreateModel(
        ApplicationContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }
}
```

更新 `CreateModel.OnPostAsync` 方法：

- 添加到的用户 ID `Contact` 模型。
- 调用授权处理程序，以验证用户有权创建联系人。

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    Contact.OwnerID = UserManager.GetUserId(User);

    // requires using ContactManager.Authorization;
    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, Contact,
        ContactOperations.Create);

    if (!isAuthorized.Succeeded)
    {
        return new ChallengeResult();
    }

    Context.Contact.Add(Contact);
    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

更新 IndexModel

更新 `OnGetAsync` 方法，以便仅被批准的联系人显示在常规用户：

```

public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                      select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserID = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status == ContactStatus.Approved
                                      || c.OwnerID == currentUserID);
        }

        Contact = await contacts.ToListAsync();
    }
}

```

更新 EditModel

添加授权处理程序以验证用户拥有联系人。正在验证资源授权，因为 `[Authorize]` 属性不足够。评估属性时，此应用程序没有对资源的访问。基于资源的授权必须是命令性。一旦应用程序有权访问该资源，通过在页模型中加载或加载内处理程序本身，则必须执行检查。你经常访问的资源，通过传入的资源键。

```

public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
            User, Contact,
            ContactOperations.Update);
    }
}

```

```

        ContactOperations.Update);
    }

    if (!isAuthorized.Succeeded)
    {
        return new ChallengeResult();
    }

    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Fetch Contact from DB to get OwnerID.
    var contact = await Context
        .Contact.AsNoTracking()
        .FirstOrDefaultAsync(m => m.ContactId == id);

    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await AuthorizationService.AuthorizeAsync(
        User, contact,
        ContactOperations.Update);

    if (!isAuthorized.Succeeded)
    {
        return new ChallengeResult();
    }

    Contact.OwnerID = contact.OwnerID;

    Context.Attach(Contact).State = EntityState.Modified;

    if (contact.Status == ContactStatus.Approved)
    {
        // If the contact is updated after approval,
        // and the user cannot approve,
        // set the status back to submitted so the update can be
        // checked and approved.
        var canApprove = await AuthorizationService.AuthorizeAsync(User,
            contact,
            ContactOperations.Approve);

        if (!canApprove.Succeeded)
        {
            contact.Status = ContactStatus.Submitted;
        }
    }

    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}

private bool ContactExists(int id)
{
    return Context.Contact.Any(e => e.ContactId == id);
}
}

```

更新 DeleteModel

更新删除页模型，使用授权处理程序来验证用户对联系人拥有删除权限。

```

public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
            User, Contact,
            ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        Contact = await Context.Contact.FindAsync(id);

        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
            User, contact,
            ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        Context.Contact.Remove(Contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}

```

将授权服务注入到视图

目前，UI 显示编辑和删除用户无法修改的数据的链接。用户界面是通过将授权处理程序应用于视图固定的。

插入中的授权服务Views/_ViewImports.cshtml文件以便它可供所有视图：

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

前面的标记添加了多种`using`语句。

更新编辑和删除链接Pages/Contacts/Index.cshtml以便它们在仅呈现具有适当权限的用户：

```
@page
@model ContactManager.Pages.Contacts.IndexModel

 @{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Contact)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Address)
                </td>
                <td>
```

```

        @Html.DisplayFor(modelItem => item.City)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.State)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.Zip)
    </td>

    <td>
        @Html.DisplayFor(modelItem => item.Email)
    </td>

    <td>
        @Html.DisplayFor(modelItem => item.Status)
    </td>
    <td>
@if ((await AuthorizationService.AuthorizeAsync(
    User, item,
    ContactOperations.Update)).Succeeded)
{
    <a asp-page="./Edit" asp-route-id="@item.ContactId">Edit</a>
    <text> | </text>
}
<a asp-page="./Details" asp-route-id="@item.ContactId">Details</a>

@if ((await AuthorizationService.AuthorizeAsync(
    User, item,
    ContactOperations.Delete)).Succeeded)
{
    <text> | </text>
    <a asp-page="./Delete" asp-route-id="@item.ContactId">Delete</a>
}
</td>
</tr>
}
</tbody>
</table>

```

警告

隐藏来自没有更改数据的权限的用户的链接不安全应用程序。隐藏链接进行应用程序更加友好的用户显示唯一有效的链接。用户可以 hack 生成的 Url 来调用编辑和删除在它们自己的数据的操作。Razor 页或控制器必须强制执行访问检查，以保护数据。

更新详细信息

更新的详细信息视图，以便经理可以批准或拒绝联系人：

```

/*Precedng markup omitted for brevity.*@

<dd>
    @Html.DisplayFor(model => model.Contact.Email)
</dd>
<dt>
    @Html.DisplayNameFor(model => model.Contact.Status)
</dt>
<dd>
    @Html.DisplayFor(model => model.Contact.Status)
</dd>
</dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
        User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
        User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-success">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
        User, Model.Contact,
        ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>

```

更新详细信息页模型：

```

public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus status)
    {
        var contact = await Context.Contact.FirstOrDefault(
            m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
            ? ContactOperations.Approve
            : ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User, contact,
            contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}

```

测试已完成的应用程序

如果你使用 Visual Studio Code 或不包括为支持 HTTPS 的测试证书的本地平台上测试：

- 设置 "LocalTest:skipSSL": true 中 `appsettings.Development.json` 文件中跳过 HTTPS 要求。跳过仅在开发计算机上的 HTTPS。

如果应用了联系人：

- 删除中的所有记录 `Contact` 表。
- 重新启动应用插入到数据库。

浏览联系人注册用户。

若要测试已完成的应用程序的简单方法是以启动三个不同的浏览器（或 incognito/InPrivate 版本）。一个在浏览器中注册一个新用户（例如，`test@example.com`）。登录到每个浏览器了不同的用户。验证以下操作：

- 已注册的用户可以查看所有已批准的联系人数据。
- 已注册的用户可以编辑/删除其自己的数据。
- 经理可以批准或拒绝联系人数据。`Details` 视图显示批准和拒绝按钮。
- 管理员可以批准/拒绝和编辑/删除任何数据。

“用户”	选项
test@example.com	可以编辑/删除自己的数据
manager@contoso.com	可以批准/拒绝和编辑/删除拥有数据
admin@contoso.com	可以编辑/删除和批准/拒绝的所有数据

在管理员的浏览器中创建一个联系人。将删除该 URL 复制和编辑从管理员的联系信息。将这些链接粘贴到测试用户的浏览器以验证测试用户无法执行这些操作。

创建初学者应用

- 创建名为“ContactManager”Razor 页应用
 - 创建应用程序与单个用户帐户。
 - 请将其命名“ContactManager”使你的命名空间匹配示例中使用的命名空间。

```
dotnet new razor -o ContactManager -au Individual -uld
```

- `-uld` 指定而不是 SQLite LocalDB

- 添加以下 `Contact` 模型：

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

- 基架 `Contact` 模型：

```
dotnet aspnet-codegenerator razorpage -m Contact -udl -dc ApplicationDbContext -outDir Pages\Contacts --referenceScriptLibraries
```

- 更新 `ContactManager` 中锚定 `Pages/_Layout.cshtml` 文件：

```
<a asp-page="/Contacts/Index" class="navbar-brand">ContactManager</a>
```

- 构建基架初始迁移并更新数据库：

```
dotnet ef migrations add initial  
dotnet ef database update
```

- 测试应用程序通过创建、编辑和删除联系人

设定数据库种子

添加 `SeedData` 类到数据文件夹。如果你已下载示例，你可以复制 `SeedData.cs` 文件为 `数据初学者` 项目的文件夹。

调用 `SeedData.Initialize` 从 `Main`：

```
public class Program  
{  
    public static void Main(string[] args)  
    {  
        var host = BuildWebHost(args);  
  
        using (var scope = host.Services.CreateScope())  
        {  
            var services = scope.ServiceProvider;  
            var context = services.GetRequiredService<ApplicationDbContext>();  
            context.Database.Migrate();  
  
            try  
            {  
                SeedData.Initialize(services, "").Wait();  
            }  
            catch (Exception ex)  
            {  
                var logger = services.GetRequiredService<ILogger<Program>>();  
                logger.LogError(ex, "An error occurred while seeding the database.");  
                throw ex;  
            }  
        }  
  
        host.Run();  
    }  
  
    public static IWebHost BuildWebHost(string[] args) =>  
        WebHost.CreateDefaultBuilder(args)  
            .UseStartup<Startup>()  
            .Build();  
    }  
}
```

测试应用程序设定种子的数据库。如果在联系人中数据库的任何行，不能运行 seed 方法。

其他资源

- [ASP.NET 核心授权实验室](#)。在本教程中引入的安全功能，此实验室将进入更多详细信息。
- [在 ASP.NET Core 中的授权：基于声明的和自定义的简单，角色](#)
- [基于自定义策略的授权](#)

在 ASP.NET Core razor 页授权约定

2018/5/14 • 2 min to read • [Edit Online](#)

作者: [Luke Latham](#)

若要控制在 Razor 页面应用程序的访问的一个方法是在启动时使用授权约定。这些约定，可以为用户授权，并允许匿名用户访问各个页的文件夹。自动本主题中所述的约定应用[授权筛选器](#)来控制访问权限。

[查看或下载示例代码\(如何下载\)](#)

需要授权可访问的页面

使用[AuthorizePage](#)通过约定[AddRazorPagesOptions](#)添加[AuthorizeFilter](#)到页中指定的路径：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

指定的路径是视图引擎路径，这是无需扩展和包含仅正斜杠的 Razor 页根相对路径。

[AuthorizePage 重载](#)当你需要指定一个授权策略才可用。

需要的页的文件夹的访问权

使用[AuthorizeFolder](#)通过约定[AddRazorPagesOptions](#)添加[AuthorizeFilter](#)到所有指定路径上的文件夹中的页：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

指定的路径是视图引擎路径，这是 Razor 页根相对路径。

[AuthorizeFolder 重载](#)当你需要指定一个授权策略才可用。

允许匿名访问的页

使用[AllowAnonymousToPage](#)通过约定[AddRazorPagesOptions](#)添加[AllowAnonymousFilter](#)到指定的路径下的网页：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

指定的路径是视图引擎路径，这是无需扩展和包含仅正斜杠的 Razor 页根相对路径。

允许匿名访问的页面的文件夹

使用[AllowAnonymousToFolder](#)通过约定[AddRazorPagesOptions](#)添加[AllowAnonymousFilter](#)到所有指定路径上的文件夹中的页：

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

指定的路径是视图引擎路径，这是 Razor 页根相对路径。

请注意有关组合授权和匿名访问

这是完全合法，若要指定的页的文件夹需要授权，并指定该文件夹中的页允许匿名访问：

```
// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

但是，反过来，不是如此。不能声明用于匿名访问的页面的文件夹，并指定授权中的页：

```
// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

需要专用的页上的授权不起作用，因为当同时 `AllowAnonymousFilter` 和 `AuthorizeFilter` 筛选器应用到页上，`AllowAnonymousFilter` wins 并控制的访问。

请参阅

- [Razor 页面自定义路由和页面模型提供程序](#)
- [PageConventionCollection类](#)

在 ASP.NET Core 中的简单授权

2018/4/10 • 1 min to read • [Edit Online](#)

通过控制在 MVC 中的授权 `AuthorizeAttribute` 属性和其各种参数。简单地说，应用 `AuthorizeAttribute` 属性设为到控制器的控制器或操作限制访问或对任何经过身份验证的用户的操作。

例如，下面的代码可访问的限制 `AccountController` 到任何经过身份验证的用户。

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

如果你想要应用于操作，而不是控制器的授权，应用 `AuthorizeAttribute` 属性设为操作本身：

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

现在，只有经过身份验证的用户可以访问 `Logout` 函数。

你还可以使用 `AllowAnonymous` 特性以允许使用的未经身份验证用户添加到各个操作进行访问。例如：

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

这将允许仅经过身份验证的用户到 `AccountController`，除 `Login` 操作，这是可访问的所有用户，而不考虑其经过身份验证或未经身份验证 / 匿名状态。

警告

[AllowAnonymous] 绕过所有授权语句。如果将应用组合 [AllowAnonymous] 和任何 [Authorize] 属性然后 Authorize 属性将始终忽略。例如, 如果将应用 [AllowAnonymous] 在控制器级别任何 [Authorize] 特性在同一个控制器上, 或其中的任何操作都将被忽略。

ASP.NET 核心中基于角色的授权

2018/5/4 • 2 min to read • [Edit Online](#)

当创建标识它可能属于一个或多个角色。例如, Tracy 可能属于管理员和用户角色, 同时 Scott 可能仅属于用户角色。如何创建和管理这些角色取决于授权过程的后备存储。角色公开给开发人员通过 `IIsInRole` 方法 `ClaimsPrincipal` 类。

添加角色检查

基于角色的授权检查均为声明性—开发人员将它们嵌入在其代码中, 针对控制器或者控制器中的某个操作指定当前用户必须是的成员来访问请求的资源的角色。

例如, 下面的代码可访问任何操作限制上 `AdministrationController` 谁是其成员的用户到 `Administrator` 角色:

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{}
```

以逗号分隔的列表, 可以指定多个角色:

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{}
```

此控制器将作为仅可访问的成员的用户的 `HRManager` 角色或 `Finance` 角色。

如果您将应用多个属性, 则访问用户必须是指定的所有角色的成员下面的示例要求用户必须是的成员 `PowerUser` 和 `ControlPanelUser` 角色。

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{}
```

通过应用在操作级别的其他角色授权属性, 可以进一步限制访问:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {

        [Authorize(Roles = "Administrator")]
        public ActionResult ShutDown()
        {
        }
    }
}
```

中的上一个代码段成员 `Administrator` 角色或 `PowerUser` 角色可以访问控制器和 `SetTime` 操作, 但只有成员

`Administrator` 角色可以访问 `Shutdown` 操作。

您可以锁定控制器，但允许匿名、未经身份验证访问各项操作。

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

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

基于策略角色检查

此外可以使用新的策略语法中，开发人员将在启动策略注册为授权服务配置的一部分的其中表示角色的要求。这通常发生在 `ConfigureServices()` 中你 `Startup.cs` 文件。

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));
    });
}
```

策略使用应用 `Policy` 属性 `AuthorizeAttribute` 属性：

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

如果你想要指定多个允许的角色中一项要求，则可作为参数来指定这些 `RequireRole` 方法：

```
options.AddPolicy("ElevatedRights", policy =>
    policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

此示例授权属于用户 `Administrator`，`PowerUser` 或 `BackupAdministrator` 角色。

ASP.NET 核心中基于声明的授权

2018/4/10 • 3 min to read • [Edit Online](#)

创建一个标识时它可能会分配一个或多个由受信任方发出的声明。声明是表示哪些使用者名称值对，可以不哪些的主题。例如，你可能具有驱动程序的许可证，本地驱动的许可证颁发机构签发。驱动程序的许可证对其具有你的出生日期。在这种情况下将声明名称 `DateOfBirth`，声明值将是你的出生日期，例如 `8th June 1970` 和颁发者是驱动的许可证颁发机构。基于声明的授权，简单地说，将检查声明的值，并允许对基于该值资源的访问。例如，如果你想夜间俱乐部访问授权过程可能是：

门安全负责人将评估你的出生声明，并且它们是否在授予你访问之前信任颁发者（驱动许可证机构）的日期的值。

标识可以包含具有多个值的多个声明，并且可以包含多个相同类型的声明。

添加声明检查

声明基于的授权检查声明性-开发人员将它们嵌入在其代码中，针对控制器或者控制器中的某个操作指定当前用户必须拥有，并选择性地声明的值必须保留访问的声明请求的资源。要求是基于策略的声明，开发人员必须生成并注册表达声明要求的策略。

最简单类型的声明策略如下所示的声明存在，并且不会检查值。

首先，你需要生成和注册策略。这是作为授权服务配置的一部分，这通常需要一部分花在 `ConfigureServices()` 中你 `Startup.cs` 文件。

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

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

在这种情况下 `EmployeeOnly` 策略将检查是否存在 `EmployeeNumber` 于当前的标识声明。

你随后可应用策略使用 `Policy` 属性 `AuthorizeAttribute` 特性以指定策略名称；

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

`AuthorizeAttribute` 特性可以应用于整个控制器，在这种情况下，仅匹配策略的标识将在控制器上允许向任何操作的访问。

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

如果您有一个控制器受 `AuthorizeAttribute` 属性，但想要允许匿名访问你应用的特定操作 `AllowAnonymousAttribute` 属性。

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

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

大多数声明附带了一个值。创建策略时，你可以指定允许的值的列表。下面的示例将仅成功执行员工的员工数是 1、2、3、4 或 5。

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
            policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

多个策略评估

如果将多个策略应用到的控制器或操作，然后授予访问权限之前也必须传递所有策略。例如：

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {

    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

在上例中任何标识了满足 `EmployeeOnly` 策略可以访问 `Payslip` 在控制器上强制执行该策略的操作。但是为了调用 `UpdateSalary` 标识必须满足的操作同时 `EmployeeOnly` 策略和 `HumanResources` 策略。

如果你希望更复杂的策略，如将出生声明的日期，计算年龄字段从它，然后检查年龄为 21 或更低版本，则需要编

[写自定义策略处理程序。](#)

ASP.NET Core中基于策略的授权

2018/4/20 • 7 min to read • [Edit Online](#)

实际上，[基于角色的授权](#)和[基于声明的授权](#)都使用要求、要求处理程序和预配置的策略。这些构建基块支持在代码中使用授权评估表达式。结果就是，授权结构更加丰富，可重复使用，并且可以测试。

授权策略包含一个或多个要求。授权策略包含一个或多个要求，并在 `Startup.ConfigureServices` 方法中作为授予权服务配置的一部分注册：

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

在前面的示例中，创建了一个“`AtLeast21`”策略。它只有一个要求—，即最低年龄，以参数的形式传递给要求。

将 `[Authorize]` 属性和策略名称配合使用即可应用策略。例如：

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller
{
    public IActionResult Login() => View();

    public IActionResult Logout() => View();
}
```

要求

授权要求是一个可供策略用来评估当前用户主体的数据参数的集合。在我们的“`AtLeast21`”策略中，此要求是单个参数—最低年龄。要求可以实现 [IAuthorizationRequirement](#)，这是一个空标记接口。参数化的最低年龄要求可以按如下方式实现：

```
using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; private set; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

注意

一项要求不需要具有数据或属性。

授权处理程序

授权处理程序负责评估要求的属性。授权处理程序会针对提供的 [AuthorizationHandlerContext](#) 来评估要求，确定是否允许访问。

一项要求可以有多个处理程序。处理程序可以继承 [AuthorizationHandler<TRequirement>](#)，其中的 `TRequirement` 是需处理的要求。另外，一个处理程序也可以通过实现 [IAuthorizationHandler](#) 来处理多个类型的要求。

为一个要求使用一个处理程序

下面是一对一关系的示例，其中的单个最低年龄要求处理程序使用单个要求：

```
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System;
using System.Security.Claims;
using System.Threading.Tasks;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    MinimumAgeRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
                                   c.Issuer == "http://contoso.com"))
        {
            //TODO: Use the following if targeting a version of
            // .NET Framework older than 4.6:
            //     return Task.FromResult(0);
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(
            context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&
                                   c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

前面的代码确定当前的用户主体是否有一个由已知的受信任颁发者颁发的出生日期声明。缺少声明时，无法进行授权，这种情况下会返回已完成的任务。存在声明时，会计算用户的年龄。如果用户满足此要求所定义的最低年龄，则可以认为授权成功。授权成功后，会调用 `context.Succeed`，使用满足的要求作为其唯一参数。

用于多个要求的处理程序

下面是一对多关系的示例，其中的权限处理程序使用三个要求：

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource) ||
                    IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission ||
                      requirement is DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }

    private bool IsOwner(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }

    private bool IsSponsor(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }
}
```

前面的代码遍历 `PendingRequirements`—不包含要求的属性标记为成功。如果用户具有读取权限，他或她必须是所有者或发起人访问请求的资源。如果用户已编辑或删除权限，他或她必须所有者才能访问请求的资源。授权成功后，会调用 `context.Succeed`，使用满足的要求作为其唯一参数。

处理程序注册

处理程序是在配置期间在服务集合中注册的。例如：

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

每个处理程序均通过调用 `services.AddSingleton<IAuthorizationHandler, YourHandlerClass>();` 添加到服务集合。

处理程序应返回什么？

请注意，`Handle` 处理程序示例中的方法不返回值。如何表明状态是成功还是失败？

- 处理程序通过调用 `context.Succeed(IAuthorizationRequirement requirement)` 并传递已成功验证的要求来表示成功。
- 处理程序通常不需要处理失败，因为同一要求的其他处理程序可能会成功。
- 若要确保失败，即使其他要求处理程序成功，调用 `context.Fail`。

设置为 `false` 时，`InvokeHandlersAfterFailure` 属性（在 ASP.NET Core 1.1 及更高版本中提供）会在已调用 `context.Fail` 的情况下不执行处理程序。`InvokeHandlersAfterFailure` 默认为 `true`，这种情况下会调用所有处理程序。这样要求以产生副作用，例如日志记录，这些始终发生即使 `context.Fail` 已在另一个处理程序调用。

为什么需要对一项要求使用多个处理程序？

在需要以 **OR** 逻辑为基础进行评估的情况下，可以针对单个要求实现多个处理程序。例如，Microsoft 的门只能使用门禁卡打开。如果你将门禁卡丢在家中，可以要求前台打印一张临时标签来开门。在这种情况下，只有一个要求，即 `BuildingEntry`，但有多个处理程序，每个处理程序针对单个要求进行检查。

BuildingEntryRequirement.cs

```
using Microsoft.AspNetCore.Authorization;

public class BuildingEntryRequirement : IAuthorizationRequirement
{}
```

BadgeEntryHandler.cs

```

using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class BadgeEntryHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId &&
                                  c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }
}

```

TemporaryStickerHandler.cs

```

using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;
using System.Security.Claims;
using System.Threading.Tasks;

public class TemporaryStickerHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.TemporaryBadgeId &&
                                  c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        // .NET Framework older than 4.6:
        //     return Task.FromResult(0);
        return Task.CompletedTask;
    }
}

```

请确保这两个处理程序已注册。当策略评估 `BuildingEntryRequirement` 时，如果有一个处理程序成功，则策略评估成功。

使用 func 来实现策略

有些情况下，策略很容易用代码实现。可以在通过 `Func<AuthorizationHandlerContext, bool>` 策略生成器配置策略时提供 `RequireAssertion`。

例如，上一个 `BadgeEntryHandler` 可以重写，如下所示：

```
services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == ClaimTypes.BadgeId ||
                 c.Type == ClaimTypes.TemporaryBadgeId) &&
                 c.Issuer == "https://microsoftsecurity")));
});
```

访问处理程序中的 MVC 请求上下文

在授权处理程序中实现的 `HandleRequirementAsync` 方法有两个参数：`AuthorizationHandlerContext` 以及你正在处理的 `TRequirement`。MVC 或 Jabbr 之类的框架可以自由地将任何对象添加到 `Resource` 中的 `AuthorizationHandlerContext` 属性，以便传递额外信息。

例如，MVC 在 [属性中传递 AuthorizationFilterContext](#) `Resource` 实例。可以通过此属性访问 `HttpContext`、`RouteData` 以及 MVC 和 Razor 页面提供的所有其他内容。

对 `Resource` 属性的使用取决于框架。使用 `Resource` 属性中的信息时，授权策略就会局限于特定的框架。应使用 `Resource` 关键字来强制转换 `as` 属性，然后确认该强制转换是否成功，确保代码在其他框架上运行时不会崩溃并抛出 `InvalidOperationException` 异常：

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

在 ASP.NET 核心要求处理程序中的依赖关系注入

2018/4/10 • 1 min to read • [Edit Online](#)

必须注册授权处理程序在配置期间服务集合中 (使用[依赖关系注入](#))。

假设您有您想要评估的授权处理程序内的规则的存储库和服务集合中注册了该存储库。授权将解析和您的构造函数中的插入。

例如, 如果你想要使用 ASP.NET 的日志记录你想要插入的基础结构 `ILoggerFactory` 到您的处理程序。此类处理可能如下所示:

```
public class LoggingAuthorizationHandler : AuthorizationHandler<MyRequirement>
{
    ILogger _logger;

    public LoggingAuthorizationHandler(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger(this.GetType().FullName);
    }

    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MyRequirement requirement)
    {
        _logger.LogInformation("Inside my handler");
        // Check if the requirement is fulfilled.
        return Task.CompletedTask;
    }
}
```

将注册处理程序替换 `services.AddSingleton()`:

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

实例的处理程序将你的应用程序启动时, 创建和插入的已注册的 DI 将 `ILoggerFactory` 到您的构造函数。

注意

使用实体框架的处理程序不应注册为单一实例。

ASP.NET 核心中基于资源的授权

2018/5/17 • 5 min to read • [Edit Online](#)

授权策略取决于要访问的资源。请考虑具有 `author` 属性的文档。仅作者允许更新文档。因此，该文档必须检索从数据存储区授权评估才能发生。

在绑定数据之前和的页处理或加载文档的操作执行之前会进行属性评估。出于这些原因，使用的声明性授权 `[Authorize]` 属性不能满足要求。相反，你可以调用自定义授权方法—称为命令性授权的样式。

使用[示例应用\(如何下载\)](#)来浏览本主题中所述的功能。

使用受授权的用户数据创建 ASP.NET Core 应用包含的示例应用程序使用的基于资源的授权。

使用命令性授权

作为实现授权 `IAuthorizationService` 服务并在服务集合中注册 `Startup` 类。该服务可通过[依赖关系注入](#)对页处理程序或操作。

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                             IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
}
```

`IAuthorizationService` 有两个 `AuthorizeAsync` 方法重载：一个接收资源和策略名称和其他接受资源和要求来评估的列表。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                                         object resource,
                                         IEnumerable<IAuthorizationRequirement> requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                                         object resource,
                                         string policyName);
```

在下面的示例中，要保护的资源加载到一个自定义 `Document` 对象。`AuthorizeAsync` 重载进行调用以确定是否允许当前用户编辑提供的文档。自定义“`EditPolicy`”授权策略被分解为决策因子。请参阅[自定义基于策略的授权](#)创建授权策略的详细信息。

注意

下面的代码示例假定已经运行了身份验证和集 `User` 属性。

- [ASP.NET Core 2.x](#)

- [ASP.NET Core 1.x](#)

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
        .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

基于资源的处理程序编写

编写处理程序的基于资源的授权不大的不同比[编写纯要求处理程序](#)。创建自定义要求类，并实现要求处理程序类。处理程序类指定的要求和资源类型。例如，处理程序利用 `SameAuthorRequirement` 要求和 `Document` 资源将如下所示：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    SameAuthorRequirement requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

注册的要求和中的处理程序 `Startup.ConfigureServices` 方法：

```

services.AddMvc();

services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();

```

操作要求

如果你正在进行决策基于 CRUD（创建、读取、更新、删除）操作的结果，使用 [OperationAuthorizationRequirement](#) 帮助器类。此类，可为每个操作类型编写单个处理程序而不是单独的类。若要使用此选项，提供一些操作名称：

```

public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}

```

处理程序实现，如下所示，使用 `OperationAuthorizationRequirement` 要求和 `Document` 资源：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                OperationAuthorizationRequirement requirement,
                                                Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

```

前面的处理程序验证使用的资源、用户的标识和要求的操作 `Name` 属性。

若要调用的操作资源处理程序，指定该操作时调用 `AuthorizeAsync` 在页处理程序或操作。下面的示例确定是否允许经过身份验证的用户若要查看提供的文档。

注意

下面的代码示例假定已经运行了身份验证和集 `User` 属性。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
        .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

如果授权成功，则返回查看文档的页。如果授权失败而用户进行身份验证，返回 `ForbidResult` 通知授权失败的任何身份验证中间件。A `ChallengeResult` 时必须执行身份验证返回。对于交互式浏览器客户端，可能适合将用户重定向到登录页。

ASP.NET 核心 mvc 视图基于授权

2018/4/10 • 1 min to read • [Edit Online](#)

开发人员通常需要显示、隐藏或修改基于当前的用户标识的用户界面。你可以访问授权服务在服务内通过的 MVC 视图 [依赖关系注入](#)。若要将授权服务注入到 Razor 视图中，使用 `@inject` 指令：

```
@using Microsoft.AspNetCore.Authorization  
@inject IAuthorizationService AuthorizationService
```

如果希望每个视图中的授权服务，将放置 `@inject` 指令插入 `*_ViewImports.cshtml` 文件视图* 目录。有关详细信息，请参阅 [视图中的依赖关系注入](#)。

使用插入的授权服务来调用 `AuthorizeAsync` 中完全相同的方式将检查期间 [基于资源的授权](#)：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
@if ((await AuthorizationService.AuthorizeAsync(User, "PolicyName")).Succeeded)  
{  
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>  
}
```

在某些情况下，资源将视图模型。调用 `AuthorizeAsync` 中完全相同的方式将检查期间 [基于资源的授权](#)：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit)).Succeeded)  
{  
    <p><a class="btn btn-default" role="button"  
        href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>  
}
```

在前面的代码中，该模型作为资源应采取的策略评估传递纳入考虑范围。

警告

不要依赖于与唯一的授权检查的应用程序的 UI 元素的切换可见性。隐藏的 UI 元素可能无法完全阻止访问到其关联的控制器操作。例如，考虑前面的代码段中的按钮。用户可以调用 `Edit` 操作方法如果他或她知道的相对资源 URL 是 `"/Document/Edit/1"`。为此，`Edit` 操作方法应执行其自己的授权检查。

使用 ASP.NET Core 中的特定方案授权

2018/4/10 • 3 min to read • [Edit Online](#)

在某些情况下，如单页面应用程序 (Spa) 很常见的是使用多个身份验证方法。例如，应用可能会使用基于 cookie 的身份验证登录和 JWT 持有者身份验证的 JavaScript 请求。在某些情况下，应用程序可能具有身份验证处理程序的多个的实例。例如，其中一个包含基本标识的两个 cookie 处理程序，另一个时创建已触发多因素身份验证 (MFA)。用户请求要求额外的安全的操作，因此，可能会触发 MFA。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

当在身份验证过程中配置身份验证服务，名为身份验证方案。例如：

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication()
        .AddCookie(options => {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options => {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });
}
```

已在前面的代码中，添加两个身份验证处理程序：一个用于 cookie，一个用于持有者。

注意

指定的默认方案会导致 `HttpContext.User` 属性设置为该标识。如果不需要该行为，禁用调用的无参数形式 `AddAuthentication`。

选择带有 `Authorize` 属性的方案

在授权，终端应用指示要使用的处理程序。选择与应用程序将授权通过传递到身份验证方案的以逗号分隔列表的处理程序 `[Authorize]`。`[Authorize]` 属性指定的身份验证方案或方案，可使用无论配置默认值。例如：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
{
    // Requires the following imports:
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.JwtBearer;
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
```

在前面的示例中，cookie 和持有者处理程序运行，并有机会在创建并追加当前用户的标识。通过指定一种方案，相应的处理程序运行。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
[Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public class MixedController : Controller
```

在前面的代码中，仅具有“Bearer”方案处理程序运行。任何基于 cookie 的标识将被忽略。

选择使用策略的方案

如果想要指定在所需的架构[策略](#)，你可以设置 `AuthenticationSchemes` 集合添加你的策略时：

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {
        policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement());
    });
});
```

在前面的示例中，“Over18”策略仅运行针对“Bearer”处理程序创建的标识。通过设置来使用策略 `[Authorize]` 特性的 `Policy` 属性：

```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
```

ASP.NET Core 中的数据保护

2018/4/10 • 1 min to read • [Edit Online](#)

- [数据保护简介](#)
- [数据保护 API 入门](#)
- [使用者 API](#)
 - [使用者 API 概述](#)
 - [目标字符串](#)
 - [目标层次结构和多租户](#)
 - [哈希密码](#)
 - [限制受保护负载的生存期](#)
 - [取消保护已撤消其密钥的负载](#)
- [配置](#)
 - [配置 ASP.NET Core 数据保护](#)
 - [默认设置](#)
 - [计算机范围的策略](#)
 - [非 DI 感知方案](#)
- [扩展性 API](#)
 - [核心加密扩展性](#)
 - [密钥管理扩展性](#)
 - [其他 API](#)
- [实现](#)
 - [已验证的加密详细信息](#)
 - [子项派生和已验证的加密](#)
 - [上下文标头](#)
 - [密钥管理](#)
 - [密钥存储提供程序](#)
 - [静态密钥加密](#)
 - [密钥永久性和设置](#)
 - [密钥存储格式](#)
 - [短数据保护提供程序](#)
- [兼容性](#)

- 在 ASP.NET Core 中替换 ASP.NET

ASP.NET Core Data Protection

2018/4/10 • 6 min to read • [Edit Online](#)

Web 应用程序通常需要存储安全敏感数据。Windows 提供 DPAPI 用于桌面应用程序，但这不适用于 web 应用程序。ASP.NET 核心数据保护堆栈提供一个简单、易于使用的加密 API，开发人员可以使用来保护数据，包括密钥管理和旋转。

ASP.NET 核心数据保护堆栈旨在用作的长期替代在 ASP.NET 中的元素 1.x-4.x。它旨在解决许多旧加密堆栈的不足之处同时为大多数现代应用程序都可能会遇到的使用情况下提供的现成可用的解决方案。

问题陈述

总体问题语句中的单个句子可以用于简单地所述：我需要保持受信任的信息，以便于以后检索，但我不信任的持久性机制。在 web 术语中，这可能会写“我需要为通过不受信任的客户端往返受信任的状态。”

此规范的示例是身份验证 cookie 或持有者令牌。服务器会生成“我是 Groot，具有 xyz 权限”令牌并将它传递到客户端。在将来的某个日期，客户端将显示该令牌发送回服务器，但服务器需要某种类型的客户端尚未伪造令牌的保证。因此第一个要求：真实性（也称为完整性、防校对）。

由于所保持的状态由服务器信任的我们预计这种状态，可能包含特定于操作环境的信息。这可能是或的服务器特定数据的其他部分的文件路径、权限、句柄或其他间接引用的形式。此类信息应通常不得泄露给不受信任的客户端。因此第二个要求：保密性。

最后，由于现代应用程序已被组件化，我们已了解是各个组件将想要利用此系统而不考虑其他组件在系统中。例如，如果持有者令牌组件使用此堆栈，它应运行，而不从一种反 CSRF 机制，也可能使用相同的堆栈的干扰。因此最后一项要求：隔离。

我们可以提供进一步的约束为了缩小我们的要求的范围。我们假定加密系统内运行的所有服务都同样受信任而数据不需要使用外部下我们直接控制服务或生成。此外，我们需要操作是尽可能快，因为 web 服务的每个请求可能会经过加密系统一个或多个时间。这样，可以对称加密适合我们的方案，而且我们可以如具有所需的时间之前折扣非对称加密。

设计理念

通过标识具有现有堆栈的问题，我们已开始。我们有的我们调查了现有的解决方案的布局，并结束，任何现有解决方案相当了我们查找的功能。我们然后工程处理基于几个指导原则的解决方案。

- 系统应提供简单的配置。理想情况下，系统将零配置并开发人员无法按完全运行。在开发人员需要配置（如密钥存储库）的特定方面的情况下，应特别注意简化这些特定的配置。
- 提供一个简单的面向使用者的 API。API 应易于正确使用且难以不正确地使用。
- 开发人员不应了解密钥管理原则。系统应处理算法选择和开发人员代表密钥的生存期。理想情况下开发人员甚至不应有权访问原始密钥材料。
- 应存放在可能的情况下保护密钥。系统应找出适当的默认保护机制，并自动应用。

牢记这些原则与我们开发一个简单、[易于使用](#)数据保护堆栈。

ASP.NET 核心数据保护 API 主要不用于机密负载的无限期持久性。其他技术喜欢 Windows CNG DPAPI 和 Azure Rights Management 更适合于以下场景：无限期存储，并且它们的相应强密钥管理功能。也就是说，无需进行任何开发人员禁止使用 ASP.NET Core 数据保护 API 进行长期保护的机密数据。

读者

数据保护系统分为五个主要的软件包。这些 API 的各个方面目标三个主要受众；

1. [使用者 API 概述](#) 目标应用程序和 framework 开发人员。

"我不想要了解有关堆栈的运行方式或配置方式。我只想要执行某项操作中的作为简单的方式尽可能与高概率的成功使用 API。"

2. [配置 API](#) 面向应用程序开发人员和系统管理员。

"我需要告诉我的环境需要非默认路径或设置数据保护系统。"

3. [扩展性 API](#) 目标开发人员负责实现自定义策略。这些 API 的使用情况将限制为极少数情况下，并且可以出现，安全感知开发人员。

"我需要更换系统中的整个组件，因为我具有真正独特的行为要求。我愿意以了解极其使用组成部分的 API 图面以生成满足我的要求的插件。"

包布局

数据保护堆栈包含五个的包。

- Microsoft.AspNetCore.DataProtection.Abstractions 包含基本的 IDataProtectionProvider 和 IDataProtector 接口。它还包含有助于使用这些类型（例如，重载 IDataProtector.Protect）的有用的扩展方法。请参阅[使用者接口部分](#)以了解更多信息。如果其他人负责实例化数据保护系统，并且你只需使用 API，则需要引用 Microsoft.AspNetCore.DataProtection.Abstractions。
- Microsoft.AspNetCore.DataProtection 包含数据保护系统，包括核心加密操作、密钥管理、配置和可扩展性的核心的实现。如果你要负责实例化数据保护系统（例如，将其添加到 IServiceCollection）或修改或扩展其行为，你将希望引用 Microsoft.AspNetCore.DataProtection。
- Microsoft.AspNetCore.DataProtection.Extensions 包含其他 API 的开发人员可能会发现很有用，但这不属于核心包中。例如，此程序包包含一个简单的“实例化指向没有依赖关系注入安装程序的特定密钥的存储目录系统”API（[详细信息](#)）。它还包含用于限制受保护负载（[详细信息](#)）的生存期的扩展方法。
- Microsoft.AspNetCore.DataProtection.SystemWeb 可安装到现有 ASP.NET 4.x 应用程序将重定向其操作改为使用新的数据保护堆栈。请参阅[兼容性](#)有关详细信息。
- Microsoft.AspNetCore.Cryptography.KeyDerivation 提供的哈希例程 PBKDF2 密码的实现，并且可以由系统需要安全地处理用户密码。请参阅[哈希处理密码](#)有关详细信息。

要开始使用 ASP.NET Core 中的数据保护 API

2018/4/10 • 2 min to read • [Edit Online](#)

在其最简单、保护数据包括以下步骤：

1. 从数据保护提供程序创建一个数据保护程序。
2. 调用 `Protect` 与你想要保护的数据的方法。
3. 调用 `Unprotect` 想要将返回转换为纯文本的数据的方法。

大多数框架和应用模型，如 ASP.NET 或 SignalR，已配置数据保护系统，并将其添加到通过依赖关系注入访问的服务容器。下面的示例演示配置依赖关系注入的服务容器和注册数据保护堆栈、接收通过 DI 数据保护提供程序、创建保护程序和保护然后正在取消保护数据

```

using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlALTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
*/

```

创建一个保护程序时必须提供一个或多个[目的字符串](#)。目的字符串都提供了使用者之间的隔离。例如，使用“green”的目的字符串创建一个保护程序将无法取消保护数据的目的为“purple”提供的保护程序。

提示

实例 `IDataProtectionProvider` 和 `IDataProtector` 是线程安全的多个调用方。它具有一个组件获取的引用后应，`IDataProtector` 通过调用 `CreateProtector`，它将使用该引用，以便多个调用 `Protect` 和 `Unprotect`。

调用 `Unprotect` 将引发 `CryptographicException`，如果无法验证或中译解出来的受保护的负载。某些组件可能想要忽略错误期间取消保护操作；组件它读取身份验证 cookie 可能处理此错误和请求则将视为根本具有任何 cookie，而不使迫切地请求失败。需要此行为的组件应专门捕获 `CryptographicException`，而不是忽略所有异常。

ASP.NET Core 的使用者 API

2018/4/10 • 1 min to read • [Edit Online](#)

- [使用者 API 概述](#)
- [目标字符串](#)
- [目标层次结构和多租户](#)
- [哈希密码](#)
- [限制受保护负载的生存期](#)
- [取消保护已撤消其密钥的负载](#)

有关 ASP.NET 核心的使用者 API 概述

2018/4/10 • 4 min to read • [Edit Online](#)

`IDataProtectionProvider` 和 `IDataProtector` 接口是通过该使用者使用数据保护系统的基本接口。它们位于 `Microsoft.AspNetCore.DataProtection.Abstractions` 包。

IDataProtectionProvider

提供程序接口表示数据保护系统的根目录。它不能直接用于保护或取消保护数据。相反，使用者必须获得对引用 `IDataProtector` 通过调用 `IDataProtectionProvider.CreateProtector(purpose)`，其中，目的是描述预期的使用者用例的字符串。请参阅 [目的字符串](#) 着眼于此参数以及如何选择适当的值的更多的信息。

IDataProtector

保护程序接口将返回通过调用 `CreateProtector`，和它的使用者可以用于执行此接口保护或取消保护操作。

若要保护的数据片段，将数据传递给 `Protect` 方法。基本接口定义的将 `byte[] -> byte[]` 的方法，但没有还将字符串转换的重载（提供作为扩展方法）`-> string`。提供两种方法的安全性是相同的；开发人员应选择任何重载是最方便对其用例。而不考虑的重载选择，则返回保护方法现在受保护（enciphered 和防考验）和应用程序可以将其发送到不受信任的客户端。

若要取消对以前受保护的数据片段的保护，将传递到受保护的数据 `Unprotect` 方法。（有 `byte[] -> byte[]` 基于和基于字符串的重载，为开发人员方便起见。）如果受保护的负载由以前调用生成 `Protect` 此同一 `IDataProtector`、`Unprotect` 方法将返回原始的未受保护的负载。如果已被篡改或已由不同的受保护的负载 `IDataProtector`、`Unprotect` 方法会引发 `CryptographicException`。

与不同的相同的概念 `IDataProtector` 起回目的概念。如果两个 `IDataProtector` 实例生成从同一根 `IDataProtectionProvider` 但通过不同的用途的调用中的字符串 `IDataProtectionProvider.CreateProtector`，则它们正在被视为 [不同的保护程序](#)，和一个将无法取消保护由其他生成的负载。

使用这些接口

对于 DI 感知的组件，预期的用法是，该组件需要 `IDataProtectionProvider` 在其构造函数的参数，并实例化组件时，DI 系统可以自动提供此服务。

注意

某些应用程序（如控制台应用程序或 ASP.NET 4.x 应用程序）可能不是 DI 感知因此不能使用此处所述的机制。有关这些方案，请查阅 [非 DI 感知的情境](#) 有关获取的实例的详细信息的文档 `IDataProtection` 而无需通过 DI 的提供程序。

下面的示例演示三个概念：

1. [添加数据保护系统](#) 到服务容器
2. 使用 DI 接收的实例 `IDataProtectionProvider`，和
3. 创建 `IDataProtector` 从 `IDataProtectionProvider` 并使用它来保护和取消保护数据。

```

using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZzh1AltZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
*/

```

包 `Microsoft.AspNetCore.DataProtection.Abstractions` 包含的扩展方法 `IServiceProvider.GetDataProtector` 为开发人员方便起见。它作为单个操作封装这两个检索 `IDataProtectionProvider` 从服务提供商和调用 `IDataProtectionProvider.CreateProtector`。下面的示例演示其用法。

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}
```

提示

实例 `IDataProtectionProvider` 和 `IDataProtector` 是线程安全的多个调用方。它具有一个组件获取的引用后应，`IDataProtector` 通过调用 `CreateProtector`，它将使用该引用，以便多个调用 `Protect` 和 `Unprotect`。调用 `Unprotect` 将引发 `CryptographicException`，如果无法验证或中译解出来的受保护的负载。某些组件可能想要忽略错误期间取消保护操作；组件它读取身份验证 cookie 可能处理此错误和请求则将视为根本具有任何 cookie，而不使迫切地请求失败。需要此行为的组件应专门捕获 `CryptographicException`，而不是忽略所有异常。

在 ASP.NET 核心目的字符串

2018/4/10 • 3 min to read • [Edit Online](#)

使用组件 `IDataProtectionProvider` 必须传递一个唯一目的参数 `CreateProtector` 方法。目的参数是固有的数据保护系统中，安全的因为它提供了加密的使用者之间的隔离，即使根加密密钥是相同的。

当使用者指定目的时，目的字符串用于以及根加密密钥派生加密子项唯一到该使用者。这将隔离应用程序中其他加密使用者的使用者：其他组件不能读取其有效负载，并且它无法读取任何其他组件的负载。这种隔离还会呈现不可行的整个类别的针对组件的攻击。



在上图中，`IDataProtector` 实例 A 和 B 无法读取对方的负载，仅自己。

目的字符串不一定是机密。它只应是唯一其他任何功能良好组件不断将提供相同的目的字符串。

提示

使用该组件使用的数据保护 API 的命名空间和类型名称是一个很好的经验法则，如下所示此信息将不会发生冲突的做法。

Contoso 编写的组件，它负责 minting 持有者令牌可能 `Contoso.Security.BearerToken` 用作其用途字符串。或者-甚至更好地-它可能使用 `Contoso.Security.BearerToken.v1` 作为其用途字符串。追加版本号，为未来的版本，以 `Contoso.Security.BearerToken.v2` 用作其用途，并且负载到位会完全独立于另一个不同的版本。

由于目的参数 `CreateProtector` 是一个字符串数组，上述无法已改为指定为

`["Contoso.Security.BearerToken", "v1"]`。这允许建立目的的层次结构，并打开与数据保护系统的多租户方案的可能性。

警告

组件不应允许不受信任的用户输入要用于目的链的输入的唯一来源。

例如，考虑 `Contoso.Messaging.SecureMessage` 负责存储安全消息的组件。如果安全消息的组件是为了调用 `CreateProtector(["username"])`，则恶意用户可能创建的帐户用户名“`Contoso.Security.BearerToken`”在尝试获取要调用的组件 `CreateProtector(["Contoso.Security.BearerToken"])`，从而无意中造成安全消息传递可以预见到身份验证令牌的 mint 负载的系统。

将消息传递组件的更好目的链 `CreateProtector(["Contoso.Messaging.SecureMessage", "User: username"])`，它提供适当的隔离。

通过提供隔离和行为 `IDataProtectionProvider`，`IDataProtector`，和目的如下所示：

- 为给定 `IDataProtectionProvider` 对象, `CreateProtector` 方法将创建 `IDataProtector` 对象唯一绑定到同时 `IDataProtectionProvider` 对象创建它, 并已传递到方法的目的参数。
- 目的参数不得为空。(如果目的指定为一个数组, 这意味着数组不能为零长度和数组的所有元素必须都为非 `null`。) 非空字符串目的从技术上讲允许但不建议这样做。
- 两个用途自变量是等效的当且仅当它们包含相同的字符串(使用序号比较器)中的顺序相同。单一用途参数等效于相应的单个元素目的数组。
- 两个 `IDataProtector` 对象相等, 当且仅当它们在创建等效项从 `IDataProtectionProvider` 对象, 并且等效目的形参。
- 为给定 `IDataProtector` 对象、调用 `Unprotect(protectedData)` 将返回原始 `unprotectedData` 当且仅当 `protectedData := Protect(unprotectedData)` 有关等效 `IDataProtector` 对象。

注意

我们不会考虑的如果某些组件有意选择已知与另一个组件发生冲突的目的字符串。此类组件将考虑恶意内容, 实质上是和此系统并不旨在中, 恶意代码已在运行的工作进程中提供的安全保证。

目的层次结构和 ASP.NET Core 中的多租户

2018/4/10 • 2 min to read • [Edit Online](#)

由于 `IDataProtector` 也是隐式 `IDataProtectionProvider`，目的可以链接在一起。在这个意义上来说，

`provider.CreateProtector(["purpose1", "purpose2"])` 等效于

`provider.CreateProtector("purpose1").CreateProtector("purpose2")`。

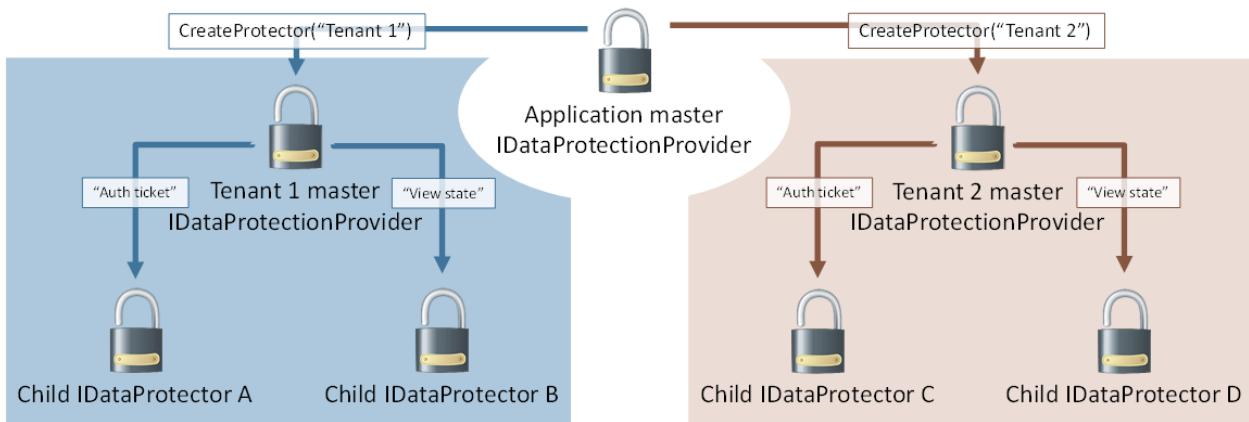
这允许通过数据保护系统的一些有趣的层次结构关系。前面的示例

中 `Contoso.Messaging.SecureMessage`, `SecureMessage` 组件可以调用

`provider.CreateProtector("Contoso.Messaging.SecureMessage")` 一次前期和到专用缓存结果 `_myProvide` 字段。然后可以通过调用创建将来的保护程序 `_myProvider.CreateProtector("User: username")`，并且这些保护程序会使用用于保护单个消息。

这可以还翻转。考虑单个逻辑应用程序可以使用其自己的身份验证和状态管理系统配置多个租户 (CMS 看起来合理) 和每个租户的主机。涵盖应用程序具有单一的主提供程序，并且它会调用

`provider.CreateProtector("Tenant 1")` 和 `provider.CreateProtector("Tenant 2")` 为每个租户提供其自己的数据保护系统的独立的切片。租户然后无法派生自己单独的保护程序，基于其自己的需求，但无论硬他们将在尝试在不能创建其中发生冲突的保护程序与其他任何租户系统中。以图形方式，这表示如下。



警告

这假定涵盖应用程序的控制哪些 API 可供各个租户和租户不能在服务器上执行任意代码。如果租户可以执行任意代码，用户无法执行私有反射来中断隔离保证，或它们只是无法直接读取主密钥材料和派生任何子项他们所需。

数据保护系统实际上使用一种在其默认现成可用配置中的多租户。默认情况下主密钥材料存储在辅助进程帐户的用户配置文件文件夹中（或注册表中的，为 IIS 应用程序池标识）。但它实际上相当通常使用单个帐户来运行多个应用程序，并且因此这些应用程序将最终共享主密钥材料。若要解决此问题，数据保护系统自动将唯一的按应用程序标识符作为整体的目的链中的第一个元素。此隐式目的提供给 [将单个应用程序隔离](#) 从另一个通过有效地将每个应用程序，像在系统和保护程序创建过程中的唯一租户上去与上面的图像相同。

在 ASP.NET 核心中的哈希密码

2018/5/4 • 1 min to read • [Edit Online](#)

基本数据保护代码包括包 `Microsoft.AspNetCore.Cryptography.KeyDerivation` 其中包含加密密钥派生函数。此包是一个独立组件，并不依赖于数据保护系统的其余部分。可以完全单独使用它。源共存于基本为方便起见数据保护代码。

包当前提供一种方法 `KeyDerivation.Pbkdf2` 这样，哈希密码使用 **PBKDF2 算法**。此 API 已非常类似于 .NET Framework 的现有 `Rfc2898DeriveBytes` 类型，但有三个重要区别：

1. `KeyDerivation.Pbkdf2` 方法支持使用多个 PRFs (当前 `HMACSHA1`, `HMACSHA256`, 和 `HMACSHA512`)，而 `Rfc2898DeriveBytes` 类型仅支持 `HMACSHA1`。
2. `KeyDerivation.Pbkdf2` 方法检测到当前操作系统，然后尝试选择最优化的实现的例程，提供更好的性能，在某些情况下，选择。(在 Windows 8 中，它提供约 10 倍的吞吐量 `Rfc2898DeriveBytes`。)
3. `KeyDerivation.Pbkdf2` 方法要求调用方指定的所有参数 (salt, PRF 和迭代计数)。`Rfc2898DeriveBytes` 类型提供的这些默认值。

```
using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Enter a password: ");
        string password = Console.ReadLine();

        // generate a 128-bit salt using a secure PRNG
        byte[] salt = new byte[128 / 8];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(salt);
        }
        Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

        // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
        string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: salt,
            prf: KeyDerivationPrf.HMACSHA1,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
        Console.WriteLine($"Hashed: {hashed}");
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
 * Hashed: /00oOer10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */
```


限制在 ASP.NET Core 的受保护负载的生存期

2018/4/10 • 3 min to read • [Edit Online](#)

有一些应用程序开发人员希望创建将在一段时间后过期的受保护的负载的情形。例如，受保护的负载可能表示只为有效一小时的密码重置令牌。当然也可以为开发人员创建他们自己包含的嵌入的到期日期的负载格式和高级开发人员可能想要执行此操作，但对于开发人员的大多数管理这些过期时间可能变得需要很长时间。

若要简化此过程对于我们的开发人员受众，包[Microsoft.AspNetCore.DataProtection.Extensions](#)用于创建在一段时间后自动过期的负载包含实用工具 API。这些 API 的挂起注销 `ITimeLimitedDataProtector` 类型。

API 使用情况

`ITimeLimitedDataProtector` 接口是用于保护和取消保护的限时 / 自即将到期的负载的核心接口。若要创建的实例 `ITimeLimitedDataProtector`，你需要先正则表达式的实例 `IDataProtector` 构造具有特定的用途。一次 `IDataProtector` 实例可用时，调用 `IDataProtector.ToTimeLimitedDataProtector` 扩展方法，能获得重新保护程序内置到期功能。

`ITimeLimitedDataProtector` 公开以下 API 图面和扩展方法：

- `CreateProtector` (字符串目的) : `ITimeLimitedDataProtector`-此 API 已类似于现有 `IDataProtectionProvider.CreateProtector` 在于它可以用于创建 [目的链](#) 从根的限时保护程序。
- `Protect(byte[] plaintext, DateTimeOffset expiration)` : `byte[]`
- 保护 (字节 [] 纯文本、`TimeSpan` 生存期) : `byte[]`
- 保护 (字节 [] 纯文本) : `byte[]`
- 保护 (字符串纯文本, `DateTimeOffset` 到期) : 字符串
- 保护 (`TimeSpan` 生存期中的字符串纯文本) : 字符串
- 保护 (字符串纯文本) : 字符串

除了核心 `Protect` 方法需要仅是纯文本形式，存在一些新重载允许指定的负载的到期日期。到期日期可以指定为绝对日期 (通过 `DateTimeOffset`) 或相对的时间 (从当前系统时间，通过 `TimeSpan`)。如果调用不带过期的重载，则负载假定为永远不会过期。

- 取消保护 (字节 [] `protectedData`, 出 `DateTimeOffset` 到期) : `byte[]`
- 取消保护 (字节 [] `protectedData`) : `byte[]`
- 取消保护 (出 `DateTimeOffset` 过期字符串 `protectedData`) : 字符串
- 取消保护 (字符串 `protectedData`) : 字符串

`Unprotect` 方法返回原始的未受保护的数据。如果尚未超过负载，绝对过期则返回为可选输出参数以及原始的未受保护数据。如果负载已过期，则取消保护方法的所有重载将都引发 `CryptographicException`。

警告

它具有不建议使用这些 API 来保护需要长期或无限期暂留的负载。“我可以承受为要在一个月后永久性不可恢复的受保护负载？”可用作一个很好的经验法则；如果问题的回答是没有然后开发人员应考虑备用 API。

使用下面的示例 [非 DI 代码路径](#) 用于实例化数据保护系统的。若要运行此示例，请确保先添加对

Microsoft.AspNetCore.DataProtection.Extensions 包的引用。

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\myapp-keys\"));  
        var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.Write("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk70k
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at ...'>>
 */
```

取消保护已吊销在 ASP.NET 核心中键的有效负载

2018/4/10 • 3 min to read • [Edit Online](#)

ASP.NET 核心数据保护 API 主要不用于机密负载的无限期持久性。其他技术喜欢 Windows CNG DPAPI 和 Azure Rights Management 更适合于以下场景：无限期存储，并且它们的相应强密钥管理功能。也就是说，无需进行任何开发人员禁止使用 ASP.NET Core 数据保护 API 进行长期保护的机密数据。密钥绝不会移除从密钥链中，因此 `IDataProtector.Unprotect`，只要键为可用，有效始终可以恢复现有的负载。

但是，则会产生问题在开发人员尝试取消保护数据作为保护与吊销键，`IDataProtector.Unprotect` 在这种情况下将引发异常。这些类型的负载可以轻松地重新创建这些系统，以及在坏的情况下站点访问者可能需要重新登录，这可能是特别适用于短期或临时负载（如身份验证令牌）的。但持久化负载，具有 `Unprotect` 抛出可能导致不可接受的数据丢失。

IPersistedDataProtector

若要支持允许负载为即使在遇到吊销密钥时未受保护的方案，数据保护系统包含 `IPersistedDataProtector` 类型。若要获取其实例 `IPersistedDataProtector`，只需获取其实例 `IDataProtector` 在正常情况和重试强制转换 `IDataProtector` 到 `IPersistedDataProtector`。

注意

并非所有 `IDataProtector` 实例可以强制转换为 `IPersistedDataProtector`。开发人员应使用 C# 作为运算符或类似以避免运行时异常导致通过无效强制转换，并且应在准备好正确地处理失败案例。

`IPersistedDataProtector` 公开以下 API 图面：

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
    out bool requiresMigration, out bool wasRevoked) : byte[]
```

此 API 采用（作为字节数组）的受保护的负载，并返回未受保护的负载。没有任何基于字符串的重载。Out 参数的两个如下所示。

- `requiresMigration`：将设置为 `true` 时用于保护此负载密钥不再是活动默认密钥，例如，用于保护此负载的关键是旧，密钥滚动操作包含以来执行的位置。调用方可能想要考虑重新保护具体取决于其业务需求的负载。
- `wasRevoked`：将设置为 `true` 时用来保护此负载的密钥已被吊销。

警告

传递时应格外谨慎 `ignoreRevocationErrors: true` 到 `DangerousUnprotect` 方法。如果调用此方法后的 `wasRevoked` 值是为 `true`，则用来保护此负载的密钥已被吊销，并且负载的真实性应被视为可疑。在这种情况下，才继续操作操作系统上不受保护的负载，如果你具有一些单独的保障，它是可信的例如所来自的安全数据库，而不是由不受信任的 web 客户端发送。

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
```

```

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector = services.GetDataProtector("Sample.DangerousUnprotect");
        Console.WriteLine("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring...");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
            var unprotectedPayload = persistedProtector.DangerousUnprotect(
                protectedData: protectedData,
                ignoreRevocationErrors: true,
                requiresMigration: out requiresMigration,
                wasRevoked: out wasRevoked);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
            Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Input: Hello!
 * Protected payload: A3B1C2D4E5F6G7H8I9J0K1L2M3N4O5P6Q7R8S9T0U1V2W3X4Y5Z6
 * Round-tripped payload: Hello!
 * Revoking all keys in the key ring...
 * Calling Unprotect...
 * Unprotected payload: Hello!
 * IPersistedDataProtector: Unhandled exception of type 'System.NotSupportedException' occurred in System.DataProtection.dll
 * IPersistedDataProtector: Object reference not set to an instance of an object
 * DangerousUnprotect...
 * DangerousUnprotect: Unhandled exception of type 'System.NotSupportedException' occurred in System.DataProtection.dll
 * DangerousUnprotect: Object reference not set to an instance of an object
 */

```

```
* Protected payload: C4UJ8LH1ZUCLZVBNZBZ...
* Round-tripped payload: Hello!
* Revoking all keys in the key ring...
* Calling Unprotect...
* CryptographicException: The key {...} has been revoked.
* Calling DangerousUnprotect...
* Unprotected payload: Hello!
* Requires migration = True, was revoked = True
*/
```

ASP.NET Core 中的数据保护配置

2018/4/10 • 1 min to read • [Edit Online](#)

访问这些主题，了解 ASP.NET Core 中的数据保护配置：

- [配置 ASP.NET Core 数据保护](#)

配置 ASP.NET Core 数据保护概述。

- [数据保护密钥管理和生存期](#)

数据保护密钥管理和生存期的相关信息。

- [数据保护的计算机范围策略支持](#)

为所有使用数据保护的应用设置默认计算机范围策略的详细信息。

- [ASP.NET Core 中数据保护的非 DI 感知情境](#)

如何通过 `DataProtectionProvider` 具体类型，在不经 DI 特定代码流程的情况下使用数据保护。

配置 ASP.NET 核心数据保护

2018/4/27 • 9 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

初始化数据保护系统时, 它将应用默认设置基于的操作环境。这些设置并在一台计算机上运行的应用程序通常适用。有情况下, 开发人员可能需要更改默认设置:

- 应用程序分布在多台计算机。
- 出于合规性原因。

对于这些情况下, 数据保护系统提供丰富的配置 API。

警告

类似于配置文件, 数据保护密钥环应保护使用适当的权限。你可以选择加密静止的密钥, 但这不能防止攻击者创建新密钥。因此, 应用的安全会受到影响。使用数据保护配置的存储位置应具有其访问仅限于应用程序本身, 你将保护配置文件工作方式相似。例如, 如果你选择存储在磁盘上的密钥令牌环, 使用文件系统权限。确保仅在标识你的 web 应用运行具有读取、写入和创建该目录的访问。如果你使用 Azure 表存储, 仅该 web 应用能够读取、写入或在表存储等中创建新条目。

扩展方法 `AddDataProtection` 返回 `IDataProtectionBuilder`。`IDataProtectionBuilder` 显示扩展方法, 你可以链接在一起以配置数据保护选项。

PersistKeysToFileSystem

将密钥存储在 UNC 共享而不是在 `%LOCALAPPDATA%` 默认位置, 配置与系统 `PersistKeysToFileSystem`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\""));
}
```

警告

如果更改密钥持久性位置, 系统将不再自动加密组合键, 其余部分, 因为它不知道 DPAPI 是否合适的加密机制。

ProtectKeysWith*

你可以将系统配置为通过调用的任何保护静止的密钥 `ProtectKeysWith*` 配置 API。请考虑以下示例中, 它将密钥存储的 UNC 共享上并对这些密钥在使用特定的 X.509 证书的其余部分进行加密:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\""))
        .ProtectKeysWithCertificate("thumbprint");
}
```

请参阅 [将加密密钥在 Rest](#) 有关更多示例和讨论的内置密钥加密机制。

SetDefaultKeyLifetime

若要配置系统而不是默认值 90 天使用的密钥生存期为 14 天，使用[SetDefaultKeyLifetime](#):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
}
```

SetApplicationName

默认情况下，数据保护系统隔离应用程序从另一个，即使它们共享相同的物理密钥存储库。这可以阻止应用了解对方的受保护的负载。若要共享两个应用程序之间的受保护的负载，使用[SetApplicationName](#)与每个应用程序相同的值：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetApplicationName("shared app name");
}
```

DisableAutomaticKeyGeneration

您可能有不希望自动轮转（创建新的密钥）的密钥，因为它们接近过期的应用的方案。这一个示例可能是应用程序设置在主要/辅助关系中，其中仅主应用程序负责密钥管理问题，而辅助应用只需密钥环的只读视图。可以配置辅助应用程序通过配置与系统以只读方式处理密钥环[DisableAutomaticKeyGeneration](#):

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

每个应用程序隔离

数据保护系统提供了 ASP.NET 核心主机，但它自动隔离了从另一个，应用程序，即使这些应用在相同的工作进程帐户下运行，并且使用相同的主密钥材料。这是某种程度上类似于 `IsolateApps` 修饰符从 `System.Web` 的 `<machineKey>` 元素。

隔离机制的工作原理是作为唯一的租户，因此考虑本地计算机上的每个应用[IDataProtector](#)取得 root 权限的任何给定的应用会自动包括为鉴别器的应用程序 ID。应用程序的唯一 ID 来自两个位置之一：

1. 如果在 IIS 中托管应用的唯一标识符是应用程序的配置路径。如果在 web 场环境中部署应用后，此值应为稳定，前提是在 web 场中的所有计算机同样配置 IIS 环境。
2. 如果应用程序不在 IIS 中承载的唯一标识符是应用程序的物理路径。

唯一标识符旨在得以重置—同时的各个应用程序和计算机本身。

此隔离机制假设应用不可恶意。恶意应用程序始终可以影响在相同的工作进程帐户下运行的任何其他应用程序。在共享宿主环境中应用是相互不受信任，托管提供商应采取措施来确保应用，包括分离应用的基础密钥的存储库之间的操作系统级别隔离。

如果由 ASP.NET 核心主机未提供数据保护系统（例如，如果通过其实例化 `DataProtectionProvider` 具体类型）应

用程序隔离在默认情况下处于禁用状态。当禁用应用程序隔离时，相同的密钥材料作为后盾的所有应用可以都共享的负载，只要它们提供相应[目的](#)。若要提供在此环境中的应用程序隔离，调用[SetApplicationName](#)方法的配置对象，并提供每个应用程序的唯一名称。

更改与 UseCryptographicAlgorithms 算法

数据保护堆栈可以更改默认的算法使用的新生成的键。执行此操作的最简单方法是调用[UseCryptographicAlgorithms](#)从配置回调：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptorConfiguration()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
}
```

默认值 `EncryptionAlgorithm AES 256 CBC`，且默认 `ValidationAlgorithm HMACSHA256`。默认策略可以设置由系统管理员通过[计算机范围策略](#)，但显式调用 `UseCryptographicAlgorithms` 将替代默认策略。

调用 `UseCryptographicAlgorithms` 允许你指定从预定义的内置列表所需的算法。你不必担心算法的实现。在上述方案中，数据保护系统会尝试在 Windows 上运行使用 AES 的 CNG 实现。否则，则会返回到托管 `System.Security.Cryptography.Aes` 类。

你可以手动指定通过调用实现[UseCustomCryptographicAlgorithms](#)。

提示

更改算法不会影响现有的密钥在密钥环。它只影响新生成的键。

指定自定义托管的算法

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要指定自定义托管的算法，创建[ManagedAuthenticatedEncryptorConfiguration](#)指向的实现类型的实例：

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptorConfiguration()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
}
```

通常*类型属性必须指向具体，可实例化（通过公共的无参数 ctor）实现的[SymmetricAlgorithm](#)和[KeyedHashAlgorithm](#)，不过系统特殊的情况下等某些值 `typeof(Aes)` 为方便起见。

注意

SymmetricAlgorithm 必须 \geq 128 位的密钥长度和块大小的 \geq 64 位, 并且它必须支持使用 PKCS #7 填充 CBC 模式下的加密。KeyedHashAlgorithm 的摘要大小必须 $>=$ 128 位, 并且它必须支持密钥长度等于 length 哈希算法的摘要。KeyedHashAlgorithm 并非是严格要求, 要 HMAC。

指定自定义 Windows CNG 算法

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要指定自定义 Windows CNG 算法通过 HMAC 验证使用 CBC 模式下加密, 创建 [CngCbcAuthenticatedEncryptorConfiguration](#) 实例, 其中包含的算法的信息:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
}
```

注意

对称块加密算法的密钥长度必须 $>=$ 128 位的块大小 $>=$ 64 位, 并且它必须支持使用 PKCS #7 填充 CBC 模式下的加密。哈希算法的摘要大小必须 $>=$ 128 位并且必须支持与 BCRYPT 打开_ALG_处理_HMAC_标志标志。*提供程序属性可以设置为空, 默认的提供程序用于为指定的算法。请参阅 [BCryptOpenAlgorithmProvider](#) 文档以了解更多信息。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

若要指定自定义 Windows CNG 算法通过验证使用 Galois/计数器模式加密, 创建 [CngGcmAuthenticatedEncryptorConfiguration](#) 实例, 其中包含的算法的信息:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
}
```

注意

对称块加密算法的密钥长度必须 \geq 128 位的块大小为完全 128 位，并且它必须支持 GCM 加密。你可以设置 [EncryptionAlgorithmProvider](#) 属性为 null，以便使用默认提供程序指定的算法。请参阅 [BCryptOpenAlgorithmProvider](#) 文档以了解更多信息。

指定其他自定义算法

尽管不作为第一类 API 公开，则数据保护系统是算法的可扩展，足以允许指定几乎任何类型。例如，很可能需要包含在硬件安全模块 (HSM) 的所有键，并为核心的自定义实现加密和解密的例程。请参阅 [IAuthenticatedEncryptor](#) 中 [核心加密可扩展性](#) 有关详细信息。

保留密钥：当在 Docker 容器中承载

在中承载时 Docker 容器，应在维护密钥：

- 是一个 Docker 卷容器的生存期结束，例如共享的卷或者主机已装入卷后仍然存在，一个文件夹。
- 外部提供程序，如 [Azure 密钥保管库](#) 或 [Redis](#)。

请参阅

- [非 DI 感知方案](#)
- [计算机范围的策略](#)

数据保护密钥管理和 ASP.NET Core 的生存时间

2018/1/30 • 3 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

密钥管理

应用程序将尝试检测其操作环境并处理其自己的密钥配置。

- 如果应用程序承载于[Azure Apps](#), 密钥保存到*%HOME%\ASP.NET\DataProtection-Keys*文件夹。此文件夹由网络存储提供支持, 并且在托管应用的所有计算机同步。
 - 密钥未保护静止。
 - 数据保护密钥文件夹提供密钥环在单个部署槽中的应用程序的所有实例。
 - 单独的部署槽, 如过渡和生产环境, 请不要共享密钥链。更换之间部署槽, 如交换过渡到生产环境或使用 A / B 测试, 使用数据保护任何应用将无法解密使用密钥链内前一槽的存储的数据。这会导致对正在注销的应用程序使用标准的 ASP.NET Core cookie 身份验证, 因为它使用数据保护来保护其 cookie 的用户。如果你需要独立于槽的密钥环, 使用 Azure Blob 存储, Azure 密钥保管库, SQL 存储区中, 之类的外部密钥环提供程序, 或 Redis 缓存。
- 如果可用的用户配置文件, 将密钥保存到*%LOCALAPPDATA%\ASP.NET\DataProtection-Keys*文件夹。如果操作系统是 Windows, 密钥被加密使用 DPAPI 对静止。
- 如果应用程序托管在 IIS 中, 密钥会保留到 HKLM 注册表仅对辅助进程帐户是 ACLed 某个特殊的注册表项中。使用 DPAPI 对密钥静态加密。
- 如果没有这些条件匹配, 密钥不保留在当前进程之外。进程关闭时, 生成所有密钥都将丢失。

开发人员可始终完全控制, 并且可重写如何和密钥的存储位置。上面的前三个选项应提供很好的默认值对于大多数应用程序类似于如何 ASP.NET **<machineKey>**过去可以正常运行自动生成例程。最终的回退选项是要求开发人员指定的唯一情形[配置](#)前部如果他们想要密钥持久性, 但此回退仅发生在极少数情况下。

当承载 Docker 容器中时, 应是(共享的卷或容器的生存期结束后仍然存在的主机装入的卷)的 Docker 卷的文件夹中保留密钥或在外部提供程序, 如[Azure 密钥保管库](#)或[Redis](#)。如果应用程序不能访问共享的网络卷, 可能也是在 web 场方案中有用外部提供程序(请参阅[PersistKeysToFileSystem](#)有关详细信息)。

警告

如果开发人员重写上面所述的规则和点的特定的密钥存储库的数据保护系统, 则会禁用自动加密对静止的密钥。在 rest 保护可能会通过重新启用[配置](#)。

密钥的生存期

默认情况下, 密钥具有 90 天的生存期。密钥过期时, 该应用将自动生成新密钥, 并将新的密钥设置为活动密钥。只要已停用的密钥保留在系统上, 你的应用程序可以解密受保护的与之任何数据。请参阅[密钥管理](#)有关详细信息。

默认的算法

使用的默认负载保护算法是 AES 256 CBC 机密性和 HMACSHA256 真实性。512 位主密钥, 每隔 90 天更改一次用于派生用于每个负载基于这些算法的两个子键。请参阅[子项派生](#)有关详细信息。

请参阅

- [密钥管理扩展性](#)

在 ASP.NET 核心中支持的数据保护计算机范围的策略

2018/4/10 • 4 min to read • [Edit Online](#)

作者: Rick Anderson

Windows 上运行时, 数据保护系统具有有限的支持设置默认计算机范围的所有应用程序使用 ASP.NET 核心数据保护策略。常规的思路是管理员可能想要更改默认设置, 如算法或密钥的生存期, 而无需手动更新每个应用程序, 在计算机上。

警告

系统管理员可以设置默认策略, 但它们无法强制执行它。应用程序开发人员始终可以重写其中一个自己的选择的任何值。默认策略仅影响应用程序开发人员未指定显式设置值的位置。

设置默认策略

若要设置默认策略, 管理员可以设置以下注册表项下系统注册表中的已知的值:

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

如果你是在 64 位操作系统上, 并想要影响的 32 位应用程序的行为, 请记住将上面的项的 Wow6432Node 等效项。

支持的值如下所示。

值	类型	描述
EncryptionType	字符串	指定应使用哪些算法来进行数据保护。值必须为 CNG CBC、CNG 的 GCM 或托管和中更详细地介绍了。
DefaultKeyLifetime	DWORD	指定新生成的键的生存期。此值以天为单位指定, 并且必须是 $> = 7$ 。
KeyEscrowSinks	字符串	指定用于密钥证书的类型。值是以分号分隔的密钥托管接收器, 其中在列表中的每个元素都实现的类型的程序集限定名称列表 IKeyEscrowSink 。

加密类型

如果 EncryptionType CNG CBC, 系统配置为使用 CBC 模式对称的块密码来进行保密性和 HMAC 真实性有通过 Windows CNG 提供服务 (请参阅[指定自定义 Windows CNG 算法](#)为更多详细信息)。支持以下的其他值, 其中每个对应于 CngCbcAuthenticatedEncryptionSettings 类型上的属性。

值	类型	描述
EncryptionAlgorithm	字符串	理解的 CNG 对称的块密码算法的名称。此算法在 CBC 模式下打开。

值	类型	描述
EncryptionAlgorithmProvider	字符串	可以生成 EncryptionAlgorithm 的算法的 CNG 提供程序实现的名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称块加密算法的密钥长度(以位为单位)。
HashAlgorithm	字符串	理解的 CNG 哈希算法的名称。此算法在 HMAC 模式打开。
HashAlgorithmProvider	字符串	可以生成算法的 HashAlgorithm CNG 提供程序实现的名称。

如果 EncryptionType CNG GCM, 系统配置为使用保密性和身份验证与服务提供的 Windows CNG Galois/计数器模式对称的块密码(请参阅[指定自定义 Windows CNG 算法](#)有关详细信息)。支持以下的其他值, 其中每个对应于 CngGcmAuthenticatedEncryptionSettings 类型上的属性。

值	类型	描述
EncryptionAlgorithm	字符串	理解的 CNG 对称的块密码算法的名称。此算法将在 Galois/计数器模式中打开。
EncryptionAlgorithmProvider	字符串	可以生成 EncryptionAlgorithm 的算法的 CNG 提供程序实现的名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称块加密算法的密钥长度(以位为单位)。

如果管理 EncryptionType, 系统配置为用于托管的 SymmetricAlgorithm 保密性和 KeyedHashAlgorithm 真实性(请参阅[指定自定义管理算法](#)有关详细信息)。支持以下的其他值, 其中每个对应于 ManagedAuthenticatedEncryptionSettings 类型上的属性。

值	类型	描述
EncryptionAlgorithmType	字符串	实现 SymmetricAlgorithm 的类型的程序集限定名称。
EncryptionAlgorithmKeySize	DWORD	要派生的对称加密算法的密钥长度(以位为单位)。
ValidationAlgorithmType	字符串	实现 KeyedHashAlgorithm 的类型的程序集限定名称。

如果 EncryptionType 具有任何其他值非 null 或空时, 数据保护系统启动时引发的异常。

警告

在配置涉及类型名称(EncryptionAlgorithmType、ValidationAlgorithmType、KeyEscrowSinks)的默认策略设置时, 类型必须是可用于应用。这意味着, 对于在桌面 CLR 上运行的应用程序, 包含这些类型的程序集应会显示在全局程序集缓存(GAC)中。对于在.NET Core 上运行的 ASP.NET Core 应用程序, 应安装包含这些类型的包。

非 DI 感知的情境中 ASP.NET Core 的数据保护

2018/3/3 • 2 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

ASP.NET 核心数据保护系统通常是[添加到服务容器](#)并且供通过依赖关系注入 (DI) 的从属组件。但是, 一些情况下, 这不可行或所需, 尤其是在将系统导入到现有应用程序。

若要支持这些方案中, [Microsoft.AspNetCore.DataProtection.Extensions](#)包提供具体类型, [DataProtectionProvider](#), 它提供一种简单的方法为使用数据保护而不依赖于 DI。[DataProtectionProvider](#) 类型实现[IDataProtectionProvider](#)。构造[DataProtectionProvider](#) 只需提供[DirectoryInfo](#)实例, 以指示应存储提供程序的加密密钥的位置, 如下面的代码示例中所示:

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8FwBAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
*/
```

默认情况下, [DataProtectionProvider](#) 具体的类型不加密原始密钥材料之前将其保存到文件系统。这是为了支持开

发人员将指向网络共享和数据保护系统无法自动推导适当静态密钥加密机制的方案。

此外，`DataProtectionProvider` 具体的类型不隔离应用程序默认情况下。使用相同密钥的目录的所有应用程序可以共享负载长达其目的参数匹配。

`DataProtectionProvider`构造函数接受一个可选配置回调，可以用于调整的系统行为。下面的示例演示与显式调用还原隔离`SetApplicationName`。此示例还演示将系统配置为自动加密持久化的密钥使用 Windows DPAPI。如果目录指向 UNC 共享，可能想要在所有相关计算机间分发共享的证书还可将系统配置为使用基于证书的加密和调用`ProtectKeysWithCertificate`。

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}
```

提示

实例 `DataProtectionProvider` 具体类型是创建开销很大。如果应用程序维护此类型的多个实例，并且如果他们正在使用相同的密钥存储目录，应用程序性能可能会降低。如果你使用 `DataProtectionProvider` 类型，我们建议你一次创建此类型，并重复使用尽可能多地它。`DataProtectionProvider` 类型及其所有 `IDataProtector` 从它创建的实例是线程安全的多个调用方。

ASP.NET Core 数据保护扩展性 API

2018/4/10 • 1 min to read • [Edit Online](#)

- [核心加密扩展性](#)
- [密钥管理扩展性](#)
- [其他 API](#)

在 ASP.NET 核心中的核心加密可扩展性

2018/4/10 • 7 min to read • [Edit Online](#)

警告

实现的任意以下接口的类型应该是线程安全的多个调用方。

IAuthenticatedEncryptor

IAuthenticatedEncryptor 接口是加密子系统的基本构建基块。通常没有一个 IAuthenticatedEncryptor 每个键，并且 IAuthenticatedEncryptor 实例包装所有加密的密钥材料和算法执行加密操作所需的信息。

顾名思义，类型是负责提供经过身份验证的加密和解密服务。以下两个 API，它公开。

- 解密 (ArraySegment 已加密文本, ArraySegment additionalAuthenticatedData): byte]
- 加密 (ArraySegment 纯文本, ArraySegment additionalAuthenticatedData): byte]

加密方法返回的 blob，包括 enciphered 纯文本和身份验证标记。身份验证标记必须包含附加经过身份验证的数据 (AAD) 时，尽管 AAD 本身不需要从最终的负载。解密方法验证身份验证标记，并返回被解码的负载。应为 CryptographicException homogenized (除 ArgumentNullException 和类似) 的所有失败。

注意

实际上，IAuthenticatedEncryptor 实例本身不需要包含密钥材料。例如，实现可以将委托到 HSM 中的所有操作。

如何创建 IAuthenticatedEncryptor

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

IAuthenticatedEncryptorFactory 接口表示一种类型，知道如何创建 IAuthenticatedEncryptor 实例。其 API 是，如下所示。

- CreateEncryptorInstance (IKey 密钥) : IAuthenticatedEncryptor

对于任何给定的 IKey 实例，其 CreateEncryptorInstance 方法创建的任何经过身份验证的加密程序应被视为等效，如下面的代码示例。

```

// we have an IAuthenticatedEncryptorFactory instance and an IKey instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);

// the 'roundTripped' and 'plaintext' buffers should be equivalent

```

IAuthenticatedEncryptorDescriptor (ASP.NET 核心 2.x 仅)

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

IAuthenticatedEncryptorDescriptor 接口表示知道如何将自身导出到 XML 的类型。其 API 是，如下所示。

- `ExportToXml() : XmlSerializedDescriptorInfo`

XML 序列化

`IAuthenticatedEncryptor` 和 `IAuthenticatedEncryptorDescriptor` 的主要区别是描述符知道如何创建加密程序和其提供有效的参数。请考虑其实现依赖于 `SymmetricAlgorithm` 和 `KeyedHashAlgorithm` `IAuthenticatedEncryptor`。加密器的作业是使用这些类型，但它不一定知道这些类型的来源，因此它不能真正写出如何重新创建本身应用程序重启时的正确说明。描述符充当基于此较高级别。由于描述符知道如何创建加密程序实例（例如，它知道如何创建所需的算法），以便可以重新创建加密器实例，应用程序重置后，它可以序列化该知识库中的 XML 格式。

通过其 `ExportToXml` 例程，描述符可序列化。此例程返回 `XmlSerializedDescriptorInfo` 其中包含两个属性：`XElement` 表示形式的说明符和表示的类型 `IAuthenticatedEncryptorDescriptorDeserializer` 可以是用于使重新起用给定相应 `XElement` 此描述符。

序列化的描述符可能包含敏感信息，例如加密的密钥材料。数据保护系统具有已持久化到存储空间之前加密信息的内置支持。若要利用此功能，该描述符应标记包含敏感信息的属性名称“`requiresEncryption`”的元素 (`xmlns="http://schemas.asp.net/2015/03/dataProtection"`)，值“`true`”。

提示

没有用于将此属性设置的帮助器 API。调用 `XElement.MarkAsRequiresEncryption()` 位于命名空间 `Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel` 扩展方法。

也可以是其中的序列化的描述符不包含敏感信息的情况。再次考虑存储在 HSM 中的加密密钥的大小写。在序列化本身由于 HSM 就不会暴露纯文本形式材料时可描述符无法写出密钥材料。相反，描述符可以编写出密钥包装的密钥（如果 HSM 允许以这种方式导出）或密钥的 HSM 自己唯一标识符版本。

IAuthenticatedEncryptorDescriptorDeserializer

IAuthenticatedEncryptorDescriptorDeserializer 接口表示知道如何反序列化从 `XElement` `IAuthenticatedEncryptorDescriptor` 实例的类型。它公开一种方法：

- `ImportFromXml (XElement 元素) : IAuthenticatedEncryptorDescriptor`

`ImportFromXml` 方法采用返回 XElement `IAuthenticatedEncryptorDescriptor.ExportToXml` 和创建原始 `IAuthenticatedEncryptorDescriptor` 等效项。

实现 `IAuthenticatedEncryptorDescriptorDeserializer` 该类型应具有以下两个公共构造函数之一：

- `.ctor(IServiceProvider)`
- `.ctor()`

注意

`IServiceProvider` 传递到构造函数可能为 null。

顶级工厂

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

AlgorithmConfiguration 类表示知道如何创建该类型 `IAuthenticatedEncryptorDescriptor` 实例。它公开一个单一的 API。

- `CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor`

将 `AlgorithmConfiguration` 视为顶级工厂。配置用作模板。它所包装算法信息（例如，此配置生成使用 AES 128 GCM 主密钥描述符），但它尚不与特定键关联。

当调用 `CreateNewDescriptor`、仅用于此调用中，创建新的密钥材料，并且生成新 `IAuthenticatedEncryptorDescriptor` 其包装此密钥材料以及所需使用材料算法信息。无法在软件中创建（和保存在内存中）的密钥材料，它无法创建并保存在 HSM 中，依次类推。关键点是 `CreateNewDescriptor` 任何两个调用应永远不会创建等效 `IAuthenticatedEncryptorDescriptor` 实例。

`AlgorithmConfiguration` 类型用作密钥创建例程的入口点如[自动密钥滚动](#)。若要更改的所有将来的键的实现，请在 `KeyManagementOptions` 中设置 `AuthenticatedEncryptorConfiguration` 属性。

ASP.NET 核心的密钥管理可扩展性

2018/4/10 • 9 min to read • [Edit Online](#)

提示

读取[密钥管理](#)之前阅读此部分中，因为它介绍了这些 API 的基本概念的某些部分。

警告

实现的任意以下接口的类型应该是线程安全的多个调用方。

键

`IKey` 接口是加密系统中的键的基本表示。在抽象的意义上，不在“加密密钥材料”字面意义上此处使用的术语密钥。密钥具有以下属性：

- 激活、创建和到期日期
- 吊销状态
- 密钥标识符 (GUID)
- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

此外，`IKey` 公开 `CreateEncryptor` 方法用于创建 `IAuthenticatedEncryptor` 实例绑定到此密钥。

注意

没有 API 检索从原始的加密材料 `IKey` 实例。

IKeyManager

`IKeyManager` 接口表示负责常规的密钥存储、检索和操作的对象。它公开三个高级操作：

- 创建新密钥并将其保存在存储。
- 从存储中获取所有键。
- 撤消一个或多个密钥，而且将保留到存储的吊销信息。

警告

编写 `IKeyManager` 是一个非常高级的任务，和大多数开发人员不应尝试。相反，大多数开发人员应充分利用所提供的功能 `XmlKeyManager` 类。

XmlKeyManager

`XmlKeyManager` 类型是内置的具体实现 `IKeyManager`。它提供了几个有用的功能，包括密钥证书和加密对静止的密

键。在此系统的密钥将呈现为 XML 元素 (具体而言, `XElement`)。

`XmlKeyManager` 依赖于在完成其任务的过程中的其他几个组件:

- `ASP.NET Core 2.x`
- `ASP.NET Core 1.x`

• `AlgorithmConfiguration` 这决定使用新密钥的算法。

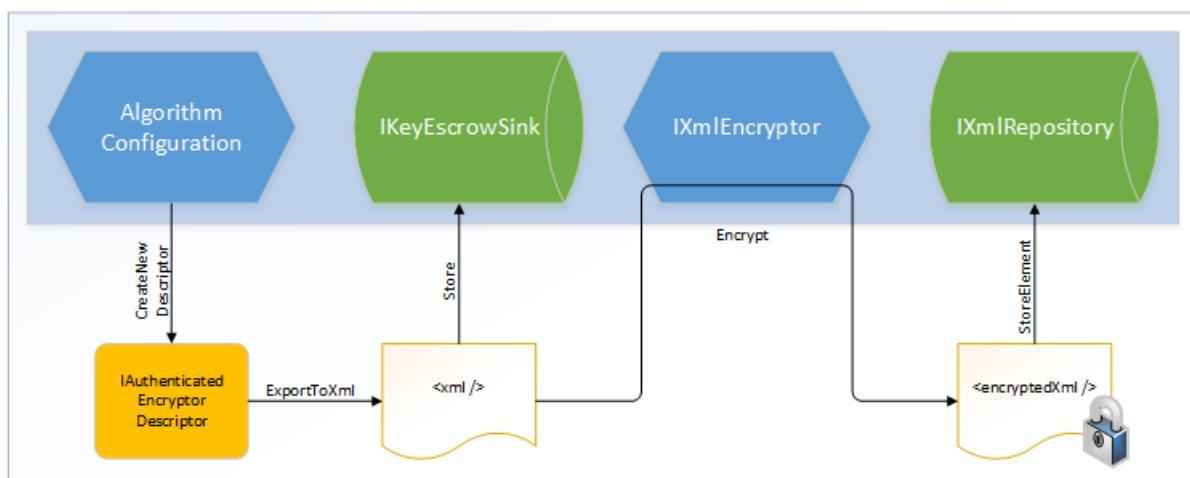
• `IKeyEscrowSink` 其中键将保留在存储中的控件。

• `IXmlEncryptor` [可选], 这允许加密对静止的密钥。

• `IKeyEscrowSink` [可选], 其提供密钥托管服务。

以下是高级关系图, 表明如何这些组件连接在一起内 `XmlKeyManager`。

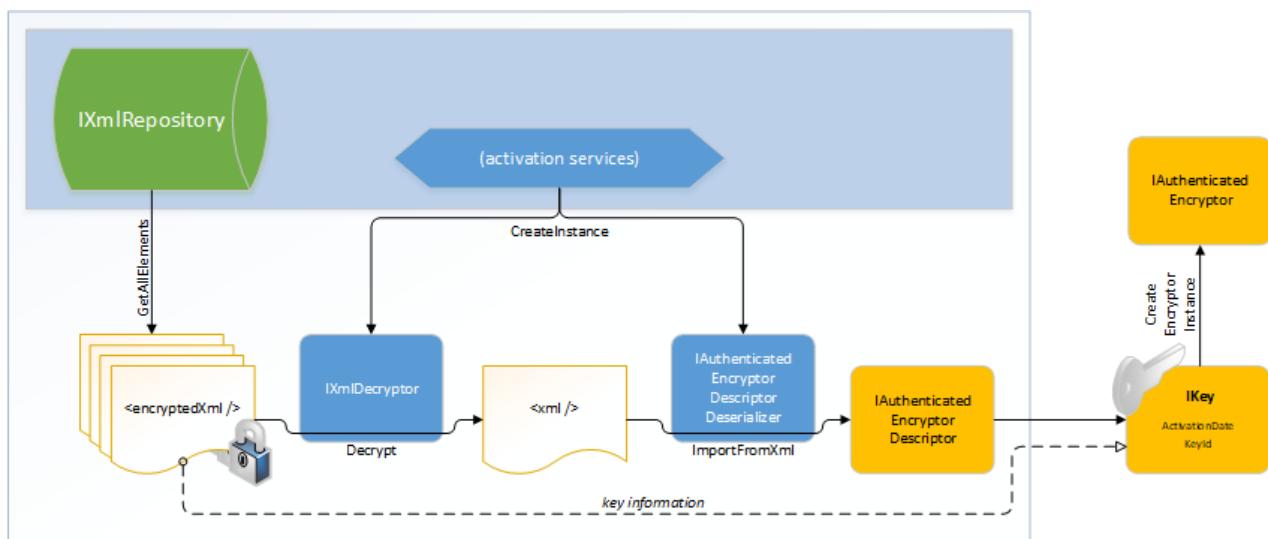
- `ASP.NET Core 2.x`
- `ASP.NET Core 1.x`



密钥创建 / `CreateNewKey`

实现中 `CreateNewKey`、`AlgorithmConfiguration` 组件用于创建唯一 `IAuthenticatedEncryptorDescriptor`, 其中然后序列化为 XML。如果存在密钥托管接收器, 则原始 (未加密) 的 XML 到接收器提供适用于长期存储。然后通过运行的未加密的 XML `IXmlEncryptor` (如果需要) 来生成加密的 XML 文档。此加密的文档保存到长期存储通过 `IXmlRepository`。(如果没有 `IXmlEncryptor` 是配置, 未加密的文档保存在 `IXmlRepository`。)

- `ASP.NET Core 2.x`
- `ASP.NET Core 1.x`



密钥检索 / GetAllKeys

实现中 `GetAllKeys`、XML 文档表示键和从基础读取吊销 `IRepository`。如果这些文档进行加密，系统将自动解密。`XmlKeyManager` 创建适当 `IAuthenticatedEncryptorDescriptorDeserializer` 实例进行反序列化文档回 `IAuthenticatedEncryptorDescriptor` 实例，然后将封装在单个 `IKey` 实例。此集合 `IKey` 实例返回给调用方。

在找不到在特定的 XML 元素上的进一步信息[密钥存储格式文档](#)。

IRepository

`IRepository` 接口表示可保留 XML 到和从一个后备存储检索 XML 的类型。它公开两个 API:

- `GetAllElements(): IReadOnlyCollection<Element>`
- `StoreElement(XElement element, string friendlyName)`

实现 `IRepository` 不需要通过其传递对 XML 进行分析。它们应视为不透明 XML 文档，让较高层担心生成和分析文档。

有两个内置的具体类型实现 `IRepository : FileSystemXmlRepository` 和 `RegistryXmlRepository`。请参阅[密钥存储提供程序文档](#)有关详细信息。注册的自定义 `IRepository` 将适当的方式为使用不同的后备存储，例如，Azure Blob 存储。

若要更改默认存储库应用程序级，注册一个自定义 `IRepository` 实例：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.Configure<KeyManagementOptions>(options => options.XmlRepository = new MyCustomXmlRepository());
```

IXmlEncryptor

`IXmlEncryptor` 接口表示可以加密纯文本 XML 元素的类型。它公开单个 API:

- `Encrypt(XElement plaintextElement): EncryptedXmlInfo`

如果一个序列化 `IAuthenticatedEncryptorDescriptor` 包含任何元素标记为“需要加密”，然后 `XmlKeyManager` 将通过配置运行这些元素 `IXmlEncryptor` 的 `Encrypt` 方法，和它将保留 `enciphered` 的元素而不是为纯文本元素 `IXmlRepository`。输出 `Encrypt` 方法是 `EncryptedXmlInfo` 对象。此对象是包装器，其中包含两个所产生的 `enciphered` `XElement` 和表示类型 `IXmlDecryptor` 可用于解密的相应元素。

有四个内置的具体类型实现 `IXmlEncryptor`：

- `CertificateXmlEncryptor`
- `DpapiNGXmlEncryptor`
- `DpapiXmlEncryptor`
- `NullXmlEncryptor`

请参阅[rest 文档的密钥加密](#)有关详细信息。

若要更改默认密钥加密在 rest 机制整个应用程序范围，注册一个自定义 `IXmlEncryptor` 实例：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.Configure<KeyManagementOptions>(options => options.XmlEncryptor = new MyCustomXmlEncryptor());
```

IXmlDecryptor

`IXmlDecryptor` 接口表示一种类型，知道如何解密 `XElement`，已通过 enciphered `IXmlEncryptor`。它公开单个 API：

- 解密 (XElement encryptedElement): XElement

`Decrypt` 方法撤消由执行加密 `IXmlEncryptor.Encrypt`。通常情况下，每个具体 `IXmlEncryptor` 实现将具有相应的具体 `IXmlDecryptor` 实现。

类型可实现 `IXmlDecryptor` 应该具有以下两个公共构造函数之一：

- .ctor(IServiceProvider)
- .ctor()

注意

`IServiceProvider` 传递给构造函数可能为 null。

IKeyEscrowSink

`IKeyEscrowSink` 接口表示可以执行托管的敏感信息的类型。回想一下，序列化的描述符可能包含敏感信息（如加密材料），这是什么导致了引入 `IXmlEncryptor` 键入第一个位置。但是，意外的发生，并且密钥环可以删除或损坏。

托管接口提供了允许访问原始序列化的 XML 转换由任何配置前紧急转义阴影 `IXmlEncryptor`。接口公开单个 API：

- 应用商店 (Guid keyId、 XElement 元素)

这就需要通过 `IKeyEscrowSink` 实现，以安全的方式与业务策略一致处理提供的元素。一个可能的实现可能是为托管接收器使用已知的企业 X.509 证书的 XML 元素进行加密其中已托管证书的私钥；`CertificateXmlEncryptor` 类型可以对此有帮助。`IKeyEscrowSink` 实现程序还负责适当地保留提供的元素。

默认情况下没有托管机制启用，但服务器管理员可以[全局配置此](#)。它还可以配置以编程方式通过

`IDataProtectionBuilder.AddKeyEscrowSink` 方法，如下面的示例中所示。`AddKeyEscrowSink` 方法重载镜像 `IServiceCollection.AddSingleton` 和 `IServiceCollection.AddInstance` 重载，为 `IKeyEscrowSink` 实例都应是单一实例。如果选择多个 `IKeyEscrowSink` 注册实例，每个过程中将调用密钥生成，因此键可以同时托管多个机制。

没有任何 API 来读取中的材料 `IKeyEscrowSink` 实例。这是与托管机制的设计理论一致：它具有用于受信任的颁发机构，方便密钥材料，并且应用程序本身不是受信任颁发机构，因为它不应具有访问自己保管材料。

下面的示例代码演示如何创建和注册 `IKeyEscrowSink` 其中托管密钥，以便只有“CONTOSO Domain 管理员”的成员可以恢复它们。

注意

若要运行此示例，你必须是在已加入域的 Windows 8 / Windows Server 2012 计算机和域控制器必须是 Windows Server 2012 或更高版本。

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
```

```

using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                services);
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIbTCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */

```


杂项 ASP.NET 核心数据保护 API

2018/4/10 • 1 min to read • [Edit Online](#)

警告

实现的任意以下接口的类型应该是线程安全的多个调用方。

ISecret

`ISecret` 接口表示一个机密的值，如加密的密钥材料。它包含以下 API 图面：

- `Length` : `int`
- `Dispose()` : `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)` : `void`

`WriteSecretIntoBuffer` 方法填充原始机密值与提供的缓冲区。此 API 将作为参数的缓冲区的原因而不返回 `byte[]` 直接是，这使调用方有机会固定限制对托管的垃圾回收器的机密暴露该缓冲区对象。

`Secret` 类型是的具体实现 `ISecret` 机密的值在进程中内存中的存储位置。在 Windows 平台上的机密的值加密通过 [CryptProtectMemory](#)。

ASP.NET Core 数据保护实现

2018/4/10 • 1 min to read • [Edit Online](#)

- [已验证的加密详细信息](#)
- [子项派生和已验证的加密](#)
- [上下文标头](#)
- [密钥管理](#)
- [密钥存储提供程序](#)
- [静态密钥加密](#)
- [密钥永久性和设置](#)
- [密钥存储格式](#)
- [短数据保护提供程序](#)

经过身份验证的加密在 ASP.NET Core 的详细信息

2018/4/10 • 2 min to read • [Edit Online](#)

对 `IDataProtector.Protect` 调用都是经过身份验证的加密操作。保护方法提供保密性和真实性，并且它会绑定到用于此特定 `IDataProtector` 实例派生其根 `IDataProtectionProvider` 目的链。

`IDataProtector.Protect` 采用 `byte[]` 纯文本参数，并且生成的字节是受保护的负载，其格式为下面所述。（此外还有一个扩展方法重载以采用字符串的纯文本参数和返回的字符串受保护的负载。如果使用此 API 的受保护的负载格式仍将下面结构，但它会[base64url 编码](#)。）

受保护的负载格式

受保护的负载格式包含三个主要组件：

- 32 位神奇标头，它标识数据保护系统的版本。
- 128 位密钥 id 标识该密钥用于保护此特定的负载。
- 受保护的负载的剩余部分[特定于加密程序按此键封装](#)。在下面的示例密钥表示 AES 256 CBC + HMACSHA256 加密器和负载进一步细分，如下所示：* 一个 128 位密钥修饰符。* 一个 128 位初始化向量。* AES 256 CBC 输出 48 个字节。* 一个 HMACSHA256 身份验证标记。

示例受保护的负载如下所示。

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

从上面的第一个 32 位或 4 个字节的负载格式是标识版本 (09 F0 C9 F0) 的幻标头

下一步的 128 位或 16 个字节是密钥标识符 (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

其余部分包含负载，并特定于所使用的格式。

警告

给定的键对受保护的所有负载将都开始与同一个 20 字节（神奇的值，密钥 id）标头。管理员可以使用这一事实以进行诊断来估计生成负载时间。例如，上面的负载对应于键 {0c819c80-6619-4019-9536-53f8aaffee57}。如果检查密钥的存储库后找到验证此特定密钥激活日期为 2015 年-01-01 和其到期日期为 2015 年-03-01，然后是合乎情理假定负载（如果未被篡改）在该窗口中，给予生成或采用一个较小在任意一侧奶油因子。

子项派生和 ASP.NET Core 中的经过身份验证的加密

2018/4/10 • 5 min to read • [Edit Online](#)

密钥链中的大多数键将包含某种形式的平均信息量，并且将有算法信息指出“CBC 模式下加密 + HMAC 验证”或“GCM 加密 + 验证”。在这些情况下，我们将嵌入的平均信息量称为此密钥的主密钥材料（或公里）和我们执行派生将使用为实际的加密操作的密钥的密钥派生函数。

注意

键是抽象的并自定义实现可能不会按如下所示的那样。如果密钥提供其自己的实现 `IAuthenticatedEncryptor` 而不是使用我们的内置工厂之一，本节中所述的机制不再适用。

其他经过身份验证的数据和子项派生

`IAuthenticatedEncryptor` 接口用作所有经过身份验证的加密操作的核心接口。其 `Encrypt` 方法采用两个缓冲区：纯文本和 `additionalAuthenticatedData` (AAD)。纯文本内容流保持不变调用 `IDataProtector.Protect`，但 AAD 由系统生成和包括三个组件：

1. 32 位神奇标头 09 F0 C9 F0 标识此版本的数据保护系统。
2. 128 位密钥 id。
3. 从创建目的链形成的可变长度字符串 `IDataProtector`，正在执行此操作。

由于 AAD 是为所有三个组件的元组唯一的我们可以使用它从密钥主机派生新密钥，而不是使用密钥主机本身中所有的我们的加密操作。每次调用 `IAuthenticatedEncryptor.Encrypt`，发生以下密钥派生过程：

$(K_E, K_H) = SP800_108_CTR_HMACSHA512(K_M, AAD, contextHeader \parallel keyModifier)$

在这里，我们正在呼叫 NIST SP800 108 KDF 中计数器模式（请参阅 [NIST SP800-108](#)，sec。5.1）使用以下参数：

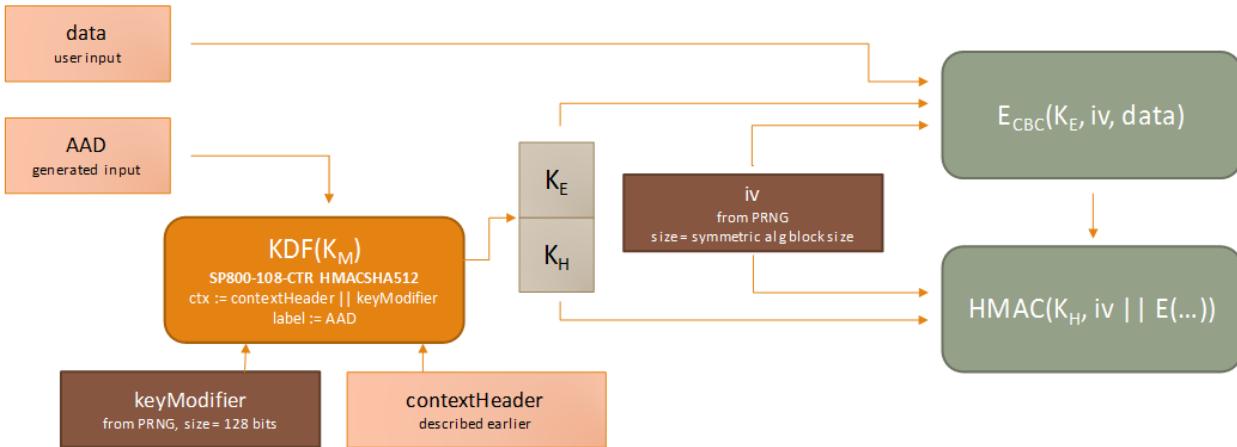
- 密钥派生密钥 (KDK) = K_M
- PRF = HMACSHA512
- 标签 = `additionalAuthenticatedData`
- 上下文 = `contextHeader \parallel keyModifier`

上下文标头的可变长度和本质就是我们要为其派生 K_E 和 K_H 的算法的指纹。密钥修饰符是每次调用随机生成一个 128 位字符串 `Encrypt` 并提供以确保与绝大多数概率 K_E 和 K_H 是唯一对于此特定身份验证加密操作，即使所有其他输入给 KDF 为常数。

CBC 模式下加密 + HMAC 验证操作， $|K_E|$ 的密钥长度是对称的块密码，和 $|K_H|$ 是的 HMAC 例程的摘要大小。对 GCM 加密 + 验证操作 $|K_H| = 0$ 。

CBC 模式下加密 + HMAC 验证

一旦 K_E 生成通过上述机制中，我们生成的随机初始化向量，并运行该对称的块密码算法来加密纯文本。然后通过使用密钥 K_H 以生成 MAC 初始向量 HMAC 例程将的初始化向量和已加密文本运行下面以图形方式表示此过程和返回值。



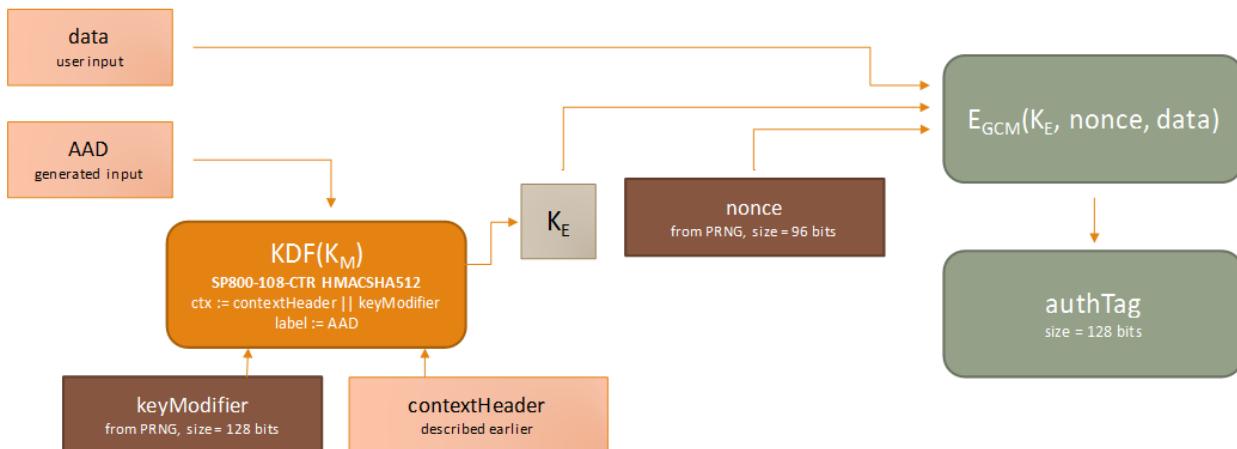
输出: = keyModifier | |iv | | $E_{cbc}(K_E, iv, 数据)$ | | $HMAC(K_H, iv | | E(...))$

注意

`IDataProtector.Protect` 实现将前面预置的幻标头和密钥 id 到之前将其返回到调用方的输出。因为神奇的标头和密钥 id 是隐式的一部分 `AAD`, 因为密钥修饰符作为输入提供给 KDF, 这意味着 MAC 进行身份验证的最终返回负载的每个单字节

Galois/计数器模式加密 + 验证

一旦 K_E 生成通过上述机制中, 我们将生成随机的 96 位 nonce 和运行对称块加密算法, 来加密纯文本, 并生成的 128 位身份验证标记。



输出: = keyModifier | |nonce | | $E_{gcm}(K_E, nonce, 数据)$ | |authTag

注意

即使 GCM 本机支持 AAD 的概念, 我们有一个要仍输入 AAD 仅对原始 KDF 中, 选择要传递到 GCM 其 AAD 参数为空字符串。这样做的原因是两个折叠。首先, 以支持灵活性我们永远不希望将 K_M 直接用作加密密钥。此外, GCM 有一定非常严格的一致性要求其输入。GCM 加密例程是曾调用对两个或更清晰的概率具有相同 (键, nonce) 的输入数据集对不能超过 2^{32} 。如果我们修复 K_E 我们无法执行多个 2^{32} 加密操作之前运行与我们的 2^{32} 限制。这可能看起来非常大量的操作, 但在几天内, 这些密钥的正常生存期内良好的高流量 web 服务器可以通过 40 亿个请求。保持合规的 2^{32} 我们继续使用 128 位密钥修饰符和 96 位 nonce, 因而可真正扩展任何给定 K_M 的可用操作计数-32 概率限制。为简单起见的设计中, 我们将共享 CBC 和 GCM 操作之间的 KDF 代码路径, 并且由于 AAD 已被视为 KDF 中没有无需将其转发给 GCM 例程。

在 ASP.NET 核心的上下文头

2018/4/10 • 11 min to read • [Edit Online](#)

背景和理论上

在数据保护系统中，"密钥"意味着可以提供的对象进行身份验证加密服务。每个键由唯一的 id (GUID) 和它所携带算法信息和 entropic 材料。其用途是，每个密钥执行唯一平均信息量，但系统不能强制实施的而我们还需要考虑的开发人员可能会通过修改密钥链中的现有密钥的算法信息手动更改密钥链。若要实现提供这种情况下我们安全要求数据保护系统具有的概念的加密灵活性，这样，安全地跨多个加密算法使用单个 entropic 值。

大多数系统支持的加密灵活性做到这一点包括有关负载之内算法某些标识信息。算法的 OID 通常是适合于此。但是，我们遇到的一个问题是有很多方法来指定相同的算法："AES"(CNG) 托管 Aes、AesManaged、AesCryptoServiceProvider、AesCng 和（如果有特定的参数）的 RijndaelManaged 类都确实是相同首先，我们将需要维护所有这些到正确的 OID 的映射。如果开发人员想要提供自定义算法（或甚至另一个实现的 AES！），他们将需要告诉我们其 OID。此额外的注册步骤使系统配置特别棘手。

回顾一下，我们决定我们已从错误方向接近问题。OID 提示您的算法是什么，但我们不实际关心这。如果我们需要在两个不同的算法中安全地使用单个 entropic 值，它不是我们要知道算法的实际的所必要的。我们实际上关心是它们的行为方式。任何有效的对称块加密算法也是强伪随机排列 (PRP)：修复（键，链接模式，IV，纯文本）的输入和已加密文本输出将与急剧的概率为不同于任何其他对称的块密码提供的相同的输入的算法。同样，任何相当的加密哈希函数也是强伪函数 (PRF)，并且给定的一组固定的输入其输出将极其能不同于任何其他加密哈希函数。

我们使用强 PRPs 和 PRFs 的概念构建上下文头。此上下文头实质上是充当稳定指纹通过对于任何给定的操作，所用的算法，并提供所需的数据保护系统的加密灵活性。此标头是可重现和更高版本用作的一部分子项派生过程。有两个不同的方法来生成具体的操作模式的基础的算法取决于上下文头。

CBC 模式下加密 + HMAC 身份验证

上下文头由以下组件组成：

- [16 位]值 00 00，是一个标记，这意味着"CBC 加密 + HMAC 身份验证"。
- [32 位]对称块加密算法密钥长度（以字节为单位，big endian）。
- [32 位]块大小（以字节为单位，big endian）的对称块加密算法。
- [32 位]HMAC 算法的密钥长度（以字节为单位，big endian）。（当前密钥的大小始终与匹配的摘要大小。）
- [32 位]HMAC 算法（以字节为单位，big endian）摘要大小。
- EncCBC (K_E、IV, "")，它是提供空字符串输入对称的块密码算法的输出，并且其中 IV 是全为零的向量。下面描述了 K_E 的构造。
- MAC (K_H, "")，这是提供空字符串输入 HMAC 算法的输出。下面描述了 K_H 的构造。

理想情况下，我们无法 K_E 和 K_H 传递全为零的向量。但是，我们想要避免这种情况，其中基础算法检查存在的共享密钥的安全性执行任何操作（值得注意的是 DES 和 3DES）之前，它可以阻止使用如全为零向量的简单或可重复模式。

相反，我们使用 NIST SP800 108 KDF 中计数器模式（请参阅[NIST SP800-108](#), sec. 5.1）具有零长度键、标签和上下文和作为基础 PRF HMACSHA512。我们派生 |K_E| + |K_H| 字节的输出，然后将分解结果为 K_E 和 K_H 本身。数学上，这表示，如下所示。

$$(K_E \parallel K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")$$

示例：AES 192 CBC + HMACSHA256

作为示例，请考虑对称块加密算法是 AES 192 CBC，其中的验证算法是 HMACSHA256 这种情况。系统将生成使用以下步骤的上下文标头。

首先，让 $(K_E || K_H) = SP800_108_CTR$ ($\text{prf} = \text{HMACSHA512}$, 键 = "", 标签 = "", 上下文 = ""), 其中 $|K_E| = 192$ 位和 $|K_H| =$ 每个指定的算法的 256 位。这将导致 $K_E = 5BB6...21DD$ 和 $K_H = A04A...00A9$ 在下面的示例 00A9:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0  
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2  
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4  
B7 92 3D BF 59 90 00 A9
```

接下来，计算 $\text{Enc_CBC}(K_E, IV, "")$ 对于给定 IV 的 AES-192-CBC = 0 * 和 K_E 按上面所述。

```
result := F474B1872B3B53E4721DE19C0841DB6F
```

接下来，计算 $\text{MAC}(K_H, "")$ 为 HMACSHA256 给定 K_H 按上面所述。

```
result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C
```

这将产生下面的完整上下文标头：

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00  
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41  
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60  
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36  
22 0C
```

此上下文标头是经过身份验证的加密算法对 (AES 192 CBC 加密 + HMACSHA256 验证) 的指纹。组件，如所述[上面](#)是：

- 标记 (00 00)
- 块密码密钥长度 (00 00 00 18)
- 块密码块大小 (00 00 00 10)
- HMAC 密钥长度 (00 00 00 20)
- HMAC 摘要的大小 (00 00 00 20)
- 块密码 PRP 输出 (F4 74-DB 6F) 和
- HMAC PRF 输出 (D4 79-结束)。

注意

CBC 模式下加密 + HMAC 身份验证上下文标头生成而不考虑算法实现是否提供通过 Windows CNG 或托管 SymmetricAlgorithm 和 KeyedHashAlgorithm 类型相同的方式。这允许在不同操作系统上运行的应用程序可靠地生成相同的上下文标头，即使之间的 Os 不同算法的实现。(在实践中，KeyedHashAlgorithm 不必是正确的 HMAC。它可以是任何加密哈希算法类型。)

示例：3DES 192 CBC + HMACSHA1

首先，让 $(K_E || K_H) = SP800_108_CTR$ ($\text{prf} = \text{HMACSHA512}$, 键 = "", 标签 = "", 上下文 = ""), 其中 $|K_E| = 192$ 位和 $|K_H| =$ 每个指定的算法的 160 位。这将导致 $K_E = A219...E2BB$ 和 $K_H = DC4A...00A9$ 在下面的示例 B464:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22  
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E  
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

接下来, 计算 $\text{Enc_CBC}(K_E, IV, "")$ 对于给定 IV 的 3DES-192-CBC = 0 * 和 K_E 按上面所述。

结果:= ABB100F81E53E10E

接下来, 计算 $\text{MAC}(K_H, "")$ 为 HMACSHA1 给定 K_H 按上面所述。

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

此示例的完整上下文标头, 这是经过身份验证的指纹将产生如下所示的加密算法对 (3DES 192 CBC 加密 + HMACSHA1 验证):

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00  
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF  
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

组件, 如下所示细分:

- 标记 (00 00)
- 块密码密钥长度 (00 00 00 18)
- 块密码块大小 (00 00 00 08)
- HMAC 密钥长度 (00 00 00 14)
- HMAC 摘要的大小 (00 00 00 14)
- 块密码 PRP 输出 (AB B1-E1 0E) 和
- HMAC PRF 输出 (76 EB-结束)。

Galois/计数器模式加密 + 身份验证

上下文标头由以下组件组成:

- [16 位]值 00 01, 是一个标记, 这意味着"GCM 加密 + 身份验证"。
- [32 位]对称块加密算法密钥长度 (以字节为单位, big endian)。
- [32 位]在经过身份验证的加密操作期间使用的 nonce 大小 (以字节为单位, big endian)。(对于我们的系统, 这固定的 nonce 大小 = 96 位。)
- [32 位]块大小 (以字节为单位, big endian) 的对称块加密算法。(为 GCM, 这固定的块大小 = 128 位。)
- [32 位]身份验证标记大小 (以字节为单位, big endian) 已经过身份验证的加密函数生成。(对于我们的系统, 这固定为标记大小 = 128 位。)
- [128 位] Enc_GCM 的标记 ($K_E, \text{nonce}, ""$), 它是给定的空字符串输入对称的块密码算法的输出, 并且其中 nonce 是 96 位全为零向量。

K_E 派生使用相同的机制, 如下所示 CBC 加密 + HMAC 身份验证方案。但是, 由于在此处 play 中没有任何 K_H , 我们实质上具有 $|K_H| = 0$, 且该算法会折叠到下面的表单。

$K_E = \text{SP800_108_CTR}(\text{prf} = \text{HMACSHA512}, \text{key} = "", \text{label} = "", \text{context} = "")$

示例: AES 256 GCM

首先, 让 $K_E = SP800_108_CTR$ ($\text{prf} = \text{HMACSHA512}$, 键 = "", 标签 = "", 上下文 = ""), 其中 $|K_E| = 256$ 位。

$K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8$

接下来, 计算 Enc_GCM 的身份验证标记 $(K_E, \text{nonce}, "")$ 适用于给定 nonce 的 $\text{AES-256-GCM} = 096$ 和 K_E 按上面所述。

$\text{result} := E7DCCE66DF855A323A6BB7BD7A59BE45$

这将产生下面的完整上下文头:

```
00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00  
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59  
BE 45
```

组件, 如下所示细分:

- 标记 (00 01)
- 块密码密钥长度 (00 00 00 20)
- nonce 的大小 (00 00 00 0 C)
- 块密码块大小 (00 00 00 10)
- 身份验证标记大小 (00 00 00 10) 和
- 从运行块密码的身份验证标记 (E7 DC-结束)。

ASP.NET 核心中的密钥管理

2018/4/10 • 7 min to read • [Edit Online](#)

数据保护系统自动管理的主要密钥用于保护和取消保护负载的生存期。每个密钥可以中四个阶段之一存在：

- 创建密钥存在于密钥环，但尚未激活。密钥不应用于新保护操作直到足够的时间已过的密钥已有机会传播到使用此密钥环的所有计算机。
- 活动-密钥存在于密钥环，应使用对所有新的保护操作。
- 过期的密钥已运行其自然的生存期和应不再用于新的保护操作。
- 吊销-密钥遭到破坏，并且不必再用于新的保护操作。

创建、活动和过期密钥可能都用来取消保护传入负载。默认情况下的吊销的密钥不可能用于取消保护的负载，但应用程序开发人员可以[重写此行为](#)如有必要。

警告

开发人员可能想要从密钥链中删除密钥，（例如，通过从文件系统中删除相应的文件）。此时，保护的密钥的所有数据都永久密匙，且都没有紧急替代具有吊销键可能没有。删除键是真正破坏性的行为，并因此数据保护系统公开没有第一类 API 执行此操作。

默认密钥选择

当数据保护系统从后备存储库读取密钥环时，它将尝试查找密钥环的“默认”密钥。默认密钥用于新的保护操作。

常规的启发式方法是数据保护系统选择最新的激活日期与默认密钥的密钥。（没有小奶油因素，以允许服务器到服务器时钟偏差。）如果密钥已过期或被吊销，并且如果应用程序禁用自动密钥生成，则将生成新的密钥与每个即时激活[密钥过期和滚动](#)下面的策略。

数据保护系统的原因会立即生成新密钥，而不是回退到不同的密钥是生成新密钥应视为已激活新密钥之前的所有键的隐式过期时间。大致了解是，新密钥可能已配置了不同的算法或静态加密机制比旧的密钥，并且系统应首选的当前配置，而回退。

没有异常。如果应用程序开发人员可以[禁用自动密钥生成](#)，然后数据保护系统必须选择内容的默认密钥。在此回退方案中，系统将选择具有最新的激活日期、非吊销键，而且会优先有时间传播到群集中其他计算机的密钥。回调系统可以结束因此选择过期的默认密钥。回调系统将永远不会选择将吊销的键用作默认密钥，并将如果密钥环为空或已被吊销的每个键通过然后系统将产生在初始化时出错。

密钥的过期和滚动

创建密钥时，它自动具有提供的激活日期为 {now + 2 天} 和 {now + 90 天} 的到期日期。之前激活给予密钥的时间才能传遍整个系统的 2 天延迟。也就是说，它允许指向的后备存储在其他应用程序密钥观察在其下一步的自动刷新期内，从而最大化，密钥环执行成为的活动已传播到所有应用程序，可能需要使用它的可能性。

如果默认密钥将在 2 天内过期，并且密钥环还没有密钥将处于活动状态的默认密钥到期后，数据保护系统将自动保留密钥环的新键。此新的密钥都有 {默认密钥的过期日期} 的激活日期和到期日期的 {now + 90 天}。这允许系统自动的服务不会发生中断定期轮转密钥。

可能有情况下将使用即时激活中创建密钥。一个示例是当应用程序尚未运行的时间和过期密钥链中的所有键。当发生这种情况时，系统将为密钥提供 {现在} 的不正常的 2 天激活延迟的情况下的激活日期。

默认密钥生存期是 90 天，但这是可配置，如以下示例所示。

```
services.AddDataProtection()
    // use 14-day lifetime instead of 90-day lifetime
    .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

管理员还可以更改默认系统范围内，但显式调用 `SetDefaultKeyLifetime` 将覆盖任何系统范围的策略。默认密钥生存期不能短于 7 天。

自动密钥环刷新

数据保护系统初始化时，它从基础存储库读取密钥环，并将其缓存在内存中。此缓存允许未命中的后备存储的情况下继续保护和取消保护操作。大约每隔 24 小时或当前的默认密钥过期，不管先满足后，系统会自动查看更改的后备存储。

警告

开发人员应极少数情况下（如果有）需要直接使用密钥管理 API。数据保护系统将执行自动密钥管理，如上面所述。

数据保护系统公开接口 `IKeyManager` 可用来检查和更改的密钥链。DI 系统提供的实例 `IDataProtectionProvider` 还可以提供的实例 `IKeyManager` 供你使用。或者，你可以请求 `IKeyManager` 直接从 `IServiceProvider` 以下示例所示。

修改键环（显式创建一个新密钥或执行吊销）的任何操作将使内存中缓存失效。下次调用 `Protect` 或 `Unprotect` 将导致数据保护系统读取密钥环，并重新创建缓存。

下面的示例演示如何使用 `IKeyManager` 接口来检查和操作密钥环，包括撤销的现有密钥和手动生成新密钥。

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring
        services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
        }
    }
}
```

```

}

// revoke all keys in the key ring
keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
Console.WriteLine("Revoked all existing keys.");

// add a new key to the key ring with immediate activation and a 1-month expiration
keyManager.CreateNewKey(
    activationDate: DateTimeOffset.Now,
    expirationDate: DateTimeOffset.Now.AddMonths(1));
Console.WriteLine("Added a new key.");

// list all keys in the key ring
allKeys = keyManager.GetAllKeys();
Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
foreach (var key in allKeys)
{
    Console.WriteLine($"Key {key.KeyId}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
}
}

/*
 * SAMPLE OUTPUT
 *
 * Performed a protect operation.
 * The key ring contains 1 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
 * Revoked all existing keys.
 * Added a new key.
 * The key ring contains 2 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
 * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
*/

```

密钥存储

数据保护系统具有，由此它将尝试自动推导出适当的密钥存储位置和 rest 机制加密启发式方法。这也是由应用程序开发人员可配置的。以下文档讨论这些机制的现成实现：

- [内置密钥存储提供程序](#)
- [框 rest 服务提供商的密钥加密](#)

在 ASP.NET 核心中的密钥存储提供程序

2018/4/10 • 2 min to read • [Edit Online](#)

默认情况下，数据保护系统使用[启发式方法](#)以确定应将在其中保留加密的密钥材料。开发人员可以重写启发式方法，并手动指定的位置。

注意

如果您指定一个显式的密钥保持位置，将取消注册数据保护系统在 rest 机制启发式方法提供的默认密钥加密，以便密钥将不再加密对静止。它具有，建议你此外[指定显式密钥加密机制](#)对于生产应用程序。

数据保护系统附带了几个内置密钥存储提供程序。

文件系统

我们预计很多应用程序将使用的文件基于系统的密钥存储库。若要此配置，调用[PersistKeysToFileSystem](#)配置例程如下所示。提供 `DirectoryInfo` 指向应在其中存储密钥的存储库。

```
sc.AddDataProtection()
    // persist keys to a specific directory
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
```

`DirectoryInfo` 可以指向本地计算机上的目录或它可以指向网络共享上的文件夹。如果指向本地计算机上的目录（和的方案是，只有本地计算机上的应用程序将需要使用此存储库），请考虑使用[Windows DPAPI](#)来加密存放的密钥。否则，请考虑使用[X.509 证书](#)来加密存放的密钥。

Azure 和 Redis

`Microsoft.AspNetCore.DataProtection.AzureStorage` 和 `Microsoft.AspNetCore.DataProtection.Redis` 包允许将你的数据保护密钥存储在 Azure 存储空间或 Redis 缓存。可以在 web 应用程序的多个实例之间共享密钥。你的 ASP.NET Core 应用可以跨多个服务器共享身份验证 cookie 或 CSRF 保护。若要在 Azure 上配置，调用之一[PersistKeysToAzureBlobStorage](#)重载，如下所示。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS token>"));

    services.AddMvc();
}
```

另请参阅[Azure 测试代码](#)。

若要在 Redis 上配置，调用之一[PersistKeysToRedis](#)重载，如下所示。

```
public void ConfigureServices(IServiceCollection services)
{
    // Connect to Redis database.
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToRedis(redis, "DataProtection-Keys");

    services.AddMvc();
}
```

有关详细信息，请参阅以下主题：

- [StackExchange.Redis ConnectionMultiplexer](#)
- [Azure Redis 缓存](#)
- [Redis 测试代码。](#)

注册表

有时应用程序可能没有写到文件系统的访问权限。请考虑其中作为是虚拟服务帐户（如 w3wp.exe 的应用程序池标识）运行应用程序的方案。在这些情况下，管理员可能已设置为服务帐户标识的相应 ACL 的注册表项。调用 [PersistKeysToRegistry](#) 配置例程如下所示。提供 `RegistryKey` 指向存储加密密钥/值的位置。

```
sc.AddDataProtection()
    // persist keys to a specific location in the system registry
    .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
```

如果作为持久性机制使用系统注册表，请考虑使用[Windows DPAPI](#)来加密存放的密钥。

自定义密钥存储库

如果不适合的内置机制，开发人员可以通过提供自定义指定自己的密钥保持机制 `IXmlRepository`。

在 ASP.NET Core 中存放的密钥加密

2018/5/14 • 4 min to read • [Edit Online](#)

默认情况下，数据保护系统使用启发式方法来确定如何加密的密钥材料应加密对静止。开发人员可以重写启发式方法，并手动指定应如何静态加密密钥。

注意

如果指定在 rest 机制显式密钥加密时，数据保护系统将取消注册启发式方法提供的默认密钥存储机制。你必须[指定显式的密钥存储机制](#)，否则数据保护系统将无法启动。

数据保护系统都附带有三个框中的密钥加密机制。

Windows DPAPI

仅在 Windows 上，此机制是可用。

当使用 Windows DPAPI 时，将通过加密密钥材料[CryptProtectData](#)之前保存到存储。DPAPI 是一种用于将永远不会读取当前计算机之外的数据的合适的加密机制（尽管它可以备份到 Active Directory 这些密钥，请参阅[DPAPI 和漫游配置文件](#)）。例如，若要配置 DPAPI 密钥在 rest 加密。

```
sc.AddDataProtection()
    // only the local user account can decrypt the keys
    .ProtectKeysWithDpapi();
```

如果 `ProtectKeysWithDpapi` 调用不带任何参数，仅当前 Windows 用户帐户，才能解密持久化的密钥材料。你可以选择指定中所示，应导致计算机（而不仅仅是当前用户帐户）上的任何用户帐户能够解密的密钥材料，下面的示例。

```
sc.AddDataProtection()
    // all user accounts on the machine can decrypt the keys
    .ProtectKeysWithDpapi(protectToLocalMachine: true);
```

X.509 证书

此机制不可用于 .NET Core 1.0 或 1.1。

如果你的应用程序分布在多台计算机，则可能很方便多台计算机之间分发共享的 X.509 证书并配置应用程序，此证书用于加密存放的密钥。有关示例，请参阅下文。

```
sc.AddDataProtection()
    // searches the cert store for the cert with this thumbprint
    .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
```

由于.NET Framework 限制支持仅使用 CAPI 私钥的证书。请参阅[基于证书的加密使用 Windows DPAPI NG](#)下面这些限制的可能解决方法。

Windows DPAPI NG

此机制是仅适用于 Windows 8 / Windows Server 2012 和更高版本。

操作系统从 Windows 8 开始，支持 DPAPI NG（也称为 CNG DPAPI）。Microsoft 是布局其使用方案，如下所示。

但是，云计算，通常需要该内容的加密在一台计算机进行在另一台解密。因此，从开始 Windows 8，Microsoft 扩展的使用相对比较简单的 API 以覆盖云方案的想法。称为 DPAPI NG，此新 API，可安全地通过保护它们的一组可用于取消这些不同的计算机上保护后正确的身份验证和授权的主体到共享机密（密钥、密码、密钥材料）和消息。

从有关 CNG DPAPI

主体编码为保护描述符规则。请考虑以下示例中，该加密密钥材料，以便仅已加入域的用户具有指定 SID 都可以解密密钥材料。

```
sc.AddDataProtection()
    // uses the descriptor rule "SID=S-1-5-21-..."
    .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
    flags: DpapiNGProtectionDescriptorFlags.None);
```

另外，还有的无参数重载 `ProtectKeysWithDpapiNG`。这是用于指定规则的便捷方法“SID = 挖掘”，其中挖掘是当前 Windows 用户帐户的 SID。

```
sc.AddDataProtection()
    // uses the descriptor rule "SID={current account SID}"
    .ProtectKeysWithDpapiNG();
```

在此方案中，AD 域控制器负责分发 DPAPI NG 操作所使用的加密密钥。目标用户将能够解密从任何加入域的计算机的加密的负载（前提是进程正在其标识下运行时）。

基于证书的加密使用 Windows DPAPI NG

如果在 Windows 8.1 上运行 / Windows Server 2012 R2 或更高版本，你可以使用 Windows DPAPI NG 执行基于证书的加密，即使在.NET Core 上运行该应用程序。若要充分利用此功能，使用规则描述符字符串“证书 = HashId:thumbprint”，其中指纹是要使用的证书的十六进制编码 SHA1 指纹。有关示例，请参阅下文。

```
sc.AddDataProtection()
    // searches the cert store for the cert with this thumbprint
    .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0",
    flags: DpapiNGProtectionDescriptorFlags.None);
```

在此存储库指向任何应用程序必须在 Windows 8.1 上运行 / Windows Server 2012 R2 或更高版本能够解密此密钥。

自定义密钥加密

如果不适合的内置机制，开发人员可以通过提供自定义指定自己的密钥加密机制 `IXmlEncryptor`。

密钥不可变性和 ASP.NET Core 中的密钥设置

2018/4/10 • 1 min to read • [Edit Online](#)

一旦对象持久保留到后备存储，其表示形式是永久固定的。新的数据可以添加到后备存储，但可以永远不会更改现有数据。此行为的主要目的是为了防止数据损坏。

此行为的一个后果是，一旦给后备存储编写了一个键，不可变。其创建、激活和到期日期可以永远不会更改，但它可以通过使用吊销 `IKeyManager`。此外，其基础的算法信息、主密钥材料和加密 `rest` 属性也是不可变。

如果开发人员更改会影响密钥保持任何设置，这些更改不会生效只有在下次时都会生成一个密钥，是指在通过显式调用 `IKeyManager.CreateNewKey` 或通过数据保护系统的自己[自动密钥生成](#)行为。影响密钥持久性的设置如下所示：

- [默认密钥生存时间](#)
- [在 rest 机制密钥加密](#)
- [在项包含的算法信息](#)

如果你需要这些设置，以便早于下一步的自动密钥滚动时间启动，请考虑使显式调用 `IKeyManager.CreateNewKey` 强制创建新的密钥。请记得提供显式激活日期（{现在 + 2 天} 是一个很好的经验法则以允许更改传播的时间）和调用中的到期日期。

提示

点击存储库中的所有应用程序应指定要用的相同设置 `IDataProtectionBuilder` 扩展方法。否则，保存的密钥的属性将取决于调用的密钥生成例程的特定应用程序。

在 ASP.NET 核心中的密钥存储格式

2018/5/17 • 3 min to read • [Edit Online](#)

对象存储在 XML 表示形式中的其余部分。密钥存储的默认目录为 %localappdata%\asp.net\dataprotection-keys\。

<密钥 > 元素

注册表项存在作为密钥存储库中的顶级对象。按照约定密钥具有 filename密钥-{guid}.xml，其中 {guid} 是密钥的 id。每个此类文件包含一个密钥。文件的格式如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

<密钥 > 元素包含以下属性和子元素：

- 密钥 id。此值是被看作是授权;文件名是只需以方便人们阅读过去。
- 版本<密钥 > 元素，当前固定为 1。
- 密钥的创建、激活和到期日期。
- A<描述符 > 元素，它包含有关该注册表项中包含的经过身份验证的加密实现的信息。

在上面的示例中，密钥的 id 是 {80732141-ec8f-4b80-af9c-c4d2d1ff8901}，它已创建并将在 2015 年 3 月 19 日激活且它具有 90 天的生存期。(有时激活日期可能略有如此示例所示的创建日期之前。这是由于 n 它在 API 工作和无碍在实践中的方式。)

<描述符 > 元素

外部<描述符 > 元素包含属性 deserializerType，这是实现 IAuthenticatedEncryptorDescriptorDeserializer 的类型的程序集限定名称。此类型是负责读取内部<描述符 > 元素和分析中包含的信息。

特定的格式<描述符 > 元素取决于经过身份验证的加密程序实现的键，封装和每个反序列化程序类型所预期的此略有不同的格式。一般情况下，不过，此元素将包含算法的信息(名称、类型、Oid，或类似)和机密密钥材料。在上面的示例中，该描述符指定此密钥包装 AES 256 CBC 加密 + HMACSHA256 验证。

<EncryptedSecret > 元素

元素它包含机密的密钥材料的加密的表单可能出现如果启用了加密对静止的机密。属性 decryptorType 将实现

IXmlDecryptor 的类型的程序集限定名称。此类型是负责读取内部元素和解密以恢复原始的纯文本。

与<描述符>, 特定的格式元素取决于正在使用的静态加密机制。在上面的示例中, 每个注释中使用 Windows DPAPI 对主密钥进行加密。

<吊销>元素

吊销存在作为密钥存储库中的顶级对象。按照约定吊销具有 filename~~吊销~~-{时间戳}.xml (适用于特定日期之前撤消所有键) 或吊销-{guid}.xml (对于都撤消特定键)。每个文件包含单个<吊销>元素。

对于各个密钥吊销, 该文件的内容将如下。

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

在这种情况下, 只有指定的密钥已吊销。如果密钥 id 为 "*", 但是, 如以下示例中, 其创建日期是在指定的吊销日期之前的所有键都被都吊销。

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

<原因>由系统永远不会读取元素。它是只需存储吊销的用户可读的原因方便位置。

ASP.NET 核心中的临时数据保护提供程序

2018/4/10 • 1 min to read • [Edit Online](#)

有的方案：应用程序需要的 `IEventDataProtectionProvider`。例如，开发人员可能只需在一次性的控制台应用程序中，进行试验或应用程序本身是暂时性（已编写脚本或单元测试项目）。若要支持这些方案 `Microsoft.AspNetCore.DataProtection` 包括一种类型 `EphemeralDataProtectionProvider`。此类型提供的基本实现 `IEventDataProtectionProvider` 其密钥的存储库保留仅在内存中和不写出到任何后备存储。

每个实例 `EphemeralDataProtectionProvider` 使用其自身唯一的主密钥。因此，如果 `IDataProtector` 处于 `EphemeralDataProtectionProvider` 生成受保护的负载，该负载仅受等效 `IDataProtector`（给定相同目的链）位于同一个取得 root 权限 `EphemeralDataProtectionProvider` 实例。

下面的示例演示如何实例化 `EphemeralDataProtectionProvider` 并使用它来保护和取消保护数据。

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protect returned: CfDJ8AAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
 * Unprotect returned: Hello!
 * << throws CryptographicException >>
 */
```

ASP.NET Core 中的兼容性

2018/4/10 • 1 min to read • [Edit Online](#)

- 在 ASP.NET Core 中替换 ASP.NET <machineKey>

将在 ASP.NET Core ASP.NET machineKey

2018/4/10 • 2 min to read • [Edit Online](#)

实现 `<machineKey>` 在 ASP.NET 中的元素是可替换。这允许对 ASP.NET 加密例程的大多数调用，以通过替换数据保护机制，包括新的数据保护系统路由。

包安装

注意

新的数据保护系统只能安装到现有的 ASP.NET 应用程序面向.NET 4.5.1 或更高版本。安装将失败，如果应用程序面向.NET 4.5 或更低。

若要安装新的数据保护系统到现有的 ASP.NET 4.5.1+ 项目之后，安装包

`Microsoft.AspNetCore.DataProtection.SystemWeb`。这将实例化数据保护系统使用默认配置设置。

当你安装此程序包时，它将插入行到 `Web.config` 这是告诉 ASP.NET，以将其用于最加密操作，包括窗体身份验证、视图状态和调用 `MachineKey.Protect`。插入的行，如下所示读取。

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

提示

你可以判断是否处于活动状态通过检查类似的字段访问新的数据保护系统 `__VIEWSTATE`，这应该以开头 "CfDJ8" 以下示例所示。"CfDJ8" 是标识受数据保护系统的负载的幻 "09 F0 C9 F0" 标头的 base64 表示。

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZ0pk..." />
```

包配置

数据保护系统是具有默认零安装程序配置实例化。但是，由于默认情况下会将密钥保存到本地文件系统，这不起作用的应用程序的场中部署的状态。若要解决此问题，你可以通过创建一种类型的子类 `DataProtectionStartup` 提供配置，并重写其 `ConfigureServices` 方法。

下面是一个示例配置将在其中保留密钥，又如何要加密对静止的自定义的数据保护启动类型。它还通过提供其自己的应用程序名称来重写默认应用程序隔离策略。

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"))  

                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

提示

你还可以使用 `<machineKey applicationName="my-app" ... />` 代替 `SetApplicationName` 显式调用。这是不会强制执行开发人员可以创建 `DataProtectionStartup` 派生类型，如果他们想要配置所有已设置应用程序名称的方便机制。

若要启用此自定义配置，请返回到 Web.config，并查找 `<appSettings>` 包安装到的配置文件添加的元素。它将类似以下的标记：

```
<appSettings>
<!--
If you want to customize the behavior of the ASP.NET Core Data Protection stack, set the
"aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
type which subclasses Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
-->
<add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

填写的空白值与你刚刚创建的 `DataProtectionStartup` 派生的类型的程序集限定名称。如果应用程序的名称是 `DataProtectionDemo`，这将如下所示下面。

```
<add key="aspnet:dataProtectionStartupType"
      value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

最新配置数据保护系统现已在应用程序中使用。

强制实施 HTTPS 在 ASP.NET 核心

2018/5/17 • 4 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

本文档说明如何:

- 所有请求需要 HTTPS。
- 所有 HTTP 请求都重定向到 HTTPS。

警告

执行不使用 `RequireHttpsAttribute` 接收敏感信息的 Web API。`RequireHttpsAttribute` 使用 HTTP 状态代码将从 HTTP 到 HTTPS 的浏览器重定向。API 客户端可能无法理解或遵循从 HTTP 到 HTTPS 的重定向。此类客户端可能会通过 HTTP 发送信息。Web API 应具有下列任一:

- 不在 HTTP 上侦听。
- 关闭与状态代码 400 (错误请求) 的连接并不为请求提供服务。

需要 HTTPS

我们建议所有 ASP.NET 核心 web 应用调用 `UseHttpsRedirection` 将所有 HTTP 请求重定向到 HTTPS。如果 `UseHsts` 调用在应用中, 它必须调用之前 `UseHttpsRedirection`。

下面的代码调用 `UseHttpsRedirection` 中 `Startup` 类:

[!code-csharp@sample]

下面的代码:

[!code-csharp@sample]

- 集 `RedirectStatusCode`。
- 将 HTTPS 端口设置为 5001。

`RequireHttpsAttribute` 用于需要 HTTPS。`[RequireHttpsAttribute]` 可修饰控制器或方法, 或可以全局应用。若要全局应用该属性, 以下代码添加到 `ConfigureServices` 中 `Startup`:

[!code-csharp@]

前面的突出显示的代码中需要的所有请求都使用 `HTTPS`; 因此, HTTP 请求将被忽略。以下突出显示的代码将所有 HTTP 请求重定向到 HTTPS:

[!code-csharp@]

有关详细信息, 请参阅 [URL 重写中间件](#)。

全局需要 HTTPS (`options.Filters.Add(new RequireHttpsAttribute());`) 是最佳安全方案。应用 `[RequireHttps]` 特性应用到所有控制器/Razor 页面不会被视为尽可能安全全局需要 HTTPS。你不能保证 `[RequireHttps]` 添加新控制器和 Razor 页时应用特性。

HTTP 严格的传输安全协议 (HSTS)

每个 [OWASP](#), [HTTP 严格传输安全 \(HSTS\)](#) 是由 web 应用程序通过使用特殊的响应标头指定可以选择使用的安

全增强。支持的浏览器收到此标头后该浏览器将阻止从正在通过 HTTP 发送到指定的域的任何通信，并改为将通过 HTTPS 发送的所有通信。它还可以防止 HTTPS 单击通过在浏览器上的提示。

ASP.NET 核心 2.1 或更高版本实现与 HSTS `useHsts` 扩展方法。下面的代码调用 `UseHsts` 时应用程序不在**开发模式**:

[!code-csharp@sample]

`UseHsts` 不建议在开发过程中的因为 HSTS 标头是高度可通过浏览器缓存。默认情况下，`UseHsts` 排除本地环回地址。

下面的代码：

[!code-csharp@sample]

- 设置预加载参数 Strict 传输安全标头。预加载不属[RFC HSTS 规范](#)，但若要预加载 HSTS 站点上执行全新安装的 web 浏览器都支持。有关详细信息，请参阅 <https://hstspreload.org/>。
- 使[includeSubDomain](#)，该方案 HSTS 策略适用于主机子域。
- 显式将 Strict 传输安全标头到的最大有效期参数设置为 60 天。如果未设置，默认值为 30 天。请参阅[最长时间指令](#)有关详细信息。
- 将添加 `example.com` 到的主机排除列表。

`UseHsts` 排除以下环回主机：

- `localhost`：IPv4 环回地址。
- `127.0.0.1`：IPv4 环回地址。
- `[::1]`：IPv6 环回地址。

前面的示例演示如何添加其他主机。

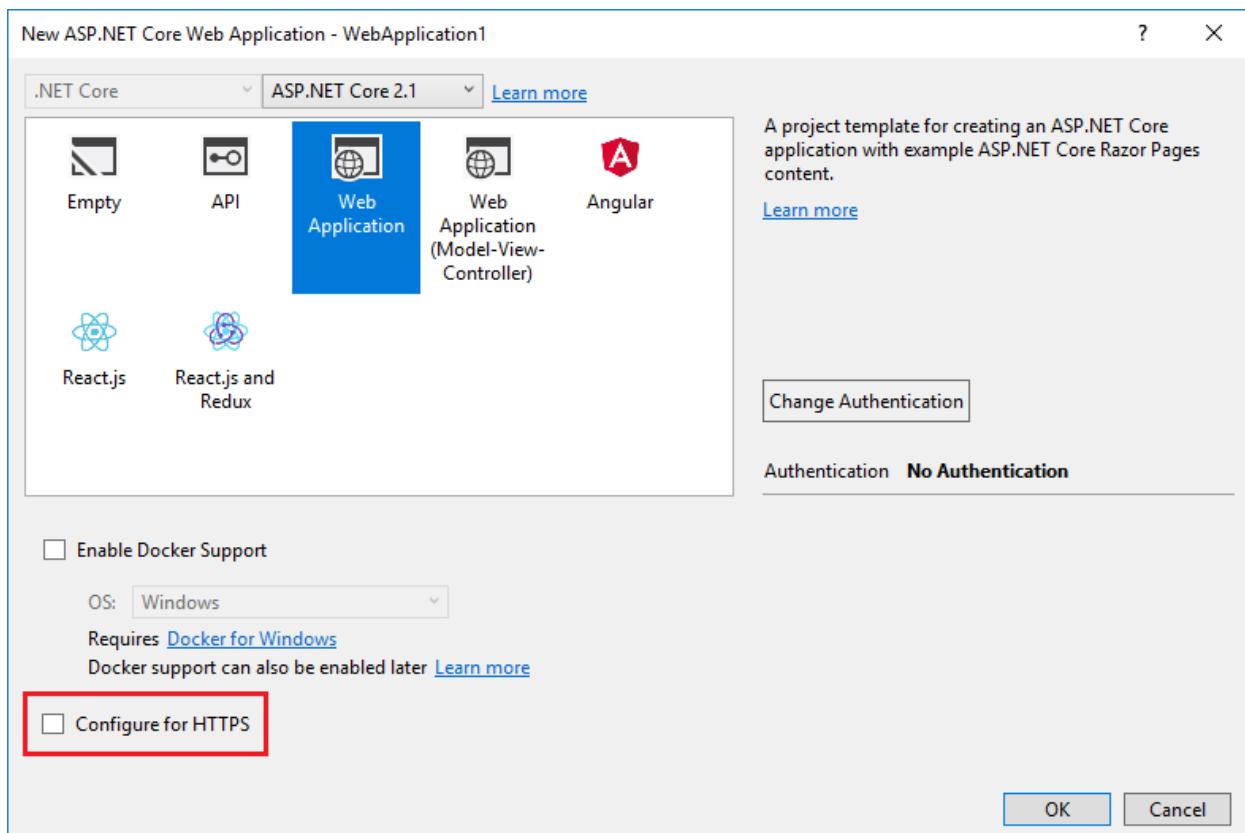
选择退出的 HTTPS 在项目创建

ASP.NET 核心 2.1 和更高版本（从 Visual Studio 或 dotnet 命令行）的 web 应用程序模板启用[HTTPS 重定向](#)和[HSTS](#)。对于不需要 HTTPS 的部署，你可以选择退出的 HTTPS。例如，不需要其中 HTTPS 正在处理外部在边缘，在每个节点使用 HTTPS 某些后端服务。

若要选择退出的 HTTPS：

- [Visual Studio](#)
- [.NET Core CLI](#)

取消选中对 **HTTPS 配置** 复选框。



如何为 Docker 设置开发人员证书

请参阅[此 GitHub 问题](#)。

安全存储中 ASP.NET Core 中开发的应用程序机密

2018/5/20 • 7 min to read • [Edit Online](#)

通过[Rick Anderson](#), [Daniel Roth](#), 和[Scott Addie](#)

[查看或下载示例代码\(如何下载\)](#)

[查看或下载示例代码\(如何下载\)](#)

本文档介绍用于存储和检索敏感数据的 ASP.NET Core 应用程序开发过程的技术。你应永远不会将密码或其他敏感数据存储在源代码中，并且你不应在开发过程中使用生产机密或测试模式。你可以存储和保护与 Azure 的测试和生产机密[Azure 密钥保管库配置提供程序](#)。

环境变量

使用环境变量以避免在代码中或在本地配置文件中的应用程序机密存储。环境变量重写所有以前指定的配置源的配置的值。

通过调用配置的环境变量值读取[AddEnvironmentVariables](#)中 `Startup` 构造函数：

```
[!code-csharp]
```

请考虑在其中一个 ASP.NET 核心 web 应用单个用户帐户已启用安全性。默认数据库连接字符串包含在项目的`appsettings.json`文件与键 `DefaultConnection`。默认连接字符串用于 LocalDB，在用户模式下运行，而且不需要密码。应用程序在部署期间，`DefaultConnection` 密钥值可以用环境变量的值重写。环境变量可以存储敏感的凭据与完整的连接字符串。

警告

环境变量通常存储在未加密的纯文本。如果计算机或进程受到攻击，则可以通过不受信任方访问环境变量。可能需要更多措施，防止用户的机密信息泄露。

密码管理器

密码管理器工具在 ASP.NET Core 项目的开发过程中存储敏感数据。在此上下文中，一种敏感数据是应用程序密钥。应用程序机密存储在单独的位置，从项目树中。应用程序机密是与特定项目关联，或共享跨多个项目。应用程序机密不签入源控件。

警告

密码管理器工具不加密存储的机密信息，并不应被视为受信任存储区。它是仅限开发目的。键和值存储在用户配置文件目录中的 JSON 配置文件中。

密码管理器工具的工作原理

密码管理器工具避开实现详细信息，例如哪里和如何存储值。无需知道这些实现的详细信息，可以使用该工具。这些值存储在[JSON](#)受系统保护用户配置文件文件夹中本地计算机上的配置文件：

- [Windows](#)

- macOS
- Linux

文件系统路径：

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

在前面文件路径中，将 `<user_secrets_id>` 与 `UserSecretsId` 中指定的值 `.csproj` 文件。

不编写代码所依赖的位置或使用密码管理器工具中保存的数据的格式。这些实现的详细信息如有更改。例如，密钥值不加密，但是可能在将来。

安装机密管理器工具

密码管理器工具是与.NET 核心 SDK 2.1 中的.NET 核心 CLI 捆绑。有关.NET 核心 SDK 2.0 及更早版本，则工具安装是必需的。

安装 `Microsoft.Extensions.SecretManager.Tools` ASP.NET Core 项目中的 NuGet 包：

```
[!code-xml]
```

在验证工具的安装命令行界面中执行以下命令：

```
dotnet user-secrets -h
```

密码管理器工具显示示例用法、选项和命令帮助：

```
Usage: dotnet user-secrets [options] [command]

Options:
-?|-h|--help           Show help information
--version               Show version information
-v|--verbose            Show verbose output
-p|--project <PROJECT> Path to project. Defaults to searching the current directory.
-c|--configuration <CONFIGURATION> The project configuration to use. Defaults to 'Debug'.
--id                   The user secret ID to use.

Commands:
clear     Deletes all the application secrets
list      Lists all the application secrets
remove    Removes the specified user secret
set      Sets the user secret to the specified value

Use "dotnet user-secrets [command] --help" for more information about a command.
```

注意

你必须在与位于同一目录 `.csproj` 文件以运行中定义的工具 `.csproj` 文件的 `DotNetCliToolReference` 元素。

设置机密

密码管理器工具进行存储用户配置文件中的特定于项目的配置设置操作。若要使用用户的机密信息，定义 `UserSecretsId` 中的元素 `PropertyGroup` 的 `.csproj` 文件。值 `UserSecretsId` 任意的但是唯一到项目。开发人员通常应生成的 GUID `UserSecretsId`。

```
[!code-xml]
```

```
[!code-xml]
```

提示

在 Visual Studio 中，右键单击解决方案资源管理器中的项目，然后选择**管理用户的机密信息**从上下文菜单。此手势添加 `UserSecretsId` 元素，为填充 GUID，`.csproj` 文件。Visual Studio 将打开 `secrets.json` 在文本编辑器中的文件。内容替换 `secrets.json` 要存储的键 / 值对。例如：

```
json { "Movies": { "ServiceApiKey": "12345", "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;Trusted_Connection=True;MultipleActiveResultSets=true" } }
```

定义一个键和其值组成应用程序密钥。密钥是与项目的关联 `UserSecretsId` 值。例如，从在其中的目录运行以下命令。`.csproj` 文件存在：

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

在前面的示例中，冒号表示 `Movies` 是一个对象文本替换 `ServiceApiKey` 属性。

密码管理器工具可以过使用从其他目录。使用 `--project` 选项提供文件系统路径，在此处 `.csproj` 文件存在。例如：

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project "C:\apps\WebApp1\src\WebApp1"
```

设置多个机密

可以通过管道传递到的 JSON 设置机密一批 `set` 命令。在下面的示例中，`input.json` 文件的内容通过管道传递给 `set` 命令。

- Windows
- macOS
- Linux

打开命令 shell，并执行以下命令：

```
type .\input.json | dotnet user-secrets set
```

访问机密

[ASP.NET 核心配置 API](#) 提供对机密 Manager 机密的访问。如果目标 .NET 核心 1.x 或 .NET Framework 安装 [Microsoft.Extensions.Configuration.UserSecrets](#) NuGet 包。

添加到用户机密配置源 `Startup` 构造函数：

```
[!code-csharp]
```

可以通过检索用户的机密信息 `Configuration` API：

```
[!code-csharp]
```

```
[!code-csharp]
```

包含密码的字符串替换

以纯文本格式存储密码是危险的。例如，数据库连接字符串存储在 `appsettings.json` 可能包括指定用户的密码：

```
{  
    "ConnectionStrings": {  
        "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User  
Id=johndoe;Password=pass123;MultipleActiveResultSets=true"  
    }  
}
```

更安全的方法是将密码存储的机密形式。例如：

```
dotnet user-secrets set "DbPassword" "pass123"
```

替换中的密码`appsettings.json`其占位符。在下面的示例中，`{0}` 用作到窗体的占位符 [复合格式字符串](#)。

```
{  
    "ConnectionStrings": {  
        "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User Id=johndoe;Password=  
{0};MultipleActiveResultSets=true"  
    }  
}
```

机密的值可以将其插入占位符以完成连接字符串：

```
[!code-csharp]
```

```
[!code-csharp]
```

列出机密

Assume the app's `secrets.json` file contains the following two secrets:

```
{  
    "Movies": {  
        "ServiceApiKey": "12345",  
        "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-  
1;Trusted_Connection=True;MultipleActiveResultSets=true"  
    }  
}
```

从在其中的目录运行以下命令。`.csproj`文件存在：

```
dotnet user-secrets list
```

显示以下输出：

```
Movies:ServiceApiKey = 12345  
Movies:ConnectionString = Server=(localdb)\\mssqllocaldb;Database=Movie-  
1;Trusted_Connection=True;MultipleActiveResultSets=true
```

在前面的示例中，在项名称中的冒号表示内的对象层次结构`secrets.json`。

删除单个机密

Assume the app's `secrets.json` file contains the following two secrets:

```
{  
  "Movies": {  
    "ServiceApiKey": "12345",  
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-  
1;Trusted_Connection=True;MultipleActiveResultSets=true"  
  }  
}
```

从在其中的目录运行以下命令 `.csproj` 文件存在：

```
dotnet user-secrets remove "Movies:ConnectionString"
```

应用程序的 `secrets.json` 已修改文件，以删除与关联的键 / 值对 `MoviesConnectionString` 密钥：

```
{  
  "Movies": {  
    "ServiceApiKey": "12345"  
  }  
}
```

运行 `dotnet user-secrets list` 显示以下消息：

```
Movies:ServiceApiKey = 12345
```

删除所有机密

Assume the app's `secrets.json` file contains the following two secrets:

```
{  
  "Movies": {  
    "ServiceApiKey": "12345",  
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-  
1;Trusted_Connection=True;MultipleActiveResultSets=true"  
  }  
}
```

从在其中的目录运行以下命令 `.csproj` 文件存在：

```
dotnet user-secrets clear
```

已从删除应用程序的所有用户机密 `secrets.json` 文件：

```
{}
```

运行 `dotnet user-secrets list` 显示以下消息：

```
No secrets configured for this application.
```

其他资源

- [ASP.NET Core 中的配置](#)

- 在 ASP.NET 核心的 azure 密钥保管库配置提供程序

在 ASP.NET 核心的 azure 密钥保管库配置提供程序

2018/5/4 • 10 min to read • [Edit Online](#)

通过Luke Latham和Andrew Stanton 护士

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

查看或下载 2.x 的示例代码：

- [基本示例\(如何下载\)](#)-读取到应用的密钥值。
- [密钥名称前缀示例\(如何下载\)](#)-读取机密值使用密钥名称作为前缀表示版本的应用，这允许你加载一组不同的每个应用程序版本的机密值。

本文档说明如何使用[Microsoft Azure 密钥保管库](#)配置提供程序以从 Azure 密钥保管库机密加载应用程序配置值。Azure 密钥保管库是一种基于云的服务，可帮助你保护加密密钥和机密由应用程序和服务。常见方案包括控制对敏感的配置数据的访问和会议的必要条件 FIPS 140-2 级别 2 硬件安全模块 (HSM) 时验证存储配置数据。此功能是可用的应用程序面向 ASP.NET 核心 1.1 或更高版本。

包

若要使用的提供程序，添加到引用[Microsoft.Extensions.Configuration.AzureKeyVault](#)包。

应用程序配置

你可以浏览的提供程序[示例应用](#)。一旦建立密钥保管库，并在保管库中创建机密，该示例应用将安全地加载到其配置的密钥值和在网页中显示它们。

提供程序添加到 `ConfigurationBuilder` 与 `AddAzureKeyVault` 扩展。在示例应用中，则该扩展将从加载的三个配置值`appsettings.json`文件。

应用程序设置	描述	示例
<code>Vault</code>	Azure 密钥保管库名称	contosovault
<code>ClientId</code>	Azure Active Directory 应用程序 Id	627e911e-43cc-61d4-992e-12db9c81b413
<code>ClientSecret</code>	Azure Active Directory 应用程序键	g58K3dtg59o1Pa+e59v2Tx829w6VxTB2yv9sv/101di=

```
config.SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", optional: false)
    .AddEnvironmentVariables();

var builtConfig = config.Build();

config.AddAzureKeyVault(
    $"https://{{builtConfig["Vault"]}}.vault.azure.net/",
    builtConfig["ClientId"],
    builtConfig["ClientSecret"]);
```

创建密钥保管库密码和加载配置值 (basic 示例)

1. 创建密钥保管库并设置应用程序的指南的 Azure Active Directory (Azure AD) [开始使用 Azure 密钥保管库](#)。

- 将机密添加到密钥保管库使用 [AzureRM 密钥保管库 PowerShell 模块](#) 可从 [PowerShell 库](#)、[Azure 密钥保管库 REST API](#), 或[Azure 门户](#)。机密创建为 [手动](#)或[证书](#)机密。证书机密是使用应用程序和服务的证书, 但不是支持配置提供程序。应使用 [手动](#)选项可创建带有配置提供程序使用的名称-值对机密。
 - 创建简单的机密的名称-值对。Azure 密钥保管库密钥名称被限制为字母数字字符和短划线。
 - 分层值 (配置节) 使用 `--` (两个短划线) 作为分隔符, 在此示例。通常用于分隔的子项中的一部分的冒号[ASP.NET 核心配置](#), 机密名称中不允许出现。因此, 两个短划线的使用和机密加载到应用程序的配置时交换冒号。
 - 创建两个手动具有以下名称-值对的机密。第一个密钥是一个简单的名称和值, 并第二个密钥创建一个部分和子项中的密钥名称机密值:
 - `SecretName : secret_value_1`
 - `Section--SecretName : secret_value_2`

- 向 Azure Active Directory 注册示例应用程序。
 - 授权应用程序访问密钥保管库。当你使用 `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet 来授权应用程序访问密钥保管库, 提供 `List` 和 `Get` 访问与机密 `-PermissionsToSecrets list,get`。
2. 更新应用程序的 `appsettings.json` 的值的文件 `Vault`, `ClientId`, 和 `ClientSecret`。
 3. 运行示例应用程序, 它可以通过获取其配置值从 `IConfigurationRoot` 机密的名称与同名。

- 非分层值: 的值 `SecretName` 一起被获取 `config["SecretName"]`。
- 分层值 (节): 使用 `:` (冒号) 表示法或 `GetSection` 扩展方法。使用这些方法之一获取配置值:
 - `config["Section:SecretName"]`
 - `config.GetSection("Section")["SecretName"]`

运行应用程序时, 网页将显示加载的机密值:

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
SecretName	SecretName	<code>Configuration["SecretName"]</code>	<code>secret_value_1</code>
Section:SecretName	Section--SecretName	<code>Configuration["Section:SecretName"]</code>	<code>secret_value_2</code>
		<code>Configuration.GetSection("Section")["SecretName"]</code>	<code>secret_value_2</code>

创建带前缀的密钥保管库密码和加载配置值 (密钥的名称的前缀的示例)

`AddAzureKeyVault` 此外提供了一个接受的实现重载 `IKeyVaultSecretManager`, 这样, 你可以控制如何密钥保管库密码将转换为配置密钥。例如, 你可以实现接口后, 可加载基于你在应用启动时提供的前缀值的密钥值。这使你, 例如, 加载基于应用程序的版本的机密。

警告

不要使用前缀在密钥保管库密码放到同一个密钥保管库的多个应用的机密, 或将环境机密 (例如, 开发与生产机密) 入同一保管库。我们建议, 不同的应用程序和开发/生产环境使用单独的密钥保管库来隔离应用程序环境安全的最高级别。

使用第二个示例应用程序, 为密钥保管库中创建一个机密 `5000-AppSecret` (密钥保管库密钥名称中不允许句点)

表示你的应用程序的版本为 5.0.0.0 应用程序密钥。有关另一个版本, 5.1.0.0, 创建密钥 5100-AppSecret。每个应用程序版本将自己机密的值加载到作为其配置 AppSecret、关闭版本剥离加载时为它的机密。示例的实现如下所示:

```
// The appVersion obtains the app version (5.0.0.0), which
// is set in the project file and obtained from the entry
// assembly. The versionPrefix holds the version without
// dot notation for the PrefixKeyVaultSecretManager.
var appVersion = Assembly.GetEntryAssembly().GetName().Version.ToString();
var versionPrefix = appVersion.Replace(".", string.Empty);

config.AddAzureKeyVault(
    $"https://{builtConfig["Vault"]}.vault.azure.net/",
    builtConfig["ClientId"],
    builtConfig["ClientSecret"],
    new PrefixKeyVaultSecretManager(versionPrefix));
```

```
public class PrefixKeyVaultSecretManager : IKeyVaultSecretManager
{
    private readonly string _prefix;

    public PrefixKeyVaultSecretManager(string prefix)
    {
        _prefix = $"{prefix}--";
    }

    public bool Load(SecretItem secret)
    {
        // Load a vault secret when its secret name starts with the
        // prefix. Other secrets won't be loaded.
        return secret.Identifier.Name.StartsWith(_prefix);
    }

    public string GetKey(SecretBundle secret)
    {
        // Remove the prefix from the secret name and replace two
        // dashes in any name with the KeyDelimiter, which is the
        // delimiter used in configuration (usually a colon). Azure
        // Key Vault doesn't allow a colon in secret names.
        return secret.SecretIdentifier.Name
            .Substring(_prefix.Length)
            .Replace("--", ConfigurationPath.KeyDelimiter);
    }
}
```

Load 方法调用通过循环访问保管库密钥来查找那些具有版本前缀的提供程序算法。当版本前缀找到具有 Load，该算法使用 GetKey 方法返回的配置名称的机密的名称。它剥离的机密的名称从版本前缀，并返回到应用程序的配置名称-值对的加载的机密名称的其余部分。

当您实现此方法：

1. 密钥保管库密码进行加载。
2. 字符串密钥 5000-AppSecret 匹配。
3. 版本， 5000 (替换为短划线)，也会丢失从离开的键名称中移出 AppSecret 包含机密的值加载到应用程序的配置。

注意

你还可以提供你自己 `KeyVaultClient` 实现 `AddAzureKeyVault`。提供自定义客户端，可共享的配置提供程序和你的应用程序的其他部件之间的单个实例。

1. 创建密钥保管库并设置应用程序的指南的 Azure Active Directory (Azure AD) [开始使用 Azure 密钥保管库](#)。
 - 将机密添加到密钥保管库使用 [AzureRM 密钥保管库 PowerShell 模块](#) 可从 [PowerShell 库](#)、[Azure 密钥保管库 REST API](#), 或[Azure 门户](#)。机密创建为手动或证书机密。证书机密是使用应用程序和服务的证书, 但不是支持配置提供程序。应使用 [手动](#)选项可创建带有配置提供程序使用的名称-值对机密。
 - 分层值 (配置节) 使用 `--` (两个短划线) 作为分隔符。
 - 创建两个 [手动](#)具有以下名称-值对的机密：
 - `5000-AppSecret : 5.0.0.0_secret_value`
 - `5100-AppSecret : 5.1.0.0_secret_value`
 - 向 Azure Active Directory 注册示例应用程序。
 - 授权应用程序访问密钥保管库。当你使用 `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet 来授权应用程序访问密钥保管库, 提供 `List` 和 `Get` 访问与机密 `-PermissionsToSecrets list,get`。
2. 更新应用程序的 `appsettings.json` 的值的文件 `Vault`, `ClientId`, 和 `ClientSecret`。
3. 运行示例应用程序, 它可以通过获取其配置值从 `IConfigurationRoot` 带前缀的机密名称与同名。在此示例中, 该前缀是应用程序的版本, 它提供给 `PrefixKeyVaultSecretManager` 添加 Azure 密钥保管库配置提供程序时。值 `AppSecret` 一起被获取 `config["AppSecret"]`。该应用程序生成的网页显示加载的值：

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	5000-AppSecret	<code>Configuration["AppSecret"]</code>	<code>5.0.0.0_secret_value</code>

4. 更改中的项目文件中的应用程序集版本 `5.0.0.0` 到 `5.1.0.0` 并再次运行该应用。此时, 返回的机密值是 `5.1.0.0_secret_value`。该应用程序生成的网页显示加载的值：

Key Vault Configuration Provider Sample

Secret	Name in Key Vault	Obtained from Configuration	Value
AppSecret	5100-AppSecret	<code>Configuration["AppSecret"]</code>	<code>5.1.0.0_secret_value</code>

控制对 ClientSecret 的访问

使用 [机密管理器工具](#)维护 `ClientSecret` 外部项目源树。使用密钥管理器中, 你将应用程序机密关联与特定项目并共享跨多个项目。

在开发时支持证书的环境中的.NET Framework 应用, 可以使用 X.509 证书验证到 Azure 密钥保管库。X.509 证书的私钥由操作系统管理。有关详细信息, 请参阅[使用而不是客户端密钥证书的身份验证](#)。使用 `AddAzureKeyVault` 接受的重载 `X509Certificate2`。

```
var store = new X509Store(StoreLocation.CurrentUser);
store.Open(OpenFlags.ReadOnly);
var cert = store.Certificates.Find(X509FindType.FindByThumbprint, config["CertificateThumbprint"], false);

builder.AddAzureKeyVault(
    config["Vault"],
    config["ClientId"],
    cert.OfType<X509Certificate2>().Single(),
    new EnvironmentSecretManager(env.ApplicationName));
store.Close();

Configuration = builder.Build();
```

重新加载机密

机密缓存直到 `IConfigurationRoot.Reload()` 调用。过期、禁用，并且应用程序之前未遵从密钥保管库中的更新的机密 `Reload` 执行。

```
Configuration.Reload();
```

禁用和过期机密

禁用和过期机密引发 `KeyVaultClientException`。若要防止你的应用引发，替换你的应用程序或更新的过期已禁用/机密。

疑难解答

当应用程序失败时加载使用提供程序的配置时，将一条错误消息写入到[ASP.NET 日志记录基础结构](#)。以下条件将阻止从加载的配置：

- 应用程序未正确配置 Azure Active Directory 中。
- Azure 密钥保管库中不存在密钥保管库。
- 应用程序未授权访问密钥保管库。
- 访问策略不包括 `Get` 和 `List` 权限。
- 在密钥保管库，配置数据（名称-值对）被命名不正确、缺少、禁用，或已过期。
- 应用程序出现错误的密钥保管库名称 (`Vault`)，Azure AD 应用程序 Id (`ClientId`)，或 Azure AD 密钥 (`ClientSecret`)。
- Azure AD 密钥 (`ClientSecret`) 已过期。
- 配置密钥（名称）不正确的应用中的你正在试图加载的值。

其他资源

- [配置](#)
- [Microsoft Azure: 密钥保管库](#)
- [Microsoft Azure: 密钥保管库文档](#)
- [如何生成和传输受 HSM 保护密钥的 Azure 密钥保管库](#)
- [KeyVaultClient 类](#)

在 ASP.NET Core 防止跨站点请求伪造 (XSRF/CSRF) 攻击

2018/4/10 • 16 min to read • [Edit Online](#)

通过Steve Smith, Fiyaz Hasan, 和Rick Anderson

跨站点请求伪造 (也称为 XSRF 或 CSRF, 发音*, 请参阅冲浪*) 是针对恶意网站凭此可以影响客户端浏览器和信任, 一个 web 应用程序之间的交互的 web 承载的应用程序的攻击浏览器。这些攻击是可能的因为 web 浏览器将自动与每个请求某些类型的身份验证令牌发送到网站。这种形式的攻击也称为是一键式攻击或会话乘坐因为攻击充分利用用户的先前进行身份验证会话。

CSRF 攻击的示例:

1. 用户登录到 `www.good-banking-site.com` 使用窗体身份验证。服务器对用户进行身份验证, 并发出包含身份验证 cookie 的响应。该站点处于易受到攻击, 因为它信任的任何请求都收到与有效的身份验证 cookie。
2. 用户访问恶意站点, `www.bad-crook-site.com`。

恶意站点, `www.bad-crook-site.com`, 包含类似于以下的 HTML 窗体:

```
<h1>Congratulations! You're a Winner!</h1>
<form action="http://good-banking-site.com/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

请注意, 窗体的 `action` 文章到易受攻击的站点上, 而不是恶意的站点。这是 CSRF 的"跨站点"部分。

3. 用户选择提交按钮。浏览器发出请求, 并自动将请求的域中, 身份验证 cookie `www.good-banking-site.com`。
4. 在上运行的请求 `www.good-banking-site.com` 与用户的身份验证上下文的服务器, 并且可以执行允许经过身份验证的用户执行任何操作。

当用户选择按钮以提交表单时, 将无法恶意站点:

- 运行自动提交该表单的脚本。
- 将窗体提交作为 AJAX 请求中发送。
- 使用 CSS 的隐藏的表单。

使用 HTTPS, 则不会阻止 CSRF 攻击。恶意站点可以将发送 `https://www.good-banking-site.com/` 请求一样轻松它可发送的不安全的请求。

一些攻击目标响应 GET 请求的终结点, 在这种情况下使用的图像标记要执行的操作。允许映像但阻止 JavaScript 的论坛站点上, 这种攻击十分常见。应用程序更改的状态 GET 请求中, 在其中修改变量或资源, 就很容易遭受恶意攻击。更改状态的 GET 请求是不安全的。最佳做法是永远不会更改在 GET 请求的状态。

针对 web 应用可用于身份验证的 cookie, 因为可能会出现 CSRF 攻击:

- 浏览器存储颁发的 web 应用的 cookie。
- 存储的 cookie 包括用于身份验证的用户的会话 cookie。
- 浏览器发送的所有 cookie 与域关联到 web 应用程序而不考虑如何向应用程序请求生成浏览器中的每个请求。

但是, CSRF 攻击不局限于利用 cookie。例如, 基本和摘要式身份验证也是易受攻击的。浏览器使用基本或摘要式身份验证的用户登录后, 直到会话才会自动发送凭据[†]结束。

[†]在此上下文中, 会话指的是在此期间用户进行身份验证的客户端会话。它是与服务器端会话无关或 ASP.NET 核心会话中间件。

用户可以通过采取预防措施来防止 CSRF 漏洞:

- 从 web 应用程序在完成后使用这些签名。
- 定期清除浏览器 cookie。

但是, CSRF 漏洞基本上是 web 应用, 而不是最终用户有问题。

身份验证基础知识

基于 cookie 的身份验证是一种身份验证的常用形式。基于令牌的身份验证系统中受欢迎程度, 特别是对于单页面应用程序 (Spa) 增长。

基于 cookie 的身份验证

当用户身份验证使用其用户名和密码时, 它们被颁发一个令牌, 包含可以用于身份验证和授权的身份验证票证。随着工作的附带的每个请求客户端的 cookie 的令牌存储。生成和验证此 cookie 的 Cookie 身份验证中间件执行。中间件的加密 cookie 序列化为一个用户主体。在后续请求, 该中间件将验证 cookie, 重新创建主体, 并将分配到主体用户属性 `HttpContext`。

基于令牌的身份验证

当用户进行身份验证时, 它们被颁发一个令牌 (不 antiforgery 令牌)。令牌包含用户信息的形式声明或指向维护应用程序中的用户状态的应用程序的引用令牌。当用户尝试访问要求进行身份验证的资源时, 令牌将发送到使用的其他授权标头中的持有者令牌的窗体应用程序。这使得应用程序无状态。在每个后续请求中, 令牌请求中传递进行服务器端验证。此令牌不加密; 它具有编码。在服务器上, 将解码令牌来访问其信息。若要在后续请求中发送令牌, 请在浏览器的本地存储中存储令牌。如果该令牌存储在浏览器的本地存储, 则不会关心 CSRF 漏洞。CSRF 是一个问题时的令牌存储在一个 cookie。

在一个域托管的多个应用程序

共享宿主环境包括易受到会话劫持、登录 CSRF 和其他的攻击。

尽管 `example1.contoso.net` 和 `example2.contoso.net` 是不同的主机下的主机之间没有隐式信任关系 `*.contoso.net` 域。此隐式信任关系允许影响对方的 cookie (控制 AJAX 请求的同源策略不一定适用于 HTTP cookie) 可能不受信任的主机。

可以通过不能共享域防止利用在同一个域上托管的应用程序之间的受信任的 cookie 的攻击。当每个应用程序托管在自身域中时, 没有任何隐式 cookie 信任关系, 以便利用。

ASP.NET 核心 antiforgery 配置

警告

ASP.NET 核心实现 antiforgery 使用 [ASP.NET 核心数据保护](#)。数据保护堆栈必须配置为在服务器场中正常工作。请参阅 [配置数据保护](#) 有关详细信息。

在 ASP.NET 核心 2.0 或更高版本，[FormTagHelper](#) antiforgery 令牌注入 HTML 窗体元素。Razor 文件中的以下标记将自动生成 antiforgery 令牌：

```
<form method="post">
  ...
</form>
```

类似地，[IHtmlHelper.BeginForm](#) antiforgery 令牌生成默认情况下，如果窗体的方法不 GET。

自动生成的 antiforgery 令牌 HTML 窗体元素发生时 `<form>` 标记包含 `method="post"` 属性和以下任一条件：

- 操作属性为空 (`action=""`)。
- 操作属性不提供 (`<form method="post">`)。

可以禁用自动生成的 antiforgery 令牌 HTML 窗体元素：

- 显式禁用 antiforgery 令牌 `asp-antiforgery` 属性：

```
<form method="post" asp-antiforgery="false">
  ...
</form>
```

- Form 元素是已选择扩展的标记帮助程序通过使用标记帮助器！[选择退出符号](#)：

```
<!form method="post">
  ...
<!/form>
```

- 删除 `FormTagHelper` 从视图。`FormTagHelper` 可以通过将以下指令添加到 Razor 视图从视图中删除：

```
@removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
Microsoft.AspNetCore.Mvc.TagHelpers
```

注意

Razor 页会自动防范 CSRF。有关详细信息，请参阅[XSRF/CSRF 和 Razor 页](#)。

防御 CSRF 攻击的最常见方法是使用同步器令牌模式(STP)。在用户请求具有窗体数据的页时，使用 STP：

1. 服务器将发送到客户端的当前用户的标识与关联的令牌。
2. 客户端返回将令牌发送到服务器以进行验证。
3. 如果服务器收到与经过身份验证的用户的标识不匹配的令牌，而拒绝该请求。

该令牌的唯一且不可预测。此外可以使用令牌以确保正确地执行序列化的一系列请求 (例如，确保请求序列的：第 1 页-页上 2-第 3 页)。ASP.NET 核心 MVC 和 Razor 页模板中的窗体的所有生成 antiforgery 令牌。以下两个视图的示例生成 antiforgery 令牌：

```

<form asp-controller="Manage" asp-action="ChangePassword" method="post">
    ...
</form>

@using (Html.BeginForm("ChangePassword", "Manage"))
{
    ...
}

```

显式添加到 antiforgery 令牌 `<form>` 没有标记帮助程序使用的 HTML 帮助程序元素 `@Html.AntiForgeryToken` :

```

<form action="/" method="post">
    @Html.AntiForgeryToken()
</form>

```

在每个前面的情况下，ASP.NET Core 添加类似于以下一个隐藏的表单字段：

```

<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">

```

ASP.NET 核心包括三个筛选器来处理 antiforgery 令牌：

- `ValidateAntiForgeryToken`
- `AutoValidateAntiforgeryToken`
- `IgnoreAntiforgeryToken`

Antiforgery 选项

自定义 `antiforgery 选项` 中 `Startup.ConfigureServices` :

```

services.AddAntiforgery(options =>
{
    options.CookieDomain = "contoso.com";
    options.CookieName = "X-CSRF-TOKEN-COOKIENAME";
    options.CookiePath = "Path";
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.RequireSsl = false;
    options.SuppressXFrameOptionsHeader = false;
});

```

选项	描述
<code>Cookie</code>	确定用于创建 antiforgery cookie 的设置。
<code>CookieDomain</code>	Cookie 的域。默认为 <code>null</code> 。此属性已过时，并在未来版本中将删除。建议的替代项是 <code>Cookie.Domain</code> 。
<code>CookieName</code>	Cookie 的名称。如果未设置，则系统会生成一个唯一的名称开头 <code>DefaultCookiePrefix ("</code> <code>AspNetCore.Antiforgery.")</code> 下。此属性已过时，并在未来版本中将删除。建议的替代项是 <code>Cookie.Name</code> 。
<code>CookiePath</code>	在 cookie 上设置的路径。此属性已过时，并在未来版本中将删除。建议的替代项是 <code>Cookie.Path</code> 。

选项	描述
FormFieldName	Antiforgery 系统用于呈现 antiforgery 令牌在视图中的隐藏的表单字段的名称。
HeaderName	Antiforgery 系统使用的标头的名称。如果 <code>null</code> ，系统会考虑仅窗体数据。
RequireSsl	指定是否由 antiforgery 系统需要 SSL。如果 <code>true</code> ，非 SSL 请求将失败。默认为 <code>false</code> 。此属性已过时，并在未来版本中将删除。建议的替代项是设置 <code>Cookie.SecurePolicy</code> 。
SuppressXFrameOptionsHeader	指定是否禁止生成 <code>X-Frame-Options</code> 标头。默认情况下，值为 "SAMEORIGIN" 生成标头。默认为 <code>false</code> 。

有关详细信息，请参阅[CookieAuthenticationOptions](#)。

使用 `IAntiforgery` 配置 antiforgery 功能

`IAntiforgery` 提供 API 来配置 antiforgery 功能。`IAntiforgery` 可以在请求 `Configure` 方法 `Startup` 类。下面的示例使用从应用程序的主页上的中间件生成 antiforgery 令牌并将其发送响应中将其作为 cookie (使用在本主题后面所述的默认角度命名约定)：

```
public void Configure(IApplicationBuilder app, IAntiforgery antiforgery)
{
    app.Use(next => context =>
    {
        string path = context.Request.Path.Value;

        if (
            string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))
        {
            // The request token can be sent as a JavaScript-readable cookie,
            // and Angular uses it by default.
            var tokens = antiforgery.GetAndStoreTokens(context);
            context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
                new CookieOptions() { HttpOnly = false });
        }

        return next(context);
    });
}
```

要求 antiforgery 验证

`ValidateAntiForgeryToken` 是操作筛选器，可以应用于单个操作、一个控制器或全局范围内。除非请求包含有效的 antiforgery 令牌，将阻止对已应用此筛选器的操作发出的请求。

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account)
{
    ManageMessageId? message = ManageMessageId.Error;
    var user = await GetCurrentUserAsync();

    if (user != null)
    {
        var result =
            await _userManager.RemoveLoginAsync(
                user, account.LoginProvider, account.ProviderKey);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            message = ManageMessageId.RemoveLoginSuccess;
        }
    }

    return RedirectToAction(nameof(ManageLogins), new { Message = message });
}

```

`ValidateAntiForgeryToken` 属性修饰，包括 HTTP GET 请求对操作方法的请求要求令牌。如果 `ValidateAntiForgeryToken` 属性应用跨应用程序的控制器，则可以重写与 `IgnoreAntiforgeryToken` 属性。

注意

ASP.NET Core 不支持自动将 antiforgery 令牌添加到 GET 请求。

自动验证 antiforgery 令牌仅不安全的 HTTP 方法

ASP.NET Core 应用不生成 antiforgery 令牌进行安全的 HTTP 方法 (GET、HEAD、选项和跟踪)。而不是广泛应用 `ValidateAntiForgeryToken` 属性，然后重写它与 `IgnoreAntiforgeryToken` 特性，`AutoValidateAntiforgeryToken` 可以使用属性。此属性适用类似 `ValidateAntiForgeryToken` 特性，只不过它不需要使用以下的 HTTP 方法发出的请求令牌：

- GET
- HEAD
- 选项
- TRACE

我们建议使用 `AutoValidateAntiforgeryToken` 广泛的非 API 方案。这可确保默认情况下保护 POST 操作。替代项是默认情况下，将忽略 antiforgery 令牌除非 `ValidateAntiForgeryToken` 应用于单个操作方法。它更有可能在此方案中保留的 POST 操作方法不受保护错误地使应用程序容易受到 CSRF 攻击。所有文章应都发送 antiforgery 令牌。

API 没有一种用于发送令牌的非 cookie 一部分的自动机制。实现可能取决于客户端代码实现。一些示例如下所示：

类级别示例：

```

[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{

```

全局示例：

```
services.AddMvc(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

替代全局或控制器 antiforgery 属性

[IgnoreAntiforgeryToken](#) 筛选器用于消除对给定操作（或控制器）的 antiforgery 令牌的需要。当应用时，此筛选器将重写 `ValidateAntiForgeryToken` 和 `AutoValidateAntiforgeryToken`（全局或在控制器上）在高级别指定筛选器。

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
    [HttpPost]
    [IgnoreAntiforgeryToken]
    public async Task<IActionResult> DoSomethingSafe(SomeViewModel model)
    {
        // no antiforgery token required
    }
}
```

身份验证后刷新令牌

用户进行身份验证通过将用户重定向到一个视图或 Razor 页页后，应刷新令牌。

JavaScript、AJAX 和 Spa

在传统的基于 HTML 的应用，antiforgery 令牌将传递到使用隐藏的表单域的服务器。在基于 JavaScript 的现代应用和 Spa 中，以编程方式进行多请求。这些 AJAX 请求可能使用其他方法（如请求标头或 cookie）将该令牌发送。

如果使用 cookie 来存储身份验证令牌，并在服务器上的 API 请求进行身份验证，CSRF 是一个潜在的问题。如果本地存储用于存储令牌，因为从本地存储的值不会自动发送到每个请求的服务器可能会缓解 CSRF 漏洞。因此，使用本地存储来存储客户端和发送令牌，因为请求标头是建议的方法上的 antiforgery 令牌。

JavaScript

使用视图支持 JavaScript，令牌可以创建使用从视图中的服务。注入 [Microsoft.AspNetCore.Antiforgery.IAntiforgery](#) 到视图并调用服务 [GetAndStoreTokens](#)：

```

@{
    ViewData["Title"] = "AJAX Demo";
}
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}

<input type="hidden" id="RequestVerificationToken"
       name="RequestVerificationToken" value="@GetAntiXsrfRequestToken()">

<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <p><input type="button" id="antiforgery" value="Antiforgery"></p>
    <script>
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
            if (xhttp.readyState == XMLHttpRequest.DONE) {
                if (xhttp.status == 200) {
                    alert(xhttp.responseText);
                } else {
                    alert('There was an error processing the AJAX request.');
                }
            }
        };
        document.addEventListener('DOMContentLoaded', function() {
            document.getElementById("antiforgery").onclick = function () {
                xhttp.open('POST', '@Url.Action("Antiforgery", "Home")', true);
                xhttp.setRequestHeader("RequestVerificationToken",
                    document.getElementById('RequestVerificationToken').value);
                xhttp.send();
            }
        });
    </script>
</div>

```

此方法不需要直接处理从服务器设置 cookie 或从客户端读取它们。

前面的示例使用 JavaScript 的 AJAX POST 标头中读取的隐藏的字段值。

JavaScript 还可以访问在 cookie 令牌并使用 cookie 的内容与令牌的值创建的标头。

```

context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken,
    new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });

```

假设脚本请求将该令牌发送调用标头中 `x-CSRF-TOKEN`，配置 antiforgery 服务以查找 `x-CSRF-TOKEN` 标头：

```

services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");

```

下面的示例使用 JavaScript 进行了相应的标头使用的 AJAX 请求：

```

function getCookie(cname) {
    var name = cname + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';");
    for(var i = 0; i <ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0) {
            return c.substring(name.length, c.length);
        }
    }
    return "";
}

var csrfToken = getCookie("CSRF-TOKEN");

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState == XMLHttpRequest.DONE) {
        if (xhttp.status == 200) {
            alert(xhttp.responseText);
        } else {
            alert('There was an error processing the AJAX request.');
        }
    }
};
xhttp.open('POST', '/api/password/changepassword', true);
xhttp.setRequestHeader("Content-type", "application/json");
xhttp.setRequestHeader("X-CSRF-TOKEN", csrfToken);
xhttp.send(JSON.stringify({ "newPassword": "ReallySecurePassword999$$$$" }));

```

AngularJS

AngularJS 使用到地址 CSRF 的约定。如果服务器发送具有该名称的 cookie `XSRF-TOKEN`，AngularJS `$http` 服务将 cookie 值添加到标头时它将请求发送到服务器。此过程是自动进行。标头不需要显式设置。标头名称是 `X-XSRF-TOKEN`。服务器应检测此标头，并验证其内容。

有关 ASP.NET 核心 API 使用此约定：

- 配置你的应用程序提供在调用 cookie 令牌 `XSRF-TOKEN`。
- 将查找名为的标头 antiforgery 服务配置为 `X-XSRF-TOKEN`。

```
services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
```

[查看或下载示例代码 \(如何下载\)](#)

扩展 antiforgery

`IAntiForgeryTokenAdditionalDataProvider` 类型允许开发人员通过往返中每个令牌的其他数据扩展的反 CSRF 系统行为。`GetAdditionalData` 每次调用方法生成的字段标记，和在生成的标记内嵌入的返回值。实施者无法返回时间戳、nonce 或任何其他值，然后调用 `ValidateAdditionalData` 时验证令牌验证此数据。客户端的用户名已嵌入在生成的令牌中，因此无需包括此信息。如果令牌包括补充数据但不是 `IAntiForgeryTokenAdditionalDataProvider` 是配置，不验证补充数据。

其他资源

- [CSRF 上打开 Web 应用程序安全项目 \(OWASP\)](#)。

防止在 ASP.NET 核心中的打开重定向攻击

2018/5/14 • 3 min to read • [Edit Online](#)

Web 应用程序将重定向到通过如查询字符串或窗体数据请求指定的 URL 可能可能被篡改将用户重定向到外部、恶意 URL。此篡改称为打开重定向攻击。

每当应用程序逻辑将重定向到指定的 URL，你必须验证重定向 URL 尚未被篡改。ASP.NET 核心具有内置功能来防止应用打开重定向（也称为打开重定向）攻击。

打开重定向攻击是什么？

Web 应用程序频繁地将用户重定向到登录页访问要求进行身份验证的资源时。重定向 typically 包括 `returnUrl` 查询字符串参数，以便用户可以在用户成功登录后返回最初请求的 url。用户进行身份验证后，它们在重定向到他们最初具有请求的 URL。

因为请求的查询字符串中指定的目标 URL，则恶意用户可能篡改查询字符串。篡改过的查询字符串可能导致要将用户重定向到外部、恶意站点的站点。这种技术称为打开重定向（或重定向）攻击。

示例攻击

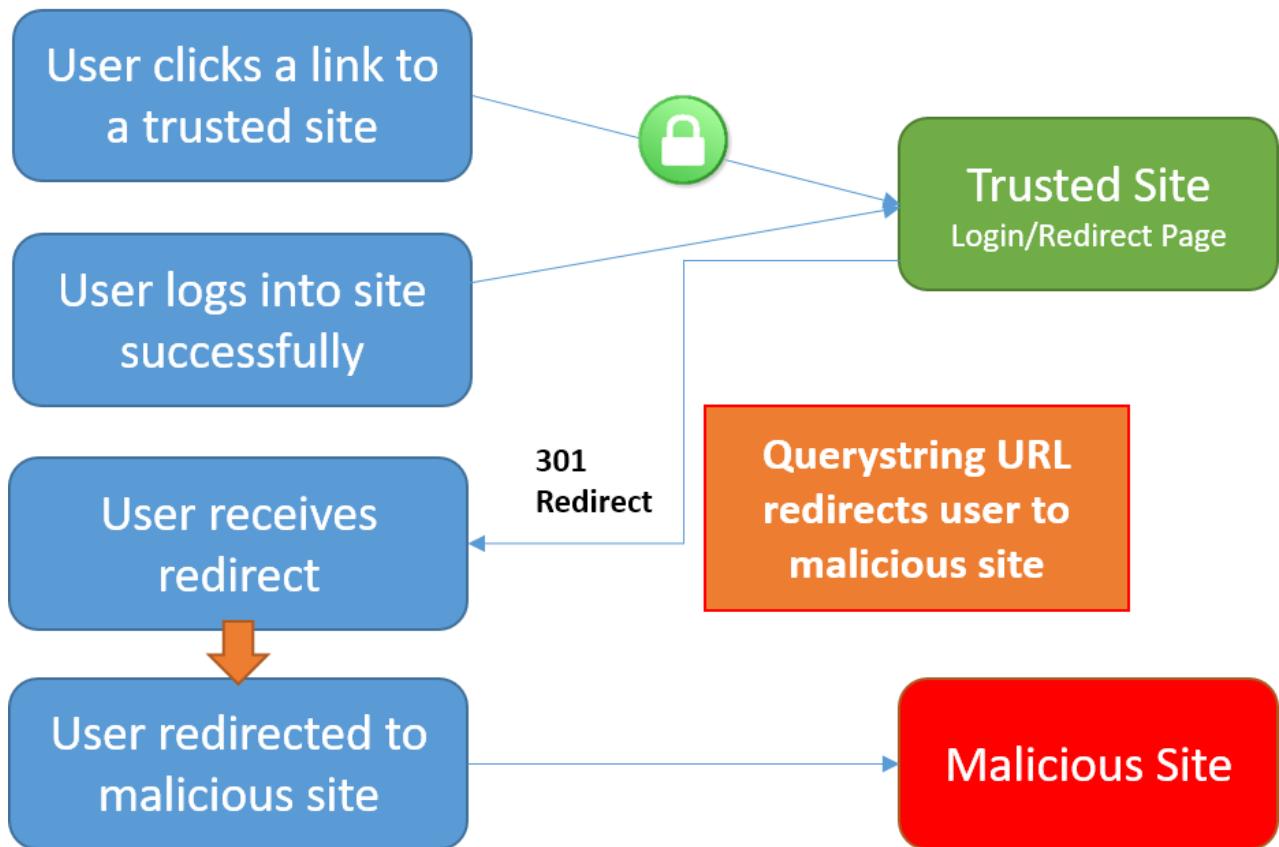
恶意用户可以开发用于允许对用户的凭据或您的应用程序上的敏感信息的恶意用户访问的攻击。若要开始攻击，它们使用户链接到网站的登录页，单击与 `returnUrl` 添加到的 URL 查询字符串值。例如，[NerdDinner.com](#)（编写 ASP.NET MVC 的）的示例应用程序包括此处这样的登录页：

`http://nerddinner.com/Account/LogOn?returnUrl=/Home/About`。这种攻击然后执行下列步骤：

1. 用户单击的链接 `http://nerddinner.com/Account/LogOn?returnUrl=http://nerddiner.com/Account/LogOn`（请注意，第二个 URL 是 `nerddiner`, 不 `nerddinner`）。
2. 用户成功登录。
3. 用户（通过站点）重定向到 `http://nerddiner.com/Account/LogOn`（如下所示的真正的站点的恶意站点）。
4. 用户再次登录（提供恶意站点其凭据）和重定向回真正的站点。

用户将可能认为其第一次尝试登录失败，并且其第二个已成功。它们将很可能仍然无法察觉其凭据已泄漏。

Open Redirection Attack Process



除了登录页的某些站点提供重定向页或终结点。假设你的应用程序的页的打开的重定向，`/Home/Redirect`。攻击者可以创建，例如，将转到一封电子邮件中的链接

`[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`。典型的用户将看一看 URL，并请参阅开始使用你的站点名称。信任的用户将单击此链接。打开重定向将为网络钓鱼站点，这看起来相同地区，然后发送用户和用户很可能会登录到他们认为是你的站点。

针对打开重定向攻击提供保护

当开发 web 应用程序，会将所有用户提供数据视为不受信任。如果你的应用程序具有基于 URL 的内容将用户重定向的功能，请确保，此类重定向仅这样的是本地应用程序中（或已知的 URL，没有任何可能在查询字符串中提供的 URL）。

LocalRedirect

使用 `LocalRedirect` 帮助器方法的基本 `Controller` 类：

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` 如果指定非本地 URL，将引发异常。否则，其行为就像 `Redirect` 方法。

IsLocalUrl

使用 `IsLocalUrl` 方法以测试 Url，然后重定向：

下面的示例演示如何检查 URL 重定向之前是本地。

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

`IsLocalUrl` 方法可防止用户无意中重定向到恶意的站点。你可以登录时需要本地 URL 的位置的情况下提供非本地 URL 提供的 URL 的详细信息。日志记录重定向 Url 可用于诊断重定向攻击。

防止跨站点脚本 (XSS) 在 ASP.NET 核心

2018/4/10 • 8 min to read • [Edit Online](#)

作者: [Rick Anderson](#)

跨站点脚本 (XSS) 是一个安全漏洞，这会使攻击者将客户端脚本（通常是 JavaScript）放入网页。当其他用户加载受影响的页的攻击者脚本将运行时，从而使攻击者窃取 cookie 和会话令牌更改通过 DOM 操作网页的内容或将浏览器重定向到另一页。当应用程序接受用户输入，并输出它在页中而不验证、编码或转义它时，通常会发生 XSS 漏洞。

保护 XSS 针对应用程序

在通过诱使你的应用程序插入到工作原理的基本级别 XSS `<script>` 标记到呈现页中，或通过插入 `on*` 插入元素的事件。开发人员应使用以下的预防步骤以避免 XSS 引入其应用程序。

1. 永远不会，将不受信任的数据放入你的 HTML 输入中，除非按照下面的步骤的其余部分。不受信任的数据是可能由攻击者、HTML 窗体输入、查询字符串、HTTP 标头，甚至数据源从数据库中，攻击者可能能够破坏您的数据库，即使它们不能违反你的应用程序控制任何数据。
2. 将 HTML 元素中的不受信任的数据之前请确保它为 HTML 编码。HTML 编码采取字符如`<`和会变成安全的形式，例如`<`。
3. 将不受信任的数据放入 HTML 特性之前请确保它是 HTML 编码的属性。HTML 特性编码为 HTML 编码的超集，并将其他字符编码如`"`和`=`。
4. 将不受信任的数据放入 JavaScript 之前将数据放在一个 HTML 元素在运行时检索其内容。如果这不是可能的则确保数据被编码 JavaScript。JavaScript 编码 `javascript` 采用危险的字符并将其替换为其十六进制，例如`<`将编码为 `\u003c`。
5. 将不受信任的数据放入一个 URL 查询字符串之前确保它是编码的 URL。

使用 Razor 的 HTML 编码

自动使用 MVC Razor 引擎将所有编码输出源自变量，除非您真正努力工作以避免其执行此操作。它使用编码规则，每当你使用的 HTML 特性`*@*`指令。为 HTML 属性编码为 HTML 编码，这意味着你不必考虑自己是否应使用 HTML 编码或 HTML 特性编码的超集。你必须确保您仅使用在 HTML 上下文中，不是在尝试将直接插入 JavaScript 不受信任的输入时。标记帮助程序还会编码标记参数中使用的输入。

考虑以下 Razor 视图：

```
@{  
    var untrustedInput = "<\\"123\\>";  
}  
  
@untrustedInput
```

此视图输出的内容`untrustedInput`变量。此变量包含在 XSS 攻击中，即使用某些字符`<`，`"`和`>`。查看源显示呈现的输出编码为：

```
&lt;&quot;123&quot;&gt;
```

警告

ASP.NET 核心 MVC 提供 `HtmlString` 在输出时不自动编码的类。这应永远不会用于与不受信任的输入结合使用这会将公开 XSS 漏洞。

使用 Razor Javascript 编码

可能有些时候你想要插入处理在视图中的 JavaScript 值。有两种方法可以实现此目的。插入简单值的最安全方法是将值放在一个标记的数据属性并检索你在 JavaScript。例如：

```
@{
    var untrustedInput = "<\"123\">";
}

<div
    id="injectedData"
    data-untrustedinput="@untrustedInput" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    document.write(clientSideUntrustedInputOldStyle);
    document.write("<br />")
    document.write(clientSideUntrustedInputHtml5);
</script>
```

这将生成以下 HTML

```
<div
    id="injectedData"
    data-untrustedinput="<"&quot;123&quot;>" />

<script>
    var injectedData = document.getElementById("injectedData");

    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    document.write(clientSideUntrustedInputOldStyle);
    document.write("<br />")
    document.write(clientSideUntrustedInputHtml5);
</script>
```

其中，运行时，将呈现以下操作；

```
<"123">
<"123">
```

你还可以直接调用 JavaScript 编码器

```
@using System.Text.Encodings.Web;
@inject JavaScriptEncoder encoder;

 @{
     var untrustedInput = "<\"123\">";
 }

<script>
    document.write("@encoder.Encode(untrustedInput)");
</script>
```

这将呈现在浏览器中，如下所示：

```
<script>
    document.write("\u003C\u0022123\u0022\u003E");
</script>
```

警告

不连接不受信任的输入，在 JavaScript 中创建 DOM 元素。应使用 `createElement()` 并将属性值分配适当如 `node.textContent=`，或使用 `element.setAttribute()` / `element[attribute]=` 否则向基于 DOM 的 XSS 公开自己。

访问代码中的编码器

HTML、JavaScript 和 URL 编码器可供代码使用两种方式，可以将它们通过注入依赖关系注入或者你可以使用中包含的默认编码器 `System.Text.Encodings.Web` 命名空间。如果你使用的默认编码器，则你应用到任何要被视为为安全的字符范围不会生效-默认编码器使用的可能的安全编码规则。

若要使用可配置编码器通过你的构造函数应采用的 DI `HtmlEncoder`, `JavaScriptEncoder` 和 `UrlEncoder` 作为适当的参数。例如，

```
public class HomeController : Controller
{
    HtmlEncoder _htmlEncoder;
    JavaScriptEncoder _javascriptEncoder;
    UrlEncoder _urlEncoder;

    public HomeController(HtmlEncoder htmlEncoder,
                          JavaScriptEncoder javascriptEncoder,
                          UrlEncoder urlEncoder)
    {
        _htmlEncoder = htmlEncoder;
        _javascriptEncoder = javascriptEncoder;
        _urlEncoder = urlEncoder;
    }
}
```

编码的 URL 参数

如果你想要构建 URL 查询字符串以不受信任的输入作为值使用 `UrlEncoder` 值进行编码。例如，应用于对象的

```
var example = "\"Quoted Value with spaces and &\"";
var encodedValue = _urlEncoder.Encode(example);
```

编码 encodedValue 后变量将包含 %22Quoted%20Value%20with%20spaces%20and%20%26%22。空格、引号、标点和其他不安全字符百分比编码为其十六进制值，例如空格字符将成为 %20。

警告

不要使用不受信任的输入的 URL 路径的一部分。始终将不受信任的输入传递作为查询字符串值。

自定义编码器

默认情况下编码器使用安全列表限制为基本拉丁 Unicode 范围并为其等效的字符代码在该范围之外的所有字符进行都编码。此行为还会影响 Razor TagHelper 和 HtmlHelper 呈现，因为它将使用编码器输出字符串。

这背后的原因是为了防止未知或未来的浏览器 bug（以前的浏览器 bug 具有触发向上分析基于非英语字符的处理）。如果您的网站进行大量使用非拉丁字符，例如中文、西里尔文或其他人这是可能不希望的行为。

你可以自定义编码器安全列表，以包括范围使用适合于在启动期间，你的应用程序中的 Unicode `ConfigureServices()`。

例如，使用默认配置可能会使用 Razor HtmlHelper 如下所示：

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

当您查看的网页的源时，你将看到已呈现，如下所示，使用中文文本编码；

```
<p>This link text is in Chinese: <a href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

若要扩大范围的字符视为安全编码器您将插入以下行到 `ConfigureServices()` 中的方法 `startup.cs`；

```
services.AddSingleton<HtmlEncoder>(
    HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
                                              UnicodeRanges.CjkUnifiedIdeographs }));
```

此示例扩大安全的列表，使其包含 Unicode 范围 CjkUnifiedIdeographs。呈现的输出将现在变为

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

安全列表范围被指定为 Unicode 代码图表、不语言。Unicode 标准具有的列表 [代码图表](#) 可用来查找包含你字符图表。每个编码器，Html、JavaScript 和 Url，必须单独配置。

注意

自定义安全列表仅影响通过 DI 来源的编码器。如果你直接访问通过编码器

```
System.Text.Encodings.Web.*Encoder.Default
```

 然后默认情况下，基本拉丁将使用仅安全列表。

编码的 take 应放置位置？

常规接受的做法是编码发生在输出和编码的值应永远不会存储在数据库中。在输出编码，可将使用数据，例如，从 HTML 为查询字符串值更改。它还使您可以轻松地搜索您的数据，而无需搜索前对值进行编码，并使您可以充分利用任何更改或对编码器所做的 bug 修复。

验证在 XSS 预防方法

验证可以在限制 XSS 攻击的有用工具。例如，一个简单的数值字符串只能包含字符 0-9 将不会触发 XSS 受到攻击。验证变得更复杂应想要接受用户输入的中的 HTML HTML 输入内容分析为难以进行，即使不是不可能。

MarkDown 以及其他文本的格式将丰富的输入的更安全选项。你应永远不会依赖于单独的验证。始终对不受信任的输入，输出之前进行编码，无论何种验证已执行。

启用 ASP.NET Core 中的跨源请求 (CORS)

2018/4/10 • 10 min to read • [Edit Online](#)

通过Mike Wasson, Shayne 贝叶, 和Tom Dykstra

浏览器安全阻止到另一个域进行 AJAX 请求 web 页。此限制称为同源策略，可防止恶意站点读取另一个站点中的敏感数据。但是，有时你可能想要允许对你的 web API 进行的跨源请求其他站点。

跨域资源共享(CORS) 是一种 W3C 标准，允许服务器放宽了同源策略。使用 CORS，服务器可以显式允许某些跨源请求时拒绝其他人。CORS 是更安全、更灵活比早期技术如JSONP。本主题演示如何在 ASP.NET Core 应用程序中启用 CORS。

什么是"相同源"？

如果它们具有相同的方案、主机和端口，两个 Url 将具有相同的原点。[\(RFC 6454\)](#)

这些两个 Url 具有相同的来源：

- `http://example.com/foo.html`
- `http://example.com/bar.html`

这些 Url 有两个比以前的不同来源：

- `http://example.net` -不同的域
- `http://www.example.com/foo.html` 的不同子域
- `https://example.com/foo.html` -不同的方案
- `http://example.com:9000/foo.html` -不同的端口

注意

比较来源时，Internet Explorer 不考虑端口。

CORS 设置

若要设置你的应用程序的 CORS 添加 `Microsoft.AspNetCore.Cors` 包到你的项目。

添加会在 `Startup.cs` 的 CORS 服务：

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

启用 CORS 的中间件

若要启用 CORS 整个应用程序添加 CORS 中间件请求管道使用 `UseCors` 扩展方法。请注意 CORS 中间件，必须在任何定义的终结点之前你想要支持跨源请求（例如应用程序中。对任何调用之前 `UseMvc`）。

添加 CORS 中间件使用时，可以指定跨域策略 `CorsPolicyBuilder` 类。有两种方法可以实现此目的。第一种是使用

lambda 调用 UseCors:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Shows UseCors with CorsPolicyBuilder.
    app.UseCors(builder =>
        builder.WithOrigins("http://example.com"));

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

注意:而无需尾部反斜杠, 必须指定的 URL (/)。如果 URL 终止与 /, 比较将返回 false 并且将返回没有标头。

Lambda 采用 `CorsPolicyBuilder` 对象。你将找到一份[配置选项](#)本主题中更高版本。在此示例中, 该策略允许来自的跨域请求 `http://example.com` 和其他的原点。

请注意 `CorsPolicyBuilder` 有 fluent API, 因此你可以将方法调用的链接:

```
app.UseCors(builder =>
    builder.WithOrigins("http://example.com")
        .AllowAnyHeader()
);
```

第二种方法是定义一个或多个命名的 CORS 策略, 并在运行时按名称然后选择的策略。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("AllowSpecificOrigin",
            builder => builder.WithOrigins("http://example.com"));
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Shows UseCors with named policy.
    app.UseCors("AllowSpecificOrigin");
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

此示例添加名为"AllowSpecificOrigin"的 CORS 策略。若要选择的策略, 将名称传递给 `UseCors`。

在 MVC 启用 CORS

MVC 或者可用于应用每个操作，每个控制器，或全局范围内的所有控制器的特定 CORS。使用 MVC 若要启用 CORS 时使用相同的 CORS 服务，但 CORS 中间件不。

每个操作

若要指定特定的操作的 CORS 策略添加 `[EnableCors]` 属性设为该操作。指定策略名称。

```
[HttpGet]
[EnableCors("AllowSpecificOrigin")]
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

每个控制器

若要指定一个特定的控制器的 CORS 策略添加 `[EnableCors]` 到控制器类属性。指定策略名称。

```
[Route("api/[controller]")]
[EnableCors("AllowSpecificOrigin")]
public class ValuesController : Controller
```

全局

你可以启用 CORS 全局范围内的所有控制器的添加 `CorsAuthorizationFilterFactory` 到全局筛选器集合的筛选器：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new CorsAuthorizationFilterFactory("AllowSpecificOrigin"));
    });
}
```

优先顺序是：操作，控制器，全局。操作级别的策略优先于控制器级别的策略，并控制器级别策略优先于全局策略。

禁用 CORS

若要禁用 CORS 控制器或操作，使用 `[DisableCors]` 属性。

```
[HttpGet("{id}")]
[DisableCors]
public string Get(int id)
{
    return "value";
}
```

CORS 策略选项

本部分介绍你可以在 CORS 策略设置的各种选项。

- [设置允许的来源](#)
- [设置允许的 HTTP 方法](#)
- [设置允许的请求标头](#)

- [设置公开的响应标头](#)
- [在跨域请求中的凭据](#)
- [将预检过期时间设置](#)

对于某些选项可能有有助于读取[如何 CORS 适用](#)第一个。

设置允许的来源

若要允许一个或多个特定来源：

```
options.AddPolicy("AllowSpecificOrigins",
builder =>
{
    builder.WithOrigins("http://example.com", "http://www.contoso.com");
});
```

若要允许所有来源：

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

namespace CorsExample4
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>
            {
                // BEGIN01
                options.AddPolicy("AllowSpecificOrigins",
                    builder =>
                {
                    builder.WithOrigins("http://example.com", "http://www.contoso.com");
                });
                // END01

                // BEGIN02
                options.AddPolicy("AllowAllOrigins",
                    builder =>
                {
                    builder.AllowAnyOrigin();
                });
                // END02

                // BEGIN03
                options.AddPolicy("AllowSpecificMethods",
                    builder =>
                {
                    builder.WithOrigins("http://example.com")
                        .WithMethods("GET", "POST", "HEAD");
                });
                // END03

                // BEGIN04
                options.AddPolicy("AllowAllMethods",
                    builder =>
```

```

        {
            builder.WithOrigins("http://example.com")
                .AllowAnyMethod();
        });
// END04

// BEGIN05
options.AddPolicy("AllowHeaders",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .WithHeaders("accept", "content-type", "origin", "x-custom-header");
});
// END05

// BEGIN06
options.AddPolicy("AllowAllHeaders",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .AllowAnyHeader();
});
// END06

// BEGIN07
options.AddPolicy("ExposeResponseHeaders",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .WithExposedHeaders("x-custom-header");
});
// END07

// BEGIN08
options.AddPolicy("AllowCredentials",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .AllowCredentials();
});
// END08

// BEGIN09
options.AddPolicy("SetPreflightExpiration",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
});
// END09
});

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseCors("AllowSpecificOrigins");
    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}

```

```
    }  
}
```

允许从任何源的请求之前应该认真考虑。这意味着按原义任何网站可 AJAX 调用你的 API。

设置允许的 HTTP 方法

若要允许所有的 HTTP 方法：

```
options.AddPolicy("AllowAllMethods",  
    builder =>  
    {  
        builder.WithOrigins("http://example.com")  
            .AllowAnyMethod();  
    });
```

这会影响预检请求和访问-控制的允许的方法标头。

设置允许的请求标头

CORS 预检请求可能包括一个访问控制的请求标头标头，列出由应用程序设置的 HTTP 标头（所谓的“创作请求标头”）。

到白名单特定的标头：

```
options.AddPolicy("AllowHeaders",  
    builder =>  
    {  
        builder.WithOrigins("http://example.com")  
            .WithHeaders("accept", "content-type", "origin", "x-custom-header");  
    });
```

若要允许所有创作请求标头：

```
options.AddPolicy("AllowAllHeaders",  
    builder =>  
    {  
        builder.WithOrigins("http://example.com")  
            .AllowAnyHeader();  
    });
```

浏览器中都不完全一致它们如何设置访问控制的请求标头。如果您设置标头为任何以外“*”，则应包含至少“接受”，“内容类型”，“源”和任何你想要支持的自定义标头。

设置公开的响应标头

默认情况下，浏览器不会公开所有向应用程序的响应标头。（See <http://www.w3.org/TR/cors/#simple-response-header>）默认为可用的响应标头是：

- Cache-Control
- Content-Language
- Content-Type
- 过期
- Last-Modified
- 杂注

CORS 规范调用这些简单响应标头。若要使应用程序的其他标头：

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .WithExposedHeaders("x-custom-header");
});
```

在跨域请求中的凭据

凭据需要 CORS 请求中的特殊处理。默认情况下，浏览器不会发送与跨域请求的任何凭据。凭据包括 cookie 和 HTTP 身份验证方案。若要发送的跨域请求的凭据，客户端必须设置为 true 的 XMLHttpRequest.withCredentials。

直接使用 XMLHttpRequest:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'http://www.example.com/api/test');
xhr.withCredentials = true;
```

在 jQuery:

```
$.ajax({
    type: 'get',
    url: 'http://www.example.com/home',
    xhrFields: {
        withCredentials: true
    }
})
```

此外，则服务器必须允许凭据。若要允许跨域凭据：

```
options.AddPolicy("AllowCredentials",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .AllowCredentials();
});
```

现在 HTTP 响应将包括一个访问控制的允许的凭据标头，它指示浏览器服务器允许跨域请求凭据。

如果浏览器发送凭据，但响应不包含有效的访问控制的允许的凭证标头，浏览器不会公开的响应应用程序，并且 AJAX 请求失败。

允许跨域凭据时要小心。在另一个域网站可以将登录的用户的凭据发送到代表该用户的用户的不知情的情况下上的应用。CORS 规范还会说明该设置来源为“*”(所有来源) 无效如果 Access-Control-Allow-Credentials 标头是否存在。

将预检过期时间设置

访问控制的最长时间标头指定可以缓存预检请求的响应的时间就越长。若要设置此标头：

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
{
    builder.WithOrigins("http://example.com")
        .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
});
```

CORS 的工作原理

本部分介绍在 CORS 请求的 HTTP 消息级别中会发生什么情况。请务必了解 CORS，以便可以正确配置的 CORS 策略工作原理以及 troubleshooted 时出现意外的行为。

CORS 规范引入了几个新的 HTTP 标头启用跨域请求。如果浏览器支持 CORS，则将设置为跨源请求自动这些标头。不需要启用 CORS 自定义 JavaScript 代码。

下面是跨域请求的示例。Origin 标头提供的域的正在发出请求的站点：

```
GET http://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: http://myclient.azurewebsites.net/
Accept: /*
Accept-Language: en-US
Origin: http://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

如果服务器允许该请求，则将在响应中设置的访问控制的允许的域标头。此标头的值匹配的 Origin 标头请求，或是通配符值“*”，允许任何来源的含义：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

如果响应不包含访问控制的允许的域标头，AJAX 请求失败。具体而言，浏览器不允许该请求。即使服务器将返回成功的响应，浏览器不会响应可供客户端应用程序。

预检请求

对于某些 CORS 请求，浏览器发送一个其他的请求，调用“预检请求”，再将发送实际请求的资源。如果以下条件为真，浏览器可以跳过预检请求：

- 请求方法是 GET、HEAD 或 POST、和
- 应用程序不设置任何请求标头以外，接受语言内容 Accept-language、内容类型或最后一个事件 ID 和
- 内容类型标头（如果设置）是以下之一：
 - application/x-www-form-urlencoded
 - multipart/form-data
 - 文本/无格式

有关请求标头规则适用于应用程序通过 XMLHttpRequest 对象上调用 setRequestHeader 将设置的标头。（CORS 规范调用这些“作者请求标头”。）该规则不适用于浏览器可以设置，如用户代理、主机或内容-长度标头。

下面是预检请求的示例：

```
OPTIONS http://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: /*
Origin: http://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

预检请求使用 HTTP OPTIONS 方法。它包括两个特殊的标头：

- 访问控制的请求的方法：将用作实际请求 HTTP 方法。
- 访问-控制-请求的标头：应用程序设置实际请求的请求标头的列表。（同样，这不包括浏览器设置的标头。）

下面是假定服务器允许请求的示例响应：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: http://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

响应包括列出了允许的方法的访问-控制的允许的方法标头和（可选）访问的控制的允许的标头标头，其中列出了允许的标头。如果预检请求成功，则浏览器发送实际请求，如前面所述。

与 ASP.NET 和 ASP.NET Core 共享在应用之间的 cookie

2018/5/17 • 7 min to read • [Edit Online](#)

通过[Rick Anderson](#)和[Luke Latham](#)

网站通常包含单独的 web 应用程序协同工作。若要提供单一登录 (SSO) 体验，在站点内的 web 应用程序必须共享身份验证 cookie。若要支持这种情况下，数据保护堆栈允许在共享 Katana cookie 身份验证和 ASP.NET Core cookie 身份验证票证。

[查看或下载示例代码\(如何下载\)](#)

此示例阐释了在使用 cookie 身份验证的三个应用间共享的 cookie:

- ASP.NET 核心 2.0 Razor 页应用而无需使用[ASP.NET 核心标识](#)
- 使用 ASP.NET Core 标识的 ASP.NET Core 2.0 MVC 应用程序
- 使用 ASP.NET Identity 的 ASP.NET Framework 4.6.1 MVC 应用程序

在下面的示例：

- 身份验证 cookie 名称设置为的一个常见值 `.AspNet.SharedCookie`。
- `AuthenticationType` 设置为 `Identity.Application` 显式或默认情况下。
- 常见的应用程序名称用于启用数据保护系统共享数据保护密钥 (`SharedCookieApp`)。
- `Identity.Application` 使用作为身份验证方案。使用任何方案，它必须一致地使用内和跨共享的 cookie 应用的默认架构或通过显式设置。加密和解密 cookie，因此必须跨应用使用一致的方案时，则使用方案。
- 一个常见[数据保护密钥](#)使用存储位置。示例应用使用名为的文件夹`KeyRing`根目录下的解决方案，用于保存数据保护密钥。
- 在 ASP.NET Core 应用中，`PersistKeysToFileSystem`用于设置的密钥存储位置。`SetApplicationName`用于配置公用的共享应用程序名。
- 在 .NET Framework 应用中，cookie 身份验证中间件使用的实现[DataProtectionProvider](#)。
`DataProtectionProvider` 提供对身份验证 cookie 负载数据进行加密和解密的数据保护服务。
`DataProtectionProvider` 实例都独立于所使用的应用程序的其他部分的数据保护系统。
 - `DataProtectionProvider.Create (System.IO.DirectoryInfo、操作<IDataProtectionBuilder >)` 接受`DirectoryInfo`指定用于数据保护密钥存储的位置。示例应用程序提供的路径`KeyRing`文件夹`DirectoryInfo`。`DataProtectionBuilderExtensions.SetApplicationName`设置常见的应用程序名称。
 - `DataProtectionProvider`需要[Microsoft.AspNetCore.DataProtection.Extensions](#) NuGet 包。若要获取此包的 ASP.NET 核心 2.0 及更高版本的应用，引用[Microsoft.AspNetCore.All](#) metapackage。如果目标.NET Framework，添加对的包引用 `Microsoft.AspNetCore.DataProtection.Extensions`。

共享 ASP.NET Core 应用之间的身份验证 cookie

当使用 ASP.NET 核心标识：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

在 `ConfigureServices` 方法，请使用[ConfigureApplicationCookie](#)扩展方法，以设置 cookie 数据保护服务。

```
services.AddDataProtection()
    .PersistKeysToFileSystem(GetKeyRingDirInfo())
    .SetApplicationName("SharedCookieApp");

services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
});
```

必须在应用之间共享数据保护密钥和应用程序名称。在示例应用中，`GetKeyRingDirInfo` 返回到的公共密钥存储位置 `PersistKeysToFileSystem` 方法。使用 `SetApplicationName` 配置公用的共享应用程序名（`SharedCookieApp` 示例中）。有关详细信息，请参阅[配置数据保护](#)。

请参阅 `CookieAuthWithIdentity.Core` 项目中[示例代码\(如何下载\)](#)。

当直接使用 cookie：

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddDataProtection()
    .PersistKeysToFileSystem(GetKeyRingDirInfo())
    .SetApplicationName("SharedCookieApp");

services.AddAuthentication("Identity.Application")
    .AddCookie("Identity.Application", options =>
{
    options.Cookie.Name = ".AspNet.SharedCookie";
});
```

必须在应用之间共享数据保护密钥和应用程序名称。在示例应用中，`GetKeyRingDirInfo` 返回到的公共密钥存储位置 `PersistKeysToFileSystem` 方法。使用 `SetApplicationName` 配置公用的共享应用程序名（`SharedCookieApp` 示例中）。有关详细信息，请参阅[配置数据保护](#)。

请参阅 `CookieAuth.Core` 项目中[示例代码\(如何下载\)](#)。

加密存放的数据保护密钥

对于生产部署，配置 `DataProtectionProvider` 来加密存放使用 DPAPI 或 x509 证书的密钥。请参阅[将加密密钥在 Rest](#) 有关详细信息。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddDataProtection()
    .ProtectKeysWithCertificate("thumbprint");
```

共享身份验证 cookie 之间 ASP.NET 4.x 和 ASP.NET Core 应用

ASP.NET 4.x 应用程序使用 Katana cookie 身份验证中间件。该对话框可以配置为生成与 ASP.NET Core cookie 身份验证中间件兼容的身份验证 cookie。这样，段落将大型站点的单个应用升级时站点内提供顺畅的 SSO 体验。

提示

当应用使用 Katana cookie 身份验证中间件时，它将调用 `UseCookieAuthentication` 在项目的 `Startup.Auth.cs` 文件。ASP.NET 4.x web 应用程序项目创建与 Visual Studio 2013 和更高版本默认情况下使用 Katana cookie 身份验证中间件。

注意

ASP.NET 4.x 应用程序必须面向.NET Framework 4.5.1 或更高版本。否则，所需的 NuGet 包将无法安装。

若要共享的 ASP.NET 4.x 应用程序和 ASP.NET Core 应用之间的身份验证 cookie，如前所述，将 ASP.NET Core 应用配置，然后通过执行以下步骤配置 ASP.NET 4.x 应用程序。

1. 安装包 [Microsoft.Owin.Security.Interop](#) 到每个 ASP.NET 4.x 应用程序。
2. 在 `Startup.Auth.cs`，找到调用 `UseCookieAuthentication` 和对其进行修改，如下所示。更改要匹配使用 ASP.NET Core cookie 身份验证中间件的名称的 cookie 名称。提供的一个实例 `DataProtectionProvider` 初始化为常见的数据保护密钥的存储位置。请确保应用程序名称设置为使用共享 cookie 的所有应用程序的常见应用程序名称 `SharedCookieApp` 示例应用中。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = "Identity.Application",
    CookieName = ".AspNet.SharedCookie",
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        OnValidateIdentity =
            SecurityStampValidator
                .OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
                    validateInterval: TimeSpan.FromMinutes(30),
                    regenerateIdentity: (manager, user) =>
                        user.GenerateUserIdentityAsync(manager))
    },
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(GetKeyRingDirInfo(),
                (builder) => { builder.SetApplicationName("SharedCookieApp"); })
            .CreateProtector(
                "Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware",
                "Identity.Application",
                "v2"))),
    CookieManager = new ChunkingCookieManager()
});
// If not setting http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier and
// http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider,
// then set UniqueClaimTypeIdentifier to a claim that distinguishes unique users.
System.Web.Helpers.AntiForgeryToken.UniqueClaimTypeIdentifier =
    "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
```

请参阅 [CookieAuthWithIdentity.NETFramework](#) 项目中 [示例代码\(如何下载\)](#)。

身份验证类型时生成的用户标识，必须与匹配中定义的类型 `AuthenticationType` 设置 `UseCookieAuthentication`。

`Models/IdentityModels.cs`:

```
public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
{
    // Note the authenticationType must match the one defined in
    CookieAuthenticationOptions.AuthenticationType
    var userIdentity = await manager.CreateIdentityAsync(this, "Identity.Application");
    // Add custom user claims here
    return userIdentity;
}
```

使用常见的用户数据库

确认每个应用程序的标识系统指向相同的用户数据库。否则，标识系统会生成在运行时失败时它尝试匹配针对其数据库中的信息的身份验证 cookie 中的信息。

ASP.NET Core 中的性能

2018/4/10 • 1 min to read • [Edit Online](#)

- [缓存响应](#)
 - [内存中缓存](#)
 - [使用分布式缓存](#)
 - [响应缓存](#)
- [响应压缩中间件](#)

在 ASP.NET Core 中缓存响应

2018/4/10 • 1 min to read • [Edit Online](#)

- [内存中缓存](#)
- [使用分布式缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存](#)
- [响应缓存中间件](#)
- [缓存标记帮助程序](#)
- [分布式缓存标记帮助程序](#)

缓存在内存中 ASP.NET 核心

2018/5/4 • 5 min to read • [Edit Online](#)

作者: [Rick Anderson](#), [John Luo](#), 和 [Steve Smith](#)

[查看或下载示例代码\(如何下载\)](#)

缓存的基础知识

通过减少生成内容所需的工作, 缓存可以显著提高应用的性能和可伸缩性。缓存对不经常更改的数据效果最佳。缓存生成的数据副本的返回速度可以比从原始源返回更快。在编写并测试应用时, 应避免依赖缓存的数据。

ASP.NET Core 支持多种不同的缓存。最简单的缓存基于 [IMemoryCache](#), 它表示存储在 Web 服务器内存中的缓存。在包含多个服务器的服务器场上运行的应用应确保在使用内存中缓存时, 会话是粘性的。粘性会话可确保来自客户端的后续请求都转到同一台服务器。例如, Azure Web 应用使用[应用程序请求路由\(ARR\)](#) 将所有的后续请求路由到同一台服务器。

Web 场中的非粘性会话需要[分布式缓存](#)以避免缓存一致性问题。对于某些应用来说, 分布式缓存可以支持比内存中缓存更高程度的横向扩展。使用分布式缓存可将缓存内存卸载到外部进程。

除非将 [CacheItemPriority](#) 设置为 `CacheItemPriority.NeverRemove`, 否则 [IMemoryCache](#) 缓存会在内存压力下清除缓存条目。可以通过设置 [CacheItemPriority](#) 来调整缓存在内存压力下清除项目的优先级。

内存中缓存可以存储任何对象; 分布式缓存接口仅限于 `byte[]`。

使用 IMemoryCache

内存中缓存是使用[依赖关系注入](#)从应用中引用的服务。请在 `ConfigureServices` 中调用 `AddMemoryCache` :

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

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

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

在构造函数中请求 [IMemoryCache](#) 实例:

```
public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
}
```

IMemoryCache 需要 NuGet 包"Microsoft.Extensions.Caching.Memory"。

下面的代码使用 TryGetValue 检查时间是否在缓存中。如果未缓存的时间，创建并添加到缓存中，使用新的条目设置。

```
public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

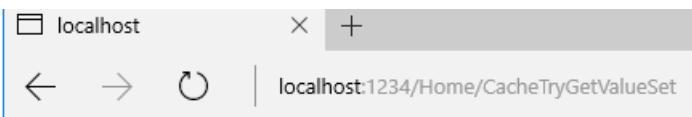
将会显示当前时间和缓存的时间：

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a href="#" asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a href="#" asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a href="#" asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a href="#" asp-controller="Home" asp-action="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>
        <li><a href="#" asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
    </ul>
</div>

<h3>Current Time: @DateTime.Now.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.ToString())</h3>
```

已缓存 DateTime 值保持在缓存中，尽管在超时期限（和由于内存压力没有逐出）的请求。下图显示当前时间以及从缓存中检索的较早时间：



Scenarios

- [Basic cache operations](#)
- [Cache entry with eviction callback](#)
- [Dependent cache entries](#)

Actions

- [TryGetValue and Set](#)
- [Get](#)
- [GetOrCreate](#)
- [GetOrCreateAsync](#)
- [Remove](#)

Current Time: 17:04:01.1913080

Cached Time: 17:03:39.9454218

下面的代码使用[GetOrCreate](#)和[GetOrCreateAsync](#)缓存数据。

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}

public async Task<IActionResult> CacheGetOrCreateAsync()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
    {
        entry SlidingExpiration = TimeSpan.FromSeconds(3);
        return Task.FromResult(DateTime.Now);
    });

    return View("Cache", cacheEntry);
}
```

以下代码调用[Get](#)来提取缓存的时间：

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

有关缓存方法的说明，请参阅[IMemoryCache 方法](#)和[CacheExtensions 方法](#)

使用 MemoryCacheEntryOptions

下面的示例执行以下操作：

- 设置绝对到期时间。这是条目可以被缓存的最长时间，防止可调过期持续更新时该条目过时太多。
- 设置可调过期时间。访问此缓存项的请求将重置可调过期时钟。
- 将缓存优先级设置为 `CacheItemPriority.NeverRemove`。
- 设置一个 `PostEvictionDelegate` 它将在条目从缓存中清除后调用。在代码中运行该回调的线程不同于从缓存中移除条目的线程。

```

public IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}

```

缓存依赖项

以下示例演示在依赖项过期时如何使缓存项过期。会将 `CancellationToken` 添加到缓存项。在 `Cancel` 上调用 `CancellationTokenSource`，时，这两个缓存条目都将被清除。

```

public IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Parent entry was evicted. Reason: {reason}.";;
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}

```

使用 `CancellationTokenSource` 可以将多个缓存条目作为一个组来清除。使用上面代码中的 `using` 模式时 `using` 块中创建的缓存条目会继承触发器和过期时间设置。

附加说明

- 使用回调重新填充缓存项时：
 - 多个请求可能会发现缓存的键值为空，因为回调尚未完成。
 - 这可能导致重新填充缓存的项的多个线程。
- 使用一个缓存条目创建另一个缓存条目时，子条目会复制父条目的过期令牌以及基于时间的过期设置。子不过期通过手动删除或更新的父项。

其他资源

- [使用分布式缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存](#)
- [响应缓存中间件](#)
- [缓存标记帮助程序](#)

- 分布式缓存标记帮助程序

使用 ASP.NET Core 中分布式缓存

2018/5/17 • 7 min to read • [Edit Online](#)

作者: Steve Smith

分布式缓存可以提高 ASP.NET Core 应用的性能和可伸缩性，尤其是托管在云或服务器场环境中时。本文解释了如何使用 ASP.NET Core 的内置分布式缓存抽象和实现。

[查看或下载示例代码\(如何下载\)](#)

什么是分布式的缓存

分布式的缓存共享由多个应用程序服务器 (请参阅[缓存基础知识](#))。缓存中的信息不存储在单独的 Web 服务器的内存中，并且缓存的数据可用于所有应用服务器。这具有几个优点：这提供了几个优点：

1. 所有 Web 服务器上的缓存数据都是一致的。用户不会因处理其请求的 Web 服务器的不同而看到不同的结果
2. 缓存的数据在 Web 服务器重新启动后和部署后仍然存在。删除或添加单独的 Web 服务器不会影响缓存。
3. 源数据存储区具有对它（不是使用多个内存中缓存或否缓存根本）发出的少数几个请求。

注意

如果使用 SQL Server 分布式缓存，则其中一些优势只有在为缓存而不是应用的源数据使用单独的数据库实例的情况下才会体现出来。

像任何缓存一样，分布式缓存可以显著提高应用的响应速度，因为通常情况下，数据从缓存中检索比从关系数据库（或 Web 服务）中检索快得多。

缓存配置是特定于实现的。本文介绍如何配置 Redis 和 SQL Server 分布式缓存。无论选择哪一种实现，应用都使用通用的 `IDistributedCache` 接口与缓存交互。

`IDistributedCache` 接口

`IDistributedCache` 接口包含同步和异步方法。接口允许在分布式缓存实现中添加、检索和删除项。

`IDistributedCache` 接口包含以下方法：

`Get`、`GetAsync`

采用字符串键并以 `byte[]` 形式检索缓存项（如果在缓存中找到）。

`Set`、`SetAsync`

使用字符串键向缓存添加项 `byte[]` 形式）。

`Refresh`、`RefreshAsync`

根据键刷新缓存中的项，并重置其可调过期超时值（如果有）。

`Remove`、`RemoveAsync`

根据键删除缓存项。

若要使用 `IDistributedCache` 接口：

1. 所需的 NuGet 包添加到项目文件。
2. 在 `Startup` 类的 `ConfigureServices` 方法中配置 `IDistributedCache` 的具体实现，并将其添加到该处的容器中。
3. 在应用的[中间件](#)或 MVC 控制器类中，从构造函数请求 `IDistributedCache` 的实例。实例将通过[依赖项注入](#) (DI) 提供。

注意

无需为 `IDistributedCache` 实例使用 Singleton 或 Scoped 生命周期(至少对内置实现来说是这样的)。你还可以创建一个实例，只要你可能需要一个(而不是使用[依赖关系注入](#))，但这会导致你的代码更难若要测试，并且与冲突[显式依赖关系原则](#)。

下面的示例演示如何在简单的中间件组件中使用 `IDistributedCache` 实例：

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Caching.Distributed;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DistCacheSample
{
    public class StartTimeHeader
    {
        private readonly RequestDelegate _next;
        private readonly IDistributedCache _cache;

        public StartTimeHeader(RequestDelegate next,
            IDistributedCache cache)
        {
            _next = next;
            _cache = cache;
        }

        public async Task Invoke(HttpContext httpContext)
        {
            string startTimeString = "Not found.";
            var value = await _cache.GetAsync("lastServerStartTime");
            if (value != null)
            {
                startTimeString = Encoding.UTF8.GetString(value);
            }

            httpContext.Response.Headers.Append("Last-Server-Start-Time", startTimeString);

            await _next.Invoke(httpContext);
        }
    }

    // Extension method used to add the middleware to the HTTP request pipeline.
    public static class StartTimeHeaderExtensions
    {
        public static IApplicationBuilder UseStartTimeHeader(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<StartTimeHeader>();
        }
    }
}

```

在上面的代码中，缓存值用于读取，不用于写入。在此示例中，该值仅在服务器启动时设置，并且不会更改。在多服务器方案中，要启动的最新服务器会覆盖由其他服务器设置的以前的任何值。`Get` 和 `Set` 方法使用 `byte[]` 类型。因此，字符串值必须使用 `Encoding.UTF8.GetString` (适用于 `Get`) 和 `Encoding.UTF8.GetBytes` (适用于 `Set`) 进行转换。

`Startup.cs` 中的以下代码显示要设置的值：

```
public void Configure(IApplicationBuilder app,
    IDistributedCache cache)
{
    var serverStartTimeString = DateTime.Now.ToString();
    byte[] val = Encoding.UTF8.GetBytes(serverStartTimeString);
    var cacheEntryOptions = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(30));
    cache.Set("lastServerStartTime", val, cacheEntryOptions);
```

注意

由于 `IDistributedCache` 是在 `ConfigureServices` 方法中配置的，因此它可以作为参数提供给 `Configure` 方法。将其添加作为参数将允许通过 DI 提供配置的实例。

使用分布式的 Redis 缓存

Redis 是一种开源的内存中数据存储，通常用作分布式缓存。可以在本地使用它，并且可以为 Azure 托管的 ASP.NET Core 应用配置 Azure Redis 缓存为你的 Azure 托管的 ASP.NET Core 应用。ASP.NET Core 应用使用 `RedisDistributedCache` 实例配置缓存实现。

可在 `ConfigureServices` 中配置 Redis 实现，并可通过请求 `IDistributedCache` 实例（请参阅上面的代码）在应用代码中访问它。

在示例代码中，为 `RedisCache` 环境配置服务器时使用 `staging` 实现。因此，`ConfigureStagingServices` 方法用于配置 `RedisCache`：

```
/// <summary>
/// Use Redis Cache in Staging
/// </summary>
/// <param name="services"></param>
public void ConfigureStagingServices(IServiceCollection services)
{
    services.AddDistributedRedisCache(options =>
    {
        options.Configuration = "localhost";
        options.InstanceName = "SampleInstance";
    });
}
```

注意

若要在本地计算机上安装 Redis，安装 chocolatey 程序包 <https://chocolatey.org/packages/redis-64/> 并运行 `redis-server` 从命令提示符。

使用 SQL Server 分布式缓存

`SqlServerCache` 实现允许分布式缓存使用 SQL Server 数据库作为其后备存储。若要创建 SQL Server 表，可以使用 `sql-cache` 工具，该工具将使用指定的名称和模式创建一个表。

若要使用 `sql-cache` 工具，请将 `SqlConfig.Tools` 添加到 `<ItemGroup>` 文件的 `.csproj` 元素，然后运行 `dotnet restore`。

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.Caching.SqlConfig.Tools" Version="1.0.0-msbuild3-final" />
</ItemGroup>
```

通过运行以下命令来测试 SqlConfig.Tools

```
C:\DistCacheSample\src\DistCacheSample>dotnet sql-cache create --help
```

sql-cache 工具将显示用法、选项和命令帮助。现在可以将表创建到 sql server 中，只需运行“sql-cache create”命令即可：

```
C:\DistCacheSample\src\DistCacheSample>dotnet sql-cache create "Data Source=(localdb)\v11.0;Initial Catalog=DistCache;Integrated Security=True;" dbo TestCache
info: Microsoft.Extensions.Caching.SqlConfig.Tools.Program[0]
Table and index were created successfully.
```

创建的表具有以下架构：

Column Name	Data Type	Allow Nulls
Id	nvarchar(900)	<input type="checkbox"/>
Value	varbinary(MAX)	<input type="checkbox"/>
ExpiresAtTime	datetimeoffset(7)	<input type="checkbox"/>
SlidingExpirationInSeconds	bigint	<input checked="" type="checkbox"/>
AbsoluteExpiration	datetimeoffset(7)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

像所有缓存实现一样，应用应该使用 `IDistributedCache`，实例来获取和设置缓存值，而不是使用 `SqlServerCache` 实例。此示例实现 `SqlServerCache` 中 `Production` 环境（以便在配置 `ConfigureProductionServices`）。

```
/// Use SQL Server Cache in Production
/// </summary>
/// <param name="services"></param>
public void ConfigureProductionServices(IServiceCollection services)
{
    services.AddDistributedSqlServerCache(options =>
    {
        options.ConnectionString = @"Data Source=(localdb)\v11.0;Initial Catalog=DistCache;Integrated Security=True;";
        options.SchemaName = "dbo";
        options.TableName = "TestCache";
    });
}
```

注意

`ConnectionString`（以及可选的 `SchemaName` 和 `TableName`）通常应该存储在源代码管理之外（例如存储在 `UserSecrets` 中），因为它们可能包含凭据。

建议

在决定哪种 `IDistributedCache` 实现适合应用时，请根据现有的基础架构和环境、性能要求和团队经验在 Redis

和 SQL Server 之间进行选择。如果团队更喜欢使用 Redis，那就使用它。如果团队倾向于 SQL Server，那么也应对这么做充满信心。请注意，传统的缓存解决方案将存储数据的内存中用于进行快速检索的数据。应该将常用数据存储在缓存中，将整个数据存储在后端持久性存储（如 SQL Server 或 Azure 存储）中。与 SQL Cache 相比，Redis Cache 是一种吞吐量高且延迟轻微的缓存解决方案。

其他资源

- [Redis 在 Azure 上的缓存](#)
- [在 Azure 上的 SQL 数据库](#)
- [内存中缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存](#)
- [响应缓存中间件](#)
- [缓存标记帮助程序](#)
- [分布式缓存标记帮助程序](#)

响应缓存在 ASP.NET 核心

2018/4/10 • 9 min to read • [Edit Online](#)

通过John Luo, Rick Anderson, Steve Smith, 和Luke Latham

注意

响应缓存Razor页与ASP.NET核心2.0中不支持。此功能将支持ASP.NET核心2.1版本。

[查看或下载示例代码\(如何下载\)](#)

响应缓存可减少客户端或代理对web服务器的请求数。响应缓存还可减少量工作的web服务器执行程序生成响应。响应缓存由标头，指定你希望客户端、代理和缓存响应的中间件如何控制。

添加时，web服务器可以缓存响应[响应缓存中间件](#)。

基于HTTP的响应缓存

[HTTP 1.1 缓存规范](#)介绍Internet缓存的行为方式。用于缓存的主HTTP标头是[缓存控制](#)，用于指定[缓存指令](#)。指令控制缓存行为，请求来自客户端对服务器进行其方法和响应进行地从服务器返回到客户端。请求和响应将通过代理服务器和代理服务器还必须遵循HTTP 1.1缓存功能规范。

常见[Cache-Control](#)指令下表中所示。

指令	操作
<code>public</code>	缓存可能会存储响应。
<code>private</code>	不能通过共享缓存中存储响应。专用缓存可以存储并重复使用响应。
<code>max-age</code>	客户端将不会接受其保留时间大于指定的秒数的响应。示例： <code>max-age=60</code> (60秒), <code>max-age=2592000</code> (1个月)
<code>no-cache</code>	在请求上： 缓存不得使用存储的响应来满足该请求。注意：对于客户端，为源服务器将重新生成响应，并且该中间件更新其缓存中存储的响应。 响应： 响应必须不能用于在不验证源服务器上的后续请求。
<code>no-store</code>	在请求上： 缓存必须不会将请求的存储。 响应： 缓存不得存储任何响应的一部分。

起缓存作用其他缓存标头下表所示。

HEADER	函数
<code>保留时间</code>	以秒为单位由于生成或者在源服务器已成功验证响应的时间量的估计值。
<code>Expires</code>	日期/时间后响应被视为是陈旧。

Pragma	存在向后兼容性与 HTTP/1.0 缓存设置 no-cache 行为。如果 Cache-Control 标头是否存在、Pragma 标头将被忽略。
改变	指定缓存的响应必须不发送除非所有的 vary 中缓存的响应的原始请求和新的请求标头字段所匹配。

基于 HTTP 的缓存方面请求的缓存控制指令

HTTP 1.1 缓存的缓存控制标头的规范需要遵守是有效的缓存 Cache-Control 客户端发送的标头。客户端可以发出请求与 no-cache 标头值和强制服务器生成的每个请求的新响应。

始终考虑客户端 Cache-Control 请求标头是有意义，如果你考虑 HTTP 缓存功能的目标。官方规范中，在缓存旨在减少在网络中客户端、代理和服务器满足请求的延迟和网络的开销。它不一定是一种方法来控制的源服务器上的负载。

没有对此缓存行为的当前开发人员控件时使用响应缓存中间件因为缓存规范正式遵循该中间件。将来对该中间件增强配置中间件，若要忽略的请求将允许 Cache-Control 时决定提供缓存的响应的标头。当你使用该中间件，这将为您提供更好地控制负载在你的服务器上的机会。

在 ASP.NET 核心中其他缓存技术

内存中缓存

内存中缓存使用的服务器内存来存储缓存的数据。此类型的缓存适合于单个服务器或多个服务器使用粘性会话。粘性会话意味着，客户端的请求始终路由至同一服务器进行处理。

有关详细信息，请参阅[缓存内存中](#)。

分布式的缓存

使用分布式的缓存时云或服务器场中承载应用程序数据存储在内存中。处理请求的服务器之间共享缓存。客户端可以提交用于客户端的缓存的数据是否可用的组中的任何服务器处理的请求。ASP.NET Core 提供 SQL Server 和分布式的 Redis 缓存。

有关详细信息，请参阅[方式使用分布式缓存](#)。

缓存标记帮助器

你可以使用缓存标记帮助器缓存中的 MVC 视图或 Razor 页的内容。缓存标记帮助器使用的内存中缓存来存储数据。

有关详细信息，请参阅[ASP.NET 核心 mvc 缓存标记帮助器](#)。

分布式缓存标记帮助程序

你可以使用分布式缓存标记帮助程序缓存的 MVC 视图或分布式的云或 web 场方案中的 Razor 页面中的内容。分布式缓存标记帮助器使用 SQL Server 或 Redis 来存储数据。

有关详细信息，请参阅[分布式缓存标记帮助器](#)。

ResponseCache 属性

[ResponseCacheAttribute](#)指定在缓存中，响应设置适当的标头的所需的参数。

警告

禁用缓存的内容，其中包含已经过身份验证的客户端的信息。仅应为不会更改基于用户的标识或用户已登录的内容启用缓存。

`VaryByQueryKeys` 存储的响应因给定的查询密钥列表的值。时的单个值 `*` 是响应的所有请求查询字符串参数提供该中间件而异。`VaryByQueryKeys` 需要 ASP.NET Core 1.1 或更高版本。

必须启用响应缓存中间件设置 `VaryByQueryKeys` 属性；否则，会引发一个运行时异常。没有为相应的 HTTP 标头 `VaryByQueryKeys` 属性。该属性是一项 HTTP 功能由响应缓存中间件。若要提供缓存的响应的中间件，对于查询字符串和查询字符串值必须匹配上一个请求。例如，考虑的请求和下表中显示结果的序列。

请求	结果
<code>http://example.com?key1=value1</code>	从服务器返回
<code>http://example.com?key1=value1</code>	返回从中间件
<code>http://example.com?key1=value2</code>	从服务器返回

第一个请求是由服务器返回，并在中间件中缓冲。由于查询字符串与上一个请求，中间件会返回第二个请求。因为查询字符串值不匹配的上一个请求，第三个请求未处于中间件缓存。

`ResponseCacheAttribute` 用于配置和创建（通过 `IFilterFactory`）`ResponseCacheFilter`。`ResponseCacheFilter` 执行的工作的更新的相应 HTTP 标头和响应功能。筛选器：

- 删除任何现有标头 `Vary`、`Cache-Control` 和 `Pragma`。
- 写出相应的标头设置的属性上基于 `ResponseCacheAttribute`。
- 更新缓存中项 HTTP 功能，如果响应 `VaryByQueryKeys` 设置。

改变

此标头只会写入时 `VaryByHeader` 属性设置。设置为 `Vary` 属性的值。下面的示例使用 `varyByHeader` 属性：

```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public IActionResult About2()
{
```

你可以查看你的浏览器的网络工具的响应标头。下图显示了输出上边缘 F12 网络选项卡上时 `About2` 刷新操作方法：

Headers Body Parameters Cookies Timings

Request URL: <http://localhost/Home/About2>

Request Method: GET

Status Code: 200 / OK

Request Headers

Accept: text/html, application/xhtml+xml, image/jxr, */*

Accept-Encoding: gzip, deflate

Accept-Language: en-US, en; q=0.8, zh-Hans-CN; q=0.5, zh-Hans; q=0.3

Connection: Keep-Alive

Host: localhost

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36...

Response Headers

Cache-Control: public, max-age=30

Content-Type: text/html; charset=utf-8

Date: Fri, 18 Nov 2016 21:45:14 GMT

Server: Kestrel

Transfer-Encoding: chunked

Vary: User-Agent

NoStore 和 Location.None

NoStore 重写的大多数其他属性。当此属性设置为 true、Cache-Control 标头设置为 no-store。如果 Location 设置为 None：

- 将 Cache-Control 设置为 no-store,no-cache。
- 将 Pragma 设置为 no-cache。

如果 NoStore 是 false 和 Location 是 None，Cache-Control 和 Pragma 设置为 no-cache。

通常情况下设置 NoStore 到 true 错误页上。例如：

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

这将导致以下标头：

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

位置和持续时间

若要启用缓存，Duration 必须设置为正值和 Location 必须是 Any (默认值) 或 Client。在这种情况下，Cache-Control 标头设置为位置值后跟 max-age 的响应。

注意

Location 选项的 Any 和 Client 将转换 Cache-Control 的标头值 public 和 private 分别。按前面所述，设置 Location 到 None 设置同时 Cache-Control 和 Pragma 标头到 no-cache。

下面生成通过设置一个示例，演示标头 Duration 并保留默认值 Location 值：

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    return View();
}
```

这将产生以下标头：

```
Cache-Control: public,max-age=60
```

缓存配置文件

而不是复制 `ResponseCache` 设置 MVC 中时，可以将许多控制器操作属性而言，缓存配置文件上的设置配置为选项 `ConfigureServices` 中的方法 `Startup`。在引用的缓存配置文件中找到值用作通过默认 `ResponseCache` 属性和重写了指定的属性上的任何属性。

设置缓存配置文件：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default",
            new CacheProfile()
            {
                Duration = 60
            });
        options.CacheProfiles.Add("Never",
            new CacheProfile()
            {
                Location = ResponseCacheLocation.None,
                NoStore = true
            });
    });
}
```

引用缓存配置文件：

```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
}
```

`ResponseCache` 特性可应用于操作（方法）和控制器（类）。方法级别的属性重写在类级别特性中指定的设置。

在上面的示例中，类级别属性指定持续时间为 30 秒，而方法级别属性引用与设置为 60 秒的持续时间的缓存配置文件。

生成的标头：

```
Cache-Control: public,max-age=60
```

其他资源

- [在缓存中存储响应](#)
- [Cache-Control](#)
- [缓存在内存](#)
- [使用分布式缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存中间件](#)
- [缓存标记帮助程序](#)
- [分布式缓存标记帮助程序](#)

响应缓存在 ASP.NET 核心中的中间件

2018/5/4 • 8 min to read • [Edit Online](#)

通过Luke Latham和John Luo

[查看或下载示例代码\(如何下载\)](#)

此文章介绍了如何在 ASP.NET Core 应用程序中配置缓存响应的中间件。该中间件确定响应何时可缓存、存储响应和从缓存充当响应。有关 HTTP 缓存功能的简介和 `ResponseCache` 属性，请参阅[响应缓存](#)。

包

若要在项目中包含该中间件，添加到引用 `Microsoft.AspNetCore.ResponseCaching` 包或使用 `Microsoft.AspNetCore.All` 包 (ASP.NET Core 2.0 或更高版本时目标.NET 核心)。

配置

在 `ConfigureServices`，将该中间件添加到服务集合。

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

配置应用程序以使用与中间件 `UseResponseCaching` 扩展方法，将该中间件添加到请求处理管道。示例应用添加 `Cache-Control` 最多 10 秒钟来缓存可缓存响应的响应的标头。该示例发送 `Vary` 标头来配置用于缓存的响应仅当该中间件 `Accept-Encoding` 的后续请求的标头与原始请求相匹配。在代码示例中，`CacheControlHeaderValue` 和 `HeaderNames` 需要 `using` 语句 `Microsoft.Net.Http.Headers` 命名空间。

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

    app.Use(async (context, next) =>
    {
        context.Response.GetTypedHeaders().CacheControl = new CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
        context.Response.Headers[HeaderNames.Vary] = new string[] { "Accept-Encoding" };

        await next();
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseMvc();
}
```

响应缓存中间件仅缓存导致状态代码为 200 (正常) 的服务器响应。任何其他响应，包括[错误页](#)，将忽略的中间件。

警告

响应包含内容的经过身份验证的客户端必须标记为不可缓存，以防止从存储并提供这些响应中间件。请参阅[缓存的条件](#)有关该中间件如何确定响应是否是可缓存的详细信息。

选项

该中间件提供三个选项用于控制响应缓存。

选项	默认值
UseCaseSensitivePaths	确定响应将在区分大小写的路径上缓存。 默认值为 <code>false</code> 。
MaximumBodySize	以字节为单位的响应正文的最大缓存大小。 默认值是 <code>64 * 1024 * 1024</code> (64 MB)。
大小限制	以字节为单位的响应缓存中间件大小限制。默认值是 <code>100 * 1024 * 1024</code> (100 MB)。

下面的示例将配置到中间件：

- 小于或等于 1024 字节的缓存响应。
- 存储响应的区分大小写的路径 (例如, `/page1` 和 `/Page1` 分开存储)。

```
services.AddResponseCaching(options =>
{
    options.UseCaseSensitivePaths = true;
    options.MaximumBodySize = 1024;
});
```

VaryByQueryKeys

当使用 MVC/Web API 控制器或 Razor 页页模型，`ResponseCache` 属性指定所需的设置适当的标头为响应缓存参数。唯一参数 `ResponseCache` 严格需要中间件的属性是 `VaryByQueryKeys`，这不对应于实际 HTTP 标头。有关详细信息，请参阅[ResponseCache 属性](#)。

当未使用 `ResponseCache` 属性中，响应缓存可能会不同 `VaryByQueryKeys` 功能。使用 `ResponseCachingFeature` 直接从 `IFeatureCollection` 的 `HttpContext`：

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();
if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

使用单个值等于 `*` 中 `VaryByQueryKeys` 随缓存所有请求查询参数而都变化。

响应缓存中间件所使用的 HTTP 标头

使用 HTTP 标头配置响应的缓存该中间件。

HEADER	详细信息
授权	如果标头存在，则不缓存响应。
Cache-Control	<p>该中间件只考虑缓存标记为响应 <code>public</code> 缓存指令。控制缓存使用以下参数：</p> <ul style="list-style-type: none">• 最长时间• <code>max-stale</code>[†]• 最小值全新• 必须重新验证• 无缓存• 无存储• 仅当-缓存• <code>private</code>• <code>public</code>• <code>s maxage</code>• 代理重新验证[‡] <p>[†]如果没有限制指定到 <code>max-stale</code>，中间件不执行任何操作。 [‡] <code>proxy-revalidate</code> 具有相同的效果 <code>must-revalidate</code>。</p> <p>有关详细信息，请参阅 RFC 7231: 请求的缓存控制指令。</p>
杂注	A <code>Pragma: no-cache</code> 请求标头中的生成相同的效果 <code>Cache-Control: no-cache</code> 。此标头中的相关指令来重写 <code>Cache-Control</code> 标头，如果存在。考虑对与 HTTP/1.0 的向后兼容性。
Set-Cookie	如果标头存在，则不缓存响应。设置一个或多个 cookie 的请求处理管道中的任何中间件防止响应缓存中间件缓存响应（例如， 基于 cookie 的 TempData 提供程序 ）。
改变	<code>Vary</code> 标头用于改变缓存的响应的另一个标头。例如，通过包括通过编码来缓存响应 <code>Vary: Accept-Encoding</code> 标头，来缓存响应的请求标头 <code>Accept-Encoding: gzip</code> 和 <code>Accept-Encoding: text/plain</code> 单独。标头值为响应 <code>*</code> 永远不会存储。
过期	通过此标头视为过时的响应不存储或检索除非重写由其他 <code>Cache-Control</code> 标头。
None-If-Match	如果该值不完整的响应从缓存提供 <code>*</code> 和 <code>ETag</code> 的响应中不匹配任何提供的值。否则，提供 304（未修改）响应。
如果-修改-自	如果 <code>If-None-Match</code> 标头不存在，如果缓存的响应日期晚于提供的值，完整的响应从缓存中提供。否则，提供 304（未修改）响应。
日期	从缓存提供服务时 <code>Date</code> 由该中间件设置标头，如果它未在原始响应上提供。
内容长度	从缓存提供服务时 <code>Content-Length</code> 由该中间件设置标头，如果它未在原始响应上提供。

HEADER	详细信息
保留时间	<code>Age</code> 在原始响应中发送的标头将被忽略。提供缓存的响应时，该中间件将计算新值。

缓存遵循请求缓存控制指令

该中间件会遵守的规则[HTTP 1.1 缓存规范](#)。在规则需要遵守是有效的缓存 `Cache-Control` 客户端发送的标头。在规范，客户端可以发出请求与 `no-cache` 标头值和强制服务器生成的每个请求的新响应。目前，没有任何开发人员控制此缓存行为时使用中间件，因为该中间件符合官方缓存规范。

为了更好地控制缓存行为，将介绍其他缓存功能的 ASP.NET Core。请参见下面的主题：

- [内存中缓存](#)
- [使用分布式缓存](#)
- [缓存 ASP.NET Core MVC 中的标记帮助器](#)
- [分布式缓存标记帮助程序](#)

疑难解答

如果缓存行为未按预期方式，确认响应是否可缓存并且能够从缓存中提供。检查请求的传入标头和响应的传出标头。启用[日志记录](#)以帮助进行调试。

当测试和故障排除缓存行为，浏览器可能设置影响不可取的方法中的缓存的请求标头。例如，浏览器可能设置 `Cache-Control` 标头到 `no-cache` 或 `max-age=0` 刷新页面时。以下工具可以显式设置请求标头和测试缓存的首选：

- [Fiddler](#)
- [Postman](#)

用于缓存的条件

- 请求必须导致服务器响应 200 (正常) 状态代码。
- 请求方法必须是 GET 或 HEAD。
- 终端中间件，如[静态文件中间件](#)，必须处理响应缓存中间件之前的响应。
- `Authorization` 标头不能存在。
- `Cache-Control` 标头参数必须是有效，并且必须标记为响应 `public` 和未标记 `private`。
- `Pragma: no-cache` 标头不能存在如果 `Cache-Control` 标头不存在，作为 `Cache-Control` 标头重写 `Pragma` 标头时存在。
- `Set-Cookie` 标头不能存在。
- `Vary` 标头参数必须是有效且不等于 `*`。
- `Content-Length` 标头值（如果设置）必须与匹配的响应正文的大小。
- [IHttpSendFileFeature](#) 未使用。
- 响应不能为指定的陈旧 `Expires` 标头和 `max-age` 和 `s-maxage` 缓存指令。
- 响应缓冲必须成功，并且响应的大小必须小于配置或默认 `SizeLimit`。
- 响应必须是可根据缓存[RFC 7234](#)规范。例如，`no-store` 指令必须在请求或响应标头字段中存在。请参阅[第 3 部分：在缓存中存储响应的 RFC 7234](#)有关详细信息。

注意

Antiforgery 系统用于生成安全令牌，以防止跨站点请求伪造 (CSRF) 攻击集 `Cache-Control` 和 `Pragma` 标头到 `no-cache` 以便不缓存响应。有关如何禁用 antiforgery 令牌 HTML 窗体元素的信息，请参阅[ASP.NET Core antiforgery 配置](#)。

其他资源

- [应用程序启动](#)
- [中间件](#)
- [内存中缓存](#)
- [使用分布式缓存](#)
- [使用更改令牌检测更改](#)
- [响应缓存](#)
- [缓存标记帮助程序](#)
- [分布式缓存标记帮助程序](#)

有关 ASP.NET 核心响应压缩中间件

2018/5/17 • 11 min to read • [Edit Online](#)

作者:Luke Latham

[查看或下载示例代码\(如何下载\)](#)

网络带宽是有限的资源。通常减小响应的大小将通常显著增加应用的响应能力。减小负载大小的一种方法是压缩应用的响应。

何时使用响应压缩中间件

使用在 IIS、Apache 或 Nginx 基于服务器的响应压缩技术。中间件的性能可能不会与匹配服务器模块。[HTTP.sys 服务器](#)和[Kestrel](#)当前不提供内置的压缩支持。

当你时，请使用响应压缩中间件：

- 无法使用以下基于服务器的压缩技术：
 - [IIS 动态压缩模块](#)
 - [Apache mod_deflate 模块](#)
 - [Nginx 压缩和解压缩](#)
- 直接在上托管：
 - [HTTP.sys 服务器](#)(以前称为[WebListener](#))
 - [Kestrel](#)

响应压缩

通常情况下，本身不压缩任何响应可从响应压缩中受益。通常本身不压缩的响应包括：CSS、JavaScript、HTML、XML 和 JSON。本机压缩的资产，如 PNG 文件，你不应将其压缩。如果你尝试进一步压缩本机压缩的响应，任何小的其他缩短的大小和传输的时间将可能屏蔽按处理压缩所花费的时间。不压缩小于大约 150-1000 字节（具体取决于该文件的内容和压缩的效率）的文件。压缩的小文件的开销可能会产生大于未压缩的文件的压缩的文件。

客户端时客户端可以处理压缩的内容，必须通过发送通知其功能的服务器 `Accept-Encoding` 与请求的标头。当服务器发送压缩的内容时，它必须包括中的信息 `Content-Encoding` 如何编码压缩的响应标头。支持的中间件的内容编码指定下表所示。

ACCEPT-ENCODING 标头值	支持的中间件	描述
<code>br</code>	否	Brotli 压缩的数据格式
<code>compress</code>	否	UNIX“压缩”的数据格式
<code>deflate</code>	否	“deflate”内的“zlib”数据格式的压缩的数据
<code>exi</code>	否	W3C 高效 XML 交换
<code>gzip</code>	是 (默认值)	gzip 文件格式

ACCEPT-ENCODING 标头值	支持的中间件	描述
identity	是	"没有编码"标识符：响应必须进行编码。
pack200-gzip	否	Java 存档的网络传输格式
*	是	编码不显式请求的任何可用内容

有关详细信息，请参阅[IANA 官方内容编码列表](#)。

该中间件，可添加额外的压缩提供程序自定义 `Accept-Encoding` 标头值。有关详细信息，请参阅[自定义提供程序](#)下面。

该中间件是能够对的质量值作出反应 (qvalue, `q`) 权重时客户端发送为优先处理压缩方案。有关详细信息，请参阅[RFC 7231: Accept-encoding](#)。

压缩算法受到压缩速度和压缩的效率之间的权衡。有效性在此上下文中引用的输出的大小压缩后。最小大小通过**最佳压缩**。

参与请求的标头，发送、缓存和接收压缩的内容是下表中所述。

HEADER	角色
<code>Accept-Encoding</code>	从客户端发送到服务器以指示编码方案可接受的客户端的内容。
<code>Content-Encoding</code>	从服务器发送到客户端以指示负载中的内容的编码。
<code>Content-Length</code>	在进行压缩时会， <code>Content-Length</code> 标头被删除，因为正文内容更改时压缩响应。
<code>Content-MD5</code>	在进行压缩时会， <code>Content-MD5</code> 移除标头，因为已更改的正文内容，且哈希将不再有效。
<code>Content-Type</code>	指定内容的 MIME 的类型。每个响应应指定其 <code>Content-Type</code> 。该中间件检查此值以确定是否应压缩响应。该中间件指定的一组 默认 MIME 类型 其可以进行编码，但您可以替换或添加 MIME 类型。
<code>Vary</code>	由值为服务器发送时 <code>Accept-Encoding</code> 到客户端和代理， <code>Vary</code> 标头指示的客户端或代理，它应缓存（有所不同）响应基于值 <code>Accept-Encoding</code> 的请求标头。返回具有内容的结果 <code>Vary: Accept-Encoding</code> 标头是同时压缩和未压缩的响应将单独进行缓存。

你可以浏览的功能与的响应压缩中间件[示例应用程序](#)。此示例阐释了：

- 使用 gzip 和自定义压缩提供程序的应用程序响应的压缩。
- 如何将 MIME 类型添加到压缩的 MIME 类型的默认列表。

包

若要在你的项目中包含该中间件，添加到引用 `Microsoft.AspNetCore.ResponseCompression` 包或使用 `Microsoft.AspNetCore.All` 包。此功能适用于面向 ASP.NET Core 1.1 或更高版本的应用。

配置

下面的代码演示如何启用响应压缩中间件与默认 gzip 压缩和默认 MIME 类型。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddResponseCompression();
    })
    .Configure(app =>
    {
        app.UseResponseCompression();

        app.Run(async context =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync(LoremIpsum.Text);
        });
    })
    .Build();
```

注意

使用之类的工具 [Fiddler](#), [Firebug](#), 或 [Postman](#) 设置 `Accept-Encoding` 请求标头并研究响应标头、大小和正文。

提交到示例应用程序而无需请求 `Accept-Encoding` 标头并观察响应是否未压缩。`Content-Encoding` 和 `Vary` 标头不在响应上存在。

The screenshot shows a Fiddler session with the following details:

- Request Headers:**
 - Client**: User-Agent: Fiddler
 - Transport**: Host: localhost:5000
- Response Headers:**
 - HTTP/1.1 200 OK
 - Date: Wed, 11 Jan 2017 18:30:11 GMT
 - Content-Type: text/plain
 - Server: Kestrel
 - Content-Length: 2032
- Raw Response:** The response body contains a large amount of placeholder text ("Lorem ipsum...") which is highlighted with a red box.

向示例应用程序与提交一个申请 `Accept-Encoding: gzip` 标头并观察压缩的响应。`Content-Encoding` 和 `Vary` 标头会显示在响应。

Headers | TextView | WebForms | HexView | Auth | Cookies | Raw | JSON | XML |

Request Headers

GET / HTTP/1.1

Client

Accept-Encoding: gzip
User-Agent: Fiddler

Transport

Host: localhost:5000

Response body is encoded. Click to decode.

Get SyntaxView Transformer Headers TextView ImageView HexView WebView Auth Caching Cookies Raw

```
HTTP/1.1 200 OK
Date: Wed, 11 Jan 2017 18:24:08 GMT
Content-Type: text/plain
Server: Kestrel
Transfer-Encoding: chunked
Content-Encoding: gzip
Vary: Accept-Encoding
```

a
00000000
384
00L9
00}00d02000000^00000G]K*000rUW0Y0n0G0S000
0-000
XH0DV000Z \ycI\@0500Z00
00\$00000"0000000 X0QY00m0m\0000vNL0Bk80D00Q0200je000V0000R000000
0'0;000*0u_0a00000:@000{0
[0<0000*%_00000\$0~0r00u80Fu00a00F}0sXPY0(PG000/0h000N00(Sgy7000i20n005001000s" | 000*0#0z0~0009`o0tC00
0000;iGP*[000w[, [00000a#0010*fD0cj>~000副000Vn@_0VX000
001+f000^00*M3n0
0000x10!0a.x0&F0 00>YG00300Pdy0[000&(0' 000vg0R0毛Y @000z000k0000*일yI0000~t0yE0X00+v00i00v0i00s>00b0
000d0'np0q0vt0(010Nz/0000Y00o#
004W0050000G;00=050080000, 0&0000@
0Yn?00yY0G0x00000_0F0j0000ET0r夷Y.00V0?^00sx01/ш60&000X7ST0000d00;0000000f0ut00[000s0m0eL^0000+0k0+qu0
0000K0000i000q0rM0Zux}00敏)Q0<L0^<0t0]0h0:7vG?0Q_ofF00u0~S0id0Q000z1hn0 00
0

提供程序

GzipCompressionProvider

使用 `GzipCompressionProvider` 以压缩使用 gzip 响应。如果未指定，这是默认压缩提供程序。你可以设置压缩级别与 `GzipCompressionProviderOptions`。

Gzip 压缩提供程序默认为最快的压缩级别 (`CompressionLevel.Fastest`)，这可能不会产生最有效的压缩。如果需要最有效的压缩，则你可以配置的中间件理想的压缩。

压缩级别	描述
<code>CompressionLevel.Fastest</code>	即使未以最佳方式压缩生成的输出，应尽可能快地完成压缩。
<code>CompressionLevel.NoCompression</code>	应执行不进行压缩。
<code>CompressionLevel.Optimal</code>	响应应以最佳方式压缩，即使压缩操作将使用更多时间才能完成。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});
```

MIME 类型

该中间件指定一组默认的 MIME 类型为压缩：

- `text/plain`
- `text/css`
- `application/javascript`
- `text/html`
- `application/xml`
- `text/xml`
- `application/json`
- `text/json`

您可以替换或追加的响应压缩中间件选项的 MIME 类型。请注意该通配符 MIME 类型，如 `text/*` 不受支持。示例应用程序添加的 MIME 类型 `image/svg+xml` 和压缩，并提供横幅图像的 ASP.NET Core (`banner.svg`)。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```
services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});
```

自定义提供程序

你可以创建自定义压缩实现与 `ICompressionProvider`。`EncodingName` 表示的内容编码此 `ICompressionProvider` 生成。该中间件使用此信息来选择基于列表中指定的提供程序 `Accept-Encoding` 的请求标头。

使用示例应用程序，客户端提交的请求 `Accept-Encoding: mycustomcompression` 标头。该中间件使用自定义压缩的实现，并返回响应，其中 `Content-Encoding: mycustomcompression` 标头。客户端必须能够解压缩顺序用于工作的自定义压缩实现的自定义编码。

- [ASP.NET Core 2.x](#)
- [ASP.NET Core 1.x](#)

```

services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});

services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});

```

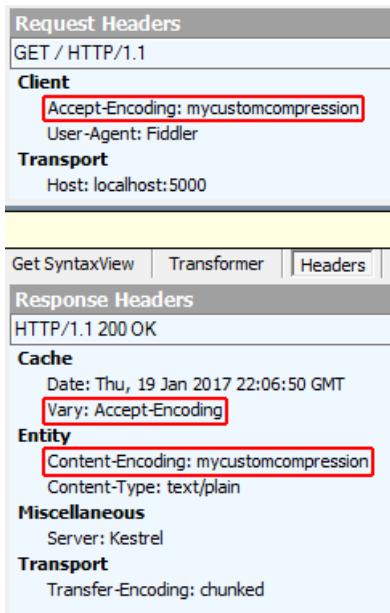
```

public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}

```

向示例应用程序与提交一个申请 `Accept-Encoding: mycustomcompression` 标头并观察响应标头。 `Vary` 和 `Content-Encoding` 标头会显示在响应。此示例未压缩响应正文（未显示）。没有在压缩的实现 `CustomCompressionProvider` 类中的示例。但是，此示例演示您可以用来实现此类的压缩算法。



使用安全的协议压缩

可以通过安全连接压缩的响应来控制 `EnableForHttps` 选项，它默认处于禁用状态。使用动态生成的页压缩可以导致安全问题如 [犯罪](#) 和 [违反](#) 攻击。

添加 Vary 标头

当压缩响应基于 `Accept-Encoding` 标头，有可能的多个压缩的版本响应和未压缩的版本。若要指示客户端和代理服务器缓存，多个版本存在，并且应存储 `Vary` 标头添加与 `Accept-Encoding` 值。在 ASP.NET Core 1.x 添加 `Vary` 手动完成到响应的标头。在 ASP.NET Core 2.x，中间件将添加 `Vary` 压缩响应时自动标头。

ASP.NET 核心 1.x 仅

```
// ONLY REQUIRED FOR ASP.NET CORE 1.x APPS
private void ManageVaryHeader(HttpContext context)
{
    // If the Accept-Encoding header is present, add the Vary header
    var accept = context.Request.Headers[HeaderNames.AcceptEncoding];
    if (!StringValues.IsNullOrEmpty(accept))
    {
        context.Response.Headers.Append(HeaderNames.Vary, HeaderNames.AcceptEncoding);
    }
}
```

后面 Nginx 反向代理时的中间件问题

当请求代理 nginx, `Accept-Encoding` 删除标头。这可以防止该中间件压缩响应。有关详细信息, 请参阅[NGINX: 压缩和解压缩](#)。此问题将跟踪[找出传递压缩 Nginx \(BasicMiddleware #123\)](#)。

使用 IIS 动态压缩

如果你有一个 active IIS 动态压缩模块你想要禁用的应用程序的服务器级别配置, 可以进行此操作而添加到你*web.config*文件。有关详细信息, 请参阅[禁用 IIS 模块](#)。

疑难解答

使用之类的工具[Fiddler](#), [Firebug](#), 或[Postman](#), 这允许你设置 `Accept-Encoding` 请求标头并研究响应标头、大小和正文。响应压缩中间件压缩满足以下条件的响应:

- `Accept-Encoding` 标头的值位于 `gzip`, `*`, 或与匹配的自定义压缩提供程序已建立的自定义编码。值不能 `identity` 或具有质量值 (`qvalue`, `q`) 设置为 0 (零)。
- MIME 类型 (`Content-Type`) 必须设置, 并且必须匹配上配置的 MIME 类型 `ResponseCompressionOptions`。
- 请求不得包括 `Content-Range` 标头。
- 请求必须使用不安全的协议 (http), 除非在响应压缩中间件选项中配置安全协议 (https)。请注意危险[上述启用安全的内容压缩时](#)。

其他资源

- [应用程序启动](#)
- [中间件](#)
- [Mozilla 开发人员网络: 接受的编码](#)
- [RFC 7231 部分 3.1.2.1: 内容 Codings](#)
- [RFC 7230 部分 4.2.3: Gzip 编码](#)
- [GZIP 文件格式规范版本 4.3](#)

迁移到 ASP.NET Core

2018/5/14 • 1 min to read • [Edit Online](#)

ASP.NET 到 ASP.NET Core

- [从 ASP.NET 迁移到 ASP.NET Core](#)
- [从 ASP.NET MVC 迁移到 ASP.NET Core MVC](#)
- [从 ASP.NET Web API 迁移到 ASP.NET Core Web API](#)
- [迁移配置](#)
- [迁移身份验证和标识](#)
- [迁移 ClaimsPrincipal.Current 使用](#)
- [将 ASP.NET 成员身份迁移到 ASP.NET Core 标识](#)
- [将 HTTP 模块迁移到中间件](#)

ASP.NET Core 1.x 到 2.0

- [从 ASP.NET Core 1.x 迁移到 2.0](#)
- [迁移身份验证和标识](#)

从 ASP.NET 迁移到 ASP.NET Core

2018/5/14 • 8 min to read • [Edit Online](#)

作者 : [Isaac Levin](#)

本文可作为从 ASP.NET 应用迁移到 ASP.NET Core 的参考指南。

系统必备

.NET Core SDK 2.0 or later

目标框架

ASP.NET Core 项目为开发人员提供了面向 .NET Core 和/或 .NET Framework 的灵活性。若要确定最合适的目标框架, 请参阅[为服务器应用选择 .NET Core 或 .NET Framework](#)。

面向 .NET Framework 时, 项目需要引用单个 NuGet 包。

得益于有 ASP.NET Core 元包, 面向 .NET Core 时可以避免进行大量的显式包引用。在项目中安装

`Microsoft.AspNetCore.All` 元包:

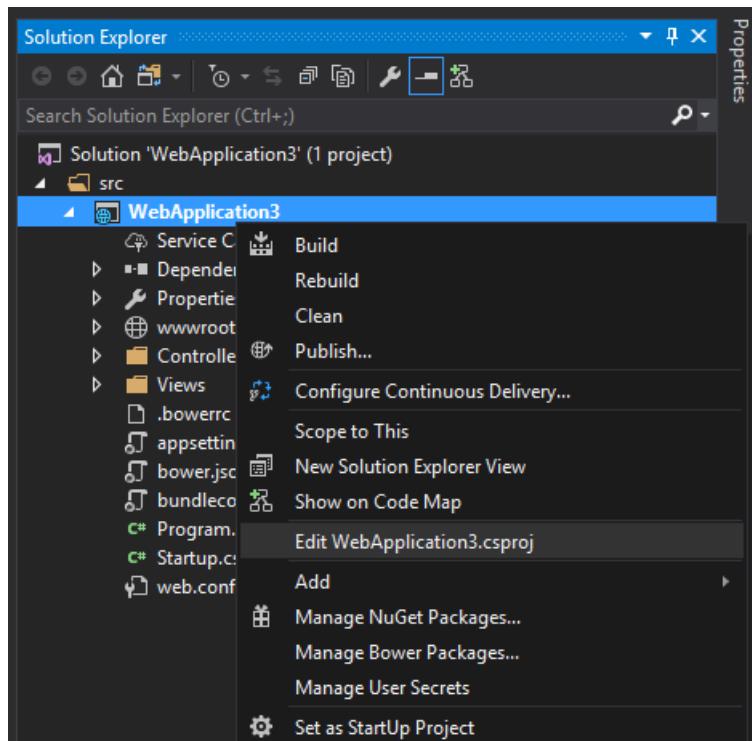
```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
```

使用此元包时, 应用不会部署元包中引用的任何包。.NET Core 运行时存储中包含这些资产, 并已预编译, 旨在提升性能。请参阅[ASP.NET Core 2.x 的 Microsoft.AspNetCore.All 元包](#)了解详细信息。

项目结构差异

ASP.NET Core 中简化了 .csproj 文件格式。下面是一些显著的更改:

- 无需显式添加, 即可将文件视作项目的一部分。服务于大型团队时, 这可减少出现 XML 合并冲突的风险。
- 没有对其他项目的基于 GUID 的引用, 这可以提高文件的可读性。
- 无需在 Visual Studio 中卸载文件即可对它进行编辑:



Global.asax 文件替换

ASP.NET Core 引入了启动应用的新机制。ASP.NET 应用程序的入口点是 Global.asax 文件。路由配置及筛选器和区域注册等任务在 Global.asax 文件中进行处理。

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

此方法会将应用程序和应用程序要部署到的服务器耦合在一起，并且它们的耦合方式会干扰实现。为了将它们分离，引入了 [OWIN](#) 来提供一种更为简便的同时使用多个框架的方法。OWIN 提供了一个管道，可以只添加所需的模块。托管环境使用 [Startup](#) 函数配置服务和应用的请求管道。[Startup](#) 在应用程序中注册一组中间件。对于每个请求，应用程序都使用现有处理程序集的链接列表的头指针调用各个中间件组件。每个中间件组件可以向请求处理管道添加一个或多个处理程序。为此，需要返回对成为列表新头的处理程序的引用。每个处理程序负责记住并调用列表中的下一个处理程序。使用 ASP.NET Core 时，应用程序的入口点是 [Startup](#)，不再具有 Global.asax 的依赖关系。结合使用 OWIN 和 .NET Framework 时，使用的管道应如下所示：

```

using Owin;
using System.Web.Http;

namespace WebApi
{
    // Note: By default all requests go through this OWIN pipeline. Alternatively you can turn this off by
    // adding an appSetting owin:AutomaticAppStartup with value "false".
    // With this turned off you can still have OWIN apps listening on specific routes by adding routes in
    global.asax file using MapOwinPath or MapOwinRoute extensions on RouteTable.Routes
    public class Startup
    {
        // Invoked once at startup to configure your application.
        public void Configuration(IAppBuilder builder)
        {
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute("Default", "{controller}/{customerID}", new { controller = "Customer",
customerID = RouteParameter.Optional });

            config.Formatters.XmlFormatter.UseXmlSerializer = true;
            config.Formatters.Remove(config.Formatters.JsonFormatter);
            // config.Formatters.JsonFormatter.UseDataContractJsonSerializer = true;

            builder.UseWebApi(config);
        }
    }
}

```

这会配置默认路由，默認為 XMLSerialization 而不是 JSON。根据需要向此管道添加其他中间件（加载服务、配置设置、静态文件等）。

ASP.NET Core 使用相似的方法，但是不依赖 OWIN 处理条目。相反，这是通过 Program.cs `Main` 方法（类似于控制台应用程序）完成，其中加载了 `Startup`。

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication2
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

`Startup` 必须包含 `Configure` 方法。在 `Configure` 中，向管道添加必要的中间件。在下列示例（位于默认的网站模板）中，使用了多个扩展方法为管道配置对以下内容的支持：

- [BrowserLink](#)
- 错误页
- 静态文件
- ASP.NET Core MVC
- 标识

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

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

现在主机和应用程序已分离，这样将来就可以灵活地迁移到其他平台。

注意

若要获取 ASP.NET Core Startup 和中间件的更深入的参考信息，请参阅 [ASP.NET Core 中的 Startup](#)

存储配置

ASP.NET 支持存储设置。这些设置可用于支持应用程序已部署到的环境（以此用途为例）。常见做法是将所有的自定义键值对存储在 Web.config 文件的 `<appSettings>` 部分中：

```
<appSettings>
    <add key="UserName" value="User" />
    <add key="Password" value="Password" />
</appSettings>
```

应用程序使用 `System.Configuration` 命名空间中的 `ConfigurationManager.AppSettings` 集合读取这些设置：

```
string userName = System.Web.Configuration.ConfigurationManager.AppSettings["UserName"];
string password = System.Web.Configuration.ConfigurationManager.AppSettings["Password"];
```

ASP.NET Core 可以将应用程序的配置数据存储在任何文件中，并可在启动中间件的过程中加载它们。项目模板中使用的默认文件是 `appsettings.json`：

```
{  
    "Logging": {  
        "IncludeScopes": false,  
        "LogLevel": {  
            "Default": "Debug",  
            "System": "Information",  
            "Microsoft": "Information"  
        }  
    },  
    // Here is where you can supply custom configuration settings, Since it is JSON, everything is  
    // represented as key: value pairs  
    // Name of section is your choice  
    "AppConfiguration": {  
        "UserName": "UserName",  
        "Password": "Password"  
    }  
}
```

将此文件加载到应用程序内的 `IConfiguration` 的实例的过程在 `Startup.cs` 中完成：

```
public Startup(IConfiguration configuration)  
{  
    Configuration = configuration;  
}  
  
public IConfiguration Configuration { get; }
```

应用读取 `Configuration` 来获得这些设置：

```
string userName = Configuration.GetSection("AppConfiguration")["UserName"];  
string password = Configuration.GetSection("AppConfiguration")["Password"];
```

此方法有扩展项，它们可使此过程更加可靠，例如使用[依存关系注入 \(DI\)](#) 来加载使用这些值的服务。DI 方法提供了一组强类型的配置对象。

```
// Assume AppConfiguration is a class representing a strongly-typed version of AppConfiguration section  
services.Configure<AppConfiguration>(Configuration.GetSection("AppConfiguration"));
```

注意

若要获取 ASP.NET Core 配置的更深入的参考信息，请参阅 [ASP.NET Core 中的配置](#)。

本机依存关系注入

生成大型可缩放应用程序时，一个重要的目标是将组件和服务松散耦合。[依赖项注入](#)不仅是可实现此目标的常用技术，还是 ASP.NET Core 的本机组件。

在 ASP.NET 应用中，开发人员依赖第三方库实现依存关系注入。其中的一个库是 Microsoft 模式和做法提供的 [Unity](#)。

实现打包 `UnityContainer` 的 `IDependencyResolver` 是使用 Unity 设置依存关系注入的一个示例：

```

using Microsoft.Practices.Unity;
using System;
using System.Collections.Generic;
using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver
{
    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }
        this.container = container;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return container.Resolve(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return container.ResolveAll(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return new List<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        var child = container.CreateChildContainer();
        return new UnityResolver(child);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected virtual void Dispose(bool disposing)
    {
        container.Dispose();
    }
}

```

创建 `UnityContainer` 的实例，注册服务，然后将 `HttpConfiguration` 的依赖关系解析程序设置为容器的 `UnityResolver` 新实例：

```
public static void Register(HttpConfiguration config)
{
    var container = new UnityContainer();
    container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());
    config.DependencyResolver = new UnityResolver(container);

    // Other Web API configuration not shown.
}
```

在必要时注入 `IProductRepository`：

```
public class ProductsController : ApiController
{
    private IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    // Other controller methods not shown.
}
```

由于依存关系注入是 ASP.NET Core 的组成部分，因此可以在 `Startup.cs` 的 `ConfigureServices` 方法中添加你的服务：

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IProductRepository, ProductRepository>();
}
```

可在任意位置注入存储库，Unity 亦是如此。

注意

若要获取 ASP.NET Core 中的依存关系注入的深入的参考信息，请参阅 [ASP.NET Core 中的依存关系注入](#)

提供静态文件

Web 开发的一个重要环节是提供客户端静态资产的功能。HTML、CSS、Javascript 和图像是最常见的静态文件示例。这些文件需要保存在应用(或 CDN)的发布位置中，并且需要引用它们，以便请求可以加载这些文件。在 ASP.NET Core 中，此过程发生了变化。

在 ASP.NET 中，静态文件存储在各种目录中，并在视图中进行引用。

在 ASP.NET Core 中，静态文件存储在“Web 根”(<内容根>/wwwroot)中，除非另有配置。通过从 `Startup.Configure` 调用 `UseStaticFiles` 扩展方法将这些文件加载到请求管道中：

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

注意

如果面向 .NET Framework, 请安装 NuGet 包 `Microsoft.AspNetCore.StaticFiles`。

例如, 可以通过浏览器从类似 `http://<app>/images/<imageFileName>` 的位置访问 wwwroot/images 文件夹中的图像资产。

注意

若要获取在 ASP.NET Core 中提供静态文件的更深入的参考信息, 请参阅[静态文件](#)。

其他资源

- [将库移植到 .NET Core](#)

将从 ASP.NET MVC 迁移到 ASP.NET 核心 MVC

2018/5/14 • 9 min to read • [Edit Online](#)

通过[Rick Anderson](#), [Daniel Roth](#), [Steve Smith](#), 和[Scott Addie](#)

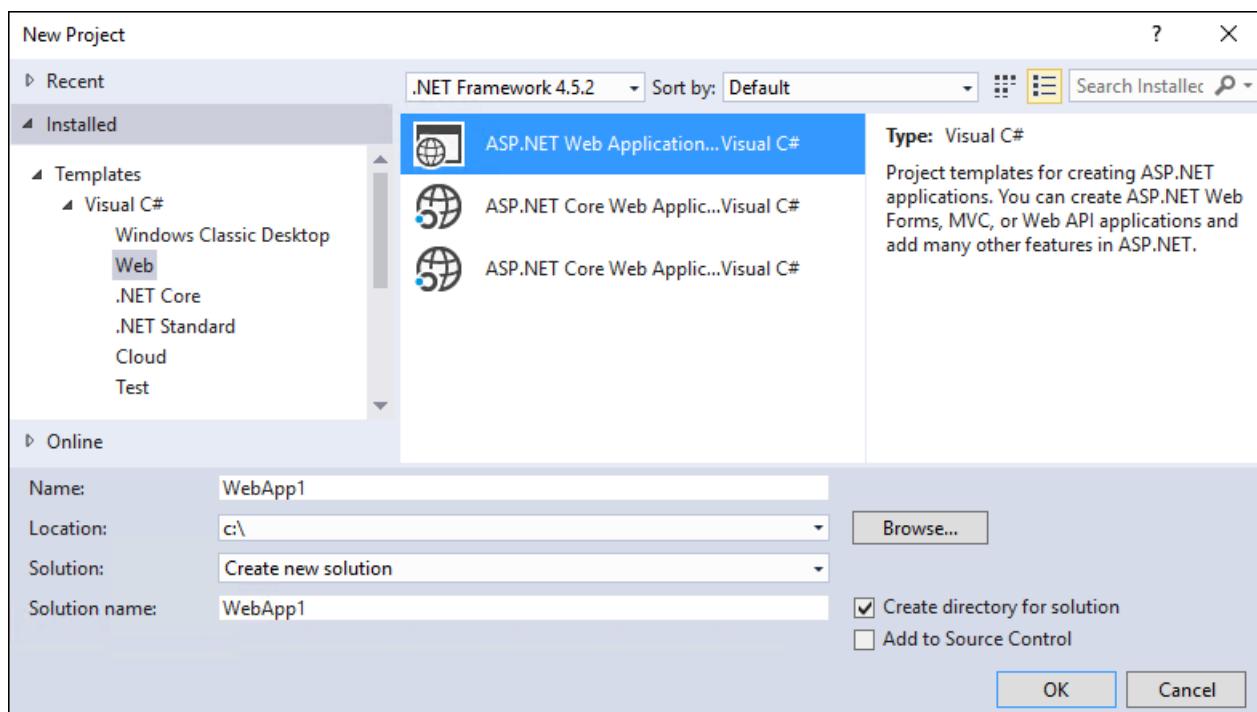
这篇文章演示如何开始迁移到 ASP.NET MVC 项目[ASP.NET 核心 MVC](#)。在过程中, 会突出显示许多已更改利用 ASP.NET MVC 的内容。从 ASP.NET MVC 迁移是一个多步骤过程, 本文涵盖初始安装程序、基本控制器和视图、静态内容和客户端依赖关系。其他文章涵盖迁移配置和标识代码在许多 ASP.NET MVC 项目中找到。

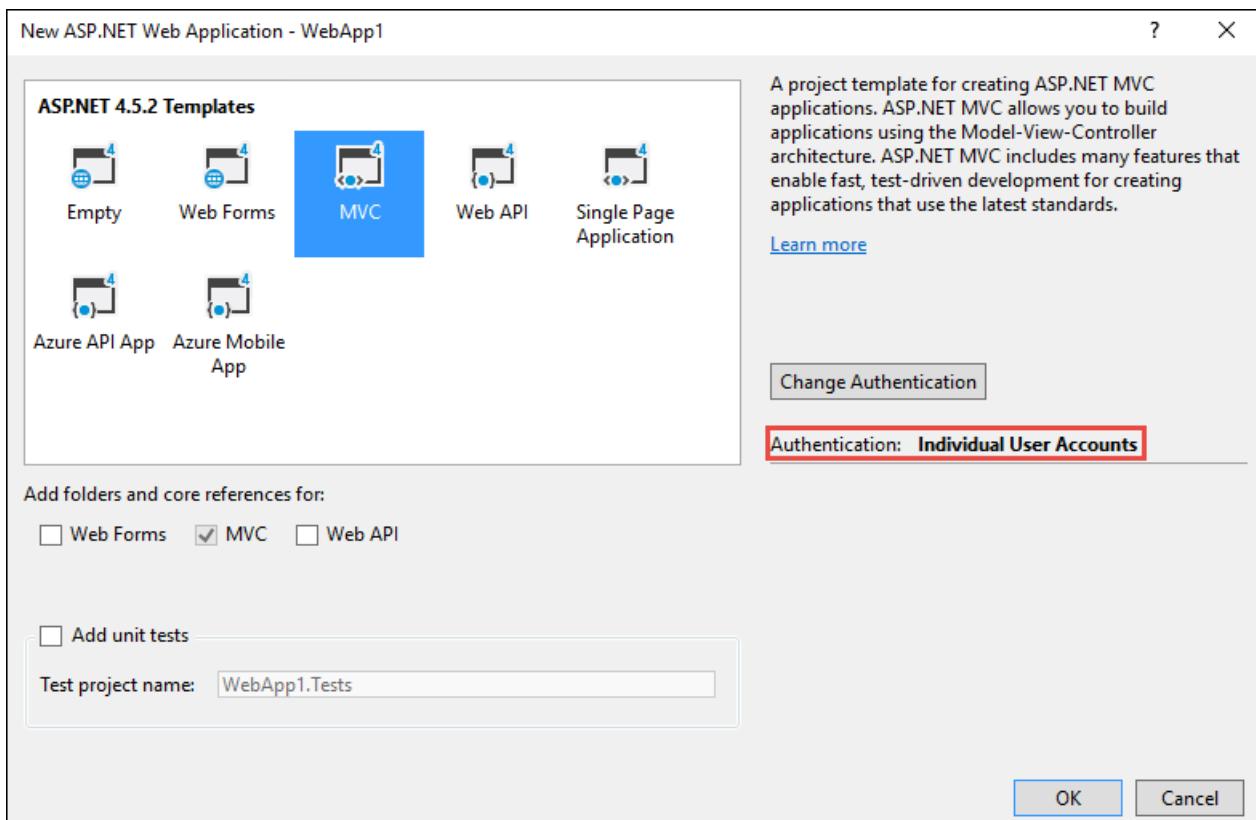
注意

这些示例中的版本号可能不是最新。你可能需要相应地更新你的项目。

创建初学者 ASP.NET MVC 项目

为了演示升级, 我们将开始通过创建 ASP.NET MVC 应用程序。创建同名 *WebApp1* 使命名空间匹配我们在下一步中创建 ASP.NET 核心项目。

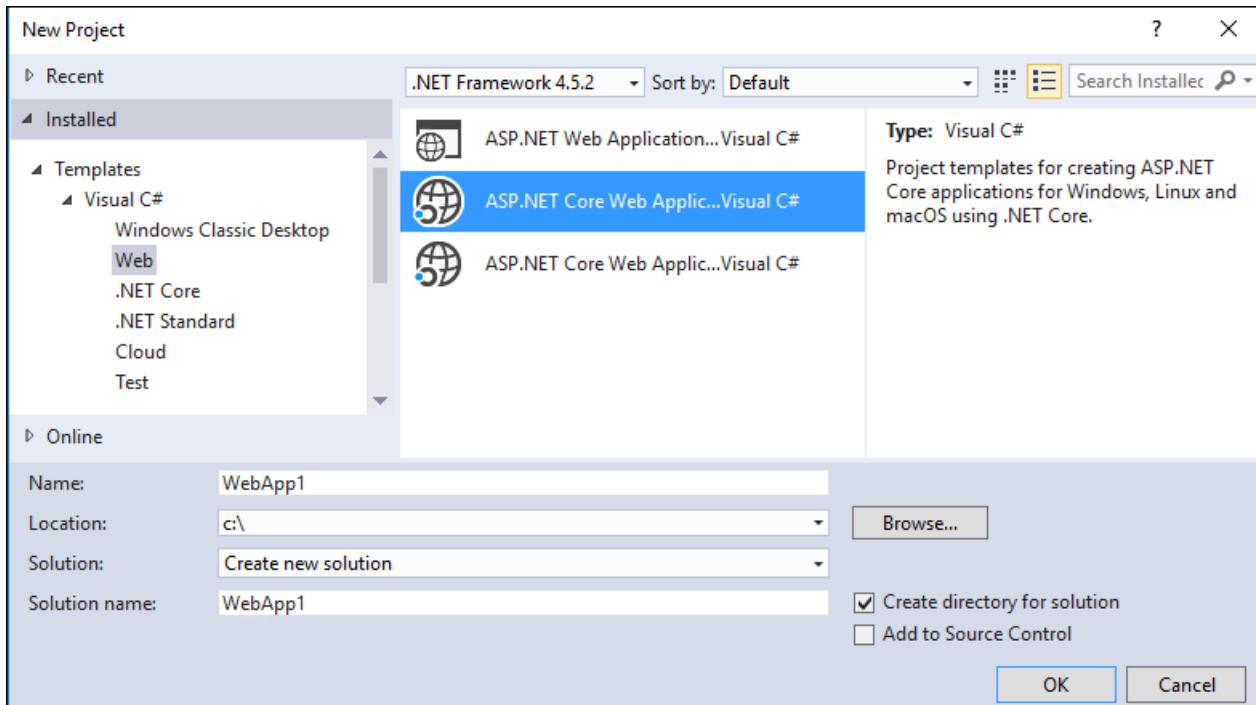


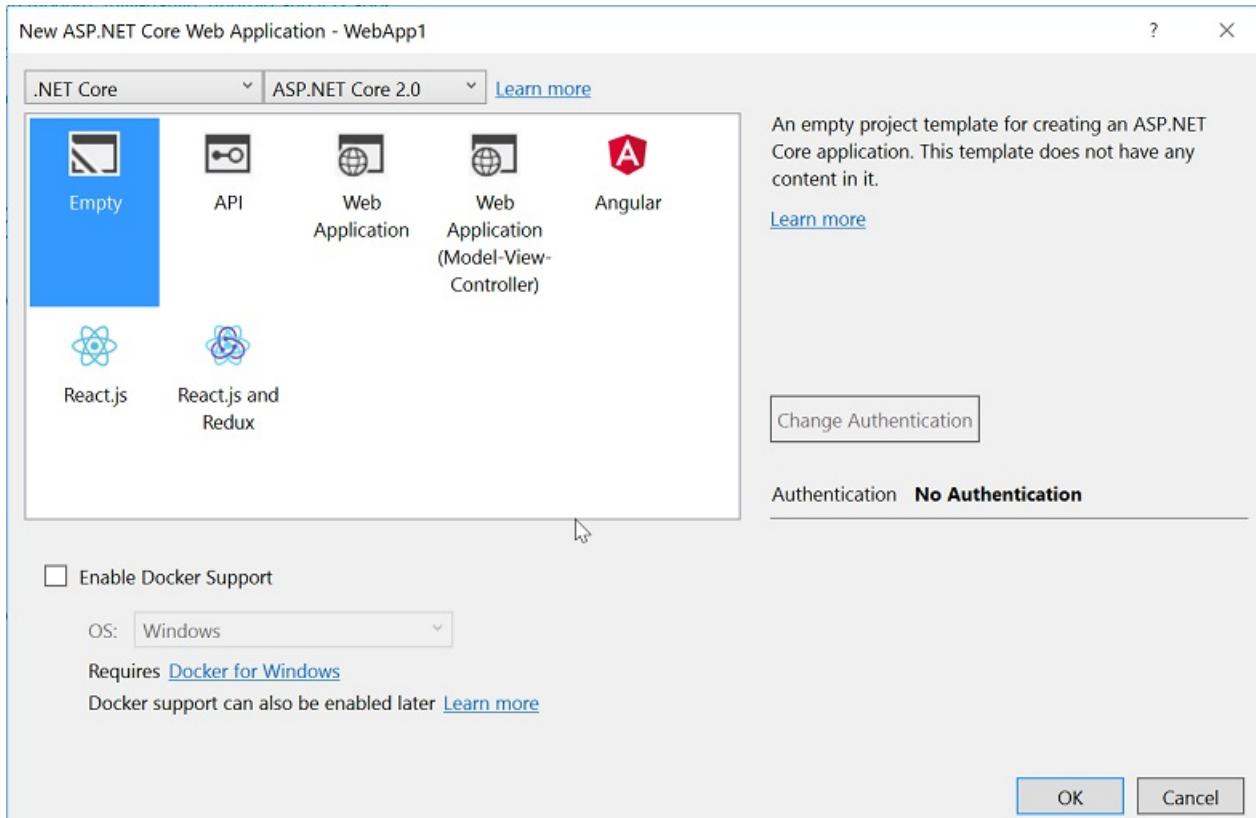


可选: 更改从解决方案的名称*WebApp1*到*Mvc5*。Visual Studio 将显示新的解决方案名称 (*Mvc5*)，从而更轻松地判断此项目从下一步的项目。

创建 ASP.NET Core 项目

创建一个新空ASP.NET Core与以前的项目同名的web应用(`WebApp1`)以便将两个项目中的命名空间匹配。使用相同的命名空间，可更轻松地复制两个项目之间的代码。你将需要在比以前的项目以使用相同的名称不同的目录中创建此项目。





- 可选: 创建新的 ASP.NET Core 应用使用Web 应用程序项目模板。将项目 `WebApp1`, 然后选择的一个身份验证选项单个用户帐户。重命名此应用程序到 `FullAspNetCore`。在转换过程中创建此项目为您节省时间。你可以查看若要查看的最终结果或将代码复制到转换项目模板生成的代码。它也是很有帮助时遇到困难在转换步骤中, 要与模板生成项目进行比较。

配置站点以使用 MVC

- 如果目标.NET 核心, 将 ASP.NET Core metapackage 添加到项目中, 调用 `Microsoft.AspNetCore.All` 默认情况下。此程序包包含如 `Microsoft.AspNetCore.Mvc` 和 `Microsoft.AspNetCore.StaticFiles`。如果面向.NET Framework, 包将引用需要 *.csproj 文件中单独列出。

`Microsoft.AspNetCore.Mvc` 是 ASP.NET 核心 MVC 框架。`Microsoft.AspNetCore.StaticFiles` 是静态文件处理程序。ASP.NET 核心运行时是一个模块化, 和中, 你必须显式选择要为静态文件服务(请参阅[静态文件](#))。

- 打开 `Startup.cs` 文件并将更改代码以匹配以下内容:

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

namespace WebApp1
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit
        https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

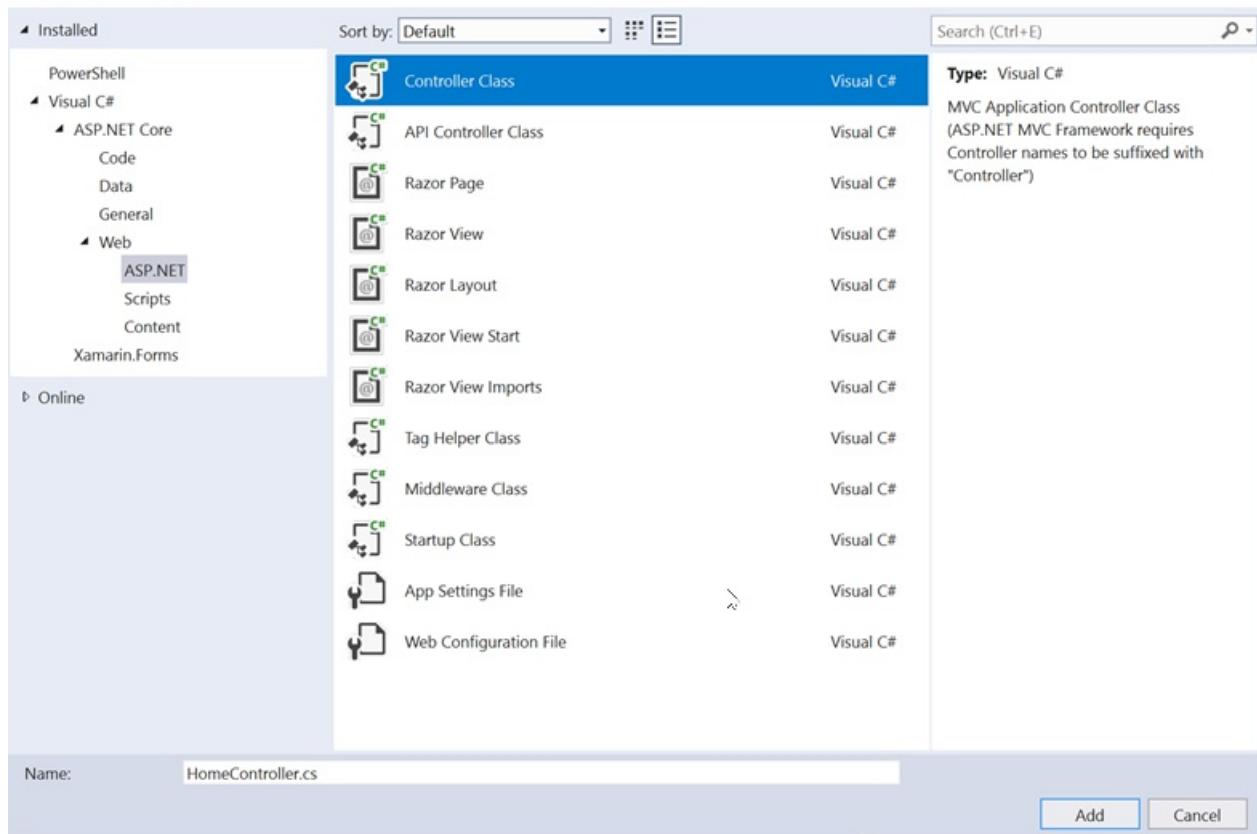
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

`UseStaticFiles` 扩展方法将添加静态文件处理程序。如前所述，ASP.NET 运行时是一个模块化，和中，你必须显式选择要为静态文件服务。`UseMvc` 扩展方法将添加路由。有关详细信息，请参阅[应用程序启动](#)和[路由](#)。

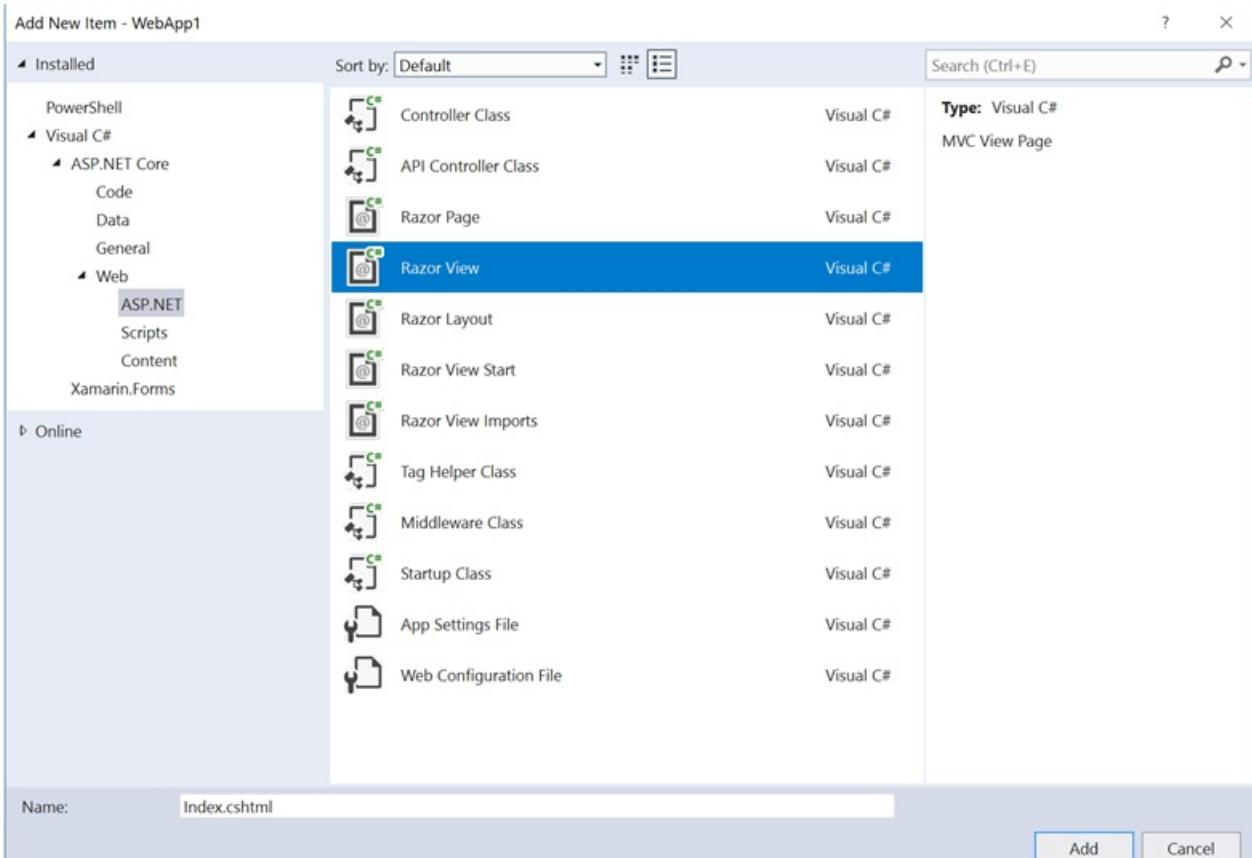
添加控制器和视图

在本部分中，你将添加一个最小控制器和视图，以用作占位符的 ASP.NET MVC 控制器和视图将在下一部分中迁移。

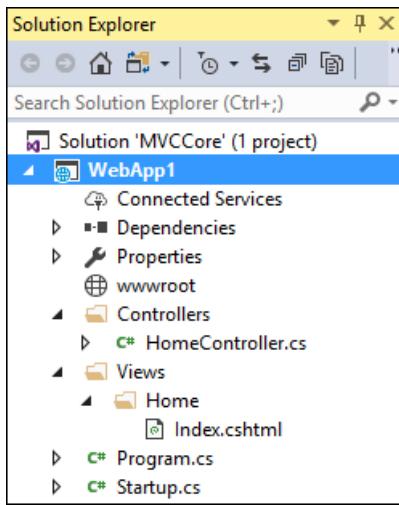
- 添加控制器文件夹。
- 添加控制器类名为*HomeController.cs*到控制器文件夹。



- 添加视图文件夹。
- 添加视图/主页文件夹。
- 添加Razor 视图名为`Index.cshtml`到视图/主页文件夹。



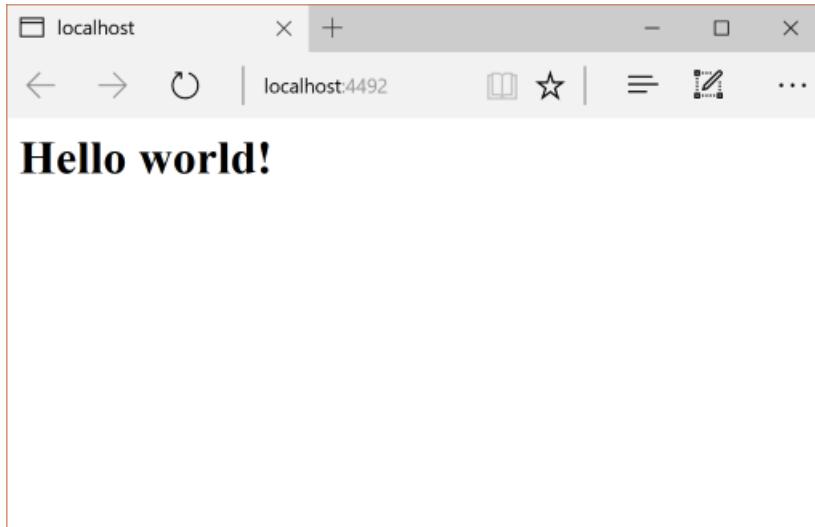
项目结构如下所示：



内容替换*Views/Home/Index.cshtml*替换为以下文件：

```
<h1>Hello world!</h1>
```

运行应用。



请参阅[控制器](#)和[视图](#)有关详细信息。

现在，我们已最小的工作 ASP.NET Core 项目，我们可以开始从 ASP.NET MVC 项目迁移功能。我们需要将以下：

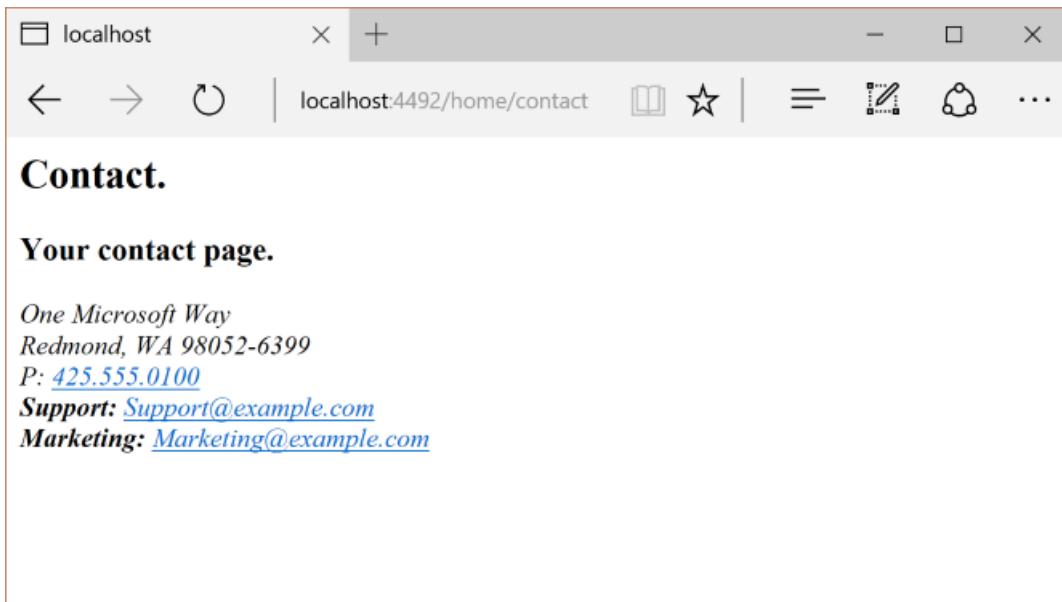
- 客户端内容（CSS、字体和脚本）
- 控制器
- 视图
- 模型
- 绑定
- 筛选器
- 输入/输出的日志、标识（这是在下一步的教程。）

控制器和视图

- 将每个方法复制利用 ASP.NET MVC `HomeController` 对新 `HomeController`。请注意，在 ASP.NET MVC 内置模板的控制器操作方法的返回类型 `ActionResult`；在 ASP.NET 核心 MVC，操作方法返回 `IActionResult` 相

反。`ActionResult` 实现 `IActionResult`，因此无需更改你的操作方法的返回类型。

- 复制 `About.cshtml`, `Contact.cshtml`, 和 `Index.cshtml` Razor 视图文件从 ASP.NET MVC 项目添加到 ASP.NET 核心项目。
- 运行 ASP.NET Core 应用和测试每个方法。我们尚未尚未进行迁移的布局文件或样式，因此呈现的视图仅包含视图文件中的内容。不会有的布局生成文件链接 `About` 和 `Contact` 视图，因此你将需要从浏览器中调用它们(替换**4492**与你的项目中使用的端口号)。
 - `http://localhost:4492/home/about`
 - `http://localhost:4492/home/contact`



请注意在缺乏样式和菜单项。此问题将在下一部分得以解决。

静态内容

在以前版本的 ASP.NET MVC，静态内容已承载 web 项目的根目录中，已使用服务器端文件混合。在 ASP.NET Core 静态内容承载于 `wwwroot` 文件夹。你将想要复制的静态内容从旧 ASP.NET MVC 应用程序到 `wwwroot` ASP.NET Core 项目文件夹中的。在此示例转换：

- 复制 `favicon.ico` 文件从旧的 MVC 项目到 `wwwroot` ASP.NET Core 项目文件夹中的。

旧 ASP.NET MVC 项目使用 `Bootstrap` 其 `Bootstrap` 文件中的样式和存储 `content` 和 `fonts` 文件夹。该模板后，生成旧的 ASP.NET MVC 项目，该引用布局文件中的 `Bootstrap` (`Views/Shared/_Layout.cshtml`)。你也可以将复制 `bootstrap.js` 和 `bootstrap.css` 文件从 ASP.NET MVC 项目合并为 `wwwroot` 在新项目中的文件夹。相反，我们将添加 `Bootstrap` 的支持 (和其他客户端库)，它在下一节中使用 `Cdn`。

迁移布局文件

- 复制 `_ViewStart.cshtml` 文件从旧的 ASP.NET MVC 项目 `视图` 文件夹导入到 ASP.NET 核心项目 `视图` 文件夹。`_ViewStart.cshtml` 文件未更改 ASP.NET 核心 mvc。
- 创建 `视图/共享` 文件夹。
- 可选：复制 `_ViewImports.cshtml` 从 `FullAspNetCore` MVC 项目 `视图` 文件夹导入到 ASP.NET 核心项目 `视图` 文件夹。删除中的任何命名空间声明 `_ViewImports.cshtml` 文件。`_ViewImports.cshtml` 文件对于视图的所有文件提供命名空间，并使 `标记帮助程序`。新的布局文件中使用标记帮助程序。`_ViewImports.cshtml` 文件是用于 ASP.NET 核心新功能。
- 复制 `_Layout.cshtml` 文件从旧的 ASP.NET MVC 项目 `视图/共享` 文件夹导入到 ASP.NET 核心项目 `视图/共享`

享文件夹。

打开 `_Layout.cshtml` 文件并进行以下更改 (已完成的代码下面显示) :

- 替换 `@Styles.Render("~/Content/css")` 与 `<link>` 元素加载 `bootstrap.css` (见下文)。
- 删除 `@Scripts.Render("~/bundles/modernizr")`。
- 注释掉 `@Html.Partial("_LoginPartial")` 行 (括在一行时与 `@*...*@`)。在将来的教程, 我们会返回到它。
- 替换 `@Scripts.Render("~/bundles/jquery")` 与 `<script>` 元素 (见下文)。
- 替换 `@Scripts.Render("~/bundles/bootstrap")` 与 `<script>` 元素 (见下文)。

Bootstrap CSS 包含替换标记:

```
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
      integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
      crossorigin="anonymous">
```

JQuery 和 Bootstrap JavaScript 包含的替换标记:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
      integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8wGNiCpd7Txa"
      crossorigin="anonymous"></script>
```

已更新 `_Layout.cshtml` 文件如下所示:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
        integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
        crossorigin="anonymous">
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @* @Html.Partial("_LoginPartial") *@
            </div>
        </div>
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
            </footer>
        </div>
        <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8wGNiCpd7Txa"
            crossorigin="anonymous"></script>
        @RenderSection("scripts", required: false)
    </body>
</html>

```

在浏览器中查看站点。它应现在正确加载，以就地预期的样式。

- 可选：可能想要尝试使用新的布局文件。对于此项目中，你可以复制中的布局文件FullAspNetCore项目。新的布局文件使用[标记帮助程序](#)并且具有其他的改进功能。

配置绑定和缩减

有关如何配置绑定和缩减的信息，请参阅[捆绑和缩减](#)。

解决 HTTP 500 错误

有许多问题的包含没有关于此问题的源的信息可能会导致 HTTP 500 错误消息。例如，如果Views/_ViewImports.cshtml文件包含在你的项目中不存在的命名空间，你将收到 HTTP 500 错误。默认情况下，

在 ASP.NET Core 应用中，`UseDeveloperExceptionPage` 扩展添加到 `IApplicationBuilder` 和执行在配置后开发。这将在下面的代码中详细说明：

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

namespace WebApp1
{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to add services to the container.
        // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

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

ASP.NET 核心将在 web 应用程序中未经处理的异常转换为 HTTP 500 错误响应。通常情况下，错误详细信息不包括在这些响应来防止有关服务器的潜在敏感信息被披露。请参阅[使用开发人员异常页中处理错误](#)有关详细信息。

其他资源

- [客户端开发](#)
- [标记帮助程序](#)

将从 ASP.NET Web API 迁移到 ASP.NET 核心

2018/5/14 • 9 min to read • [Edit Online](#)

作者:Steve Smith 和 Scott Addie

Web API 是覆盖广泛的客户端, 包括浏览器和移动设备的 HTTP 服务。ASP.NET 核心 MVC 包括用于构建提供构建 web 应用程序的单个、一致的方式的 Web API 的支持。在本文中, 我们将演示将从 ASP.NET Web API 的 Web API 实现迁移到 ASP.NET 核心 MVC 所需的步骤。

[查看或下载示例代码\(如何下载\)](#)

检查 ASP.NET Web API 项目

本文章将使用示例项目中, *ProductsApp* 文章中创建 [Getting Started with ASP.NET Web API 2](#) 作为其起点。在该项目中, 一个简单的 ASP.NET Web API 项目, 如下所示配置。

在 *Global.asax.cs*, 调用了 `WebApiConfig.Register`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

`WebApiConfig` 在中定义 `App_Start`, 并具有一个静态 `Register` 方法:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}

```

此类配置的属性路由，不过它实际上并没有用于在项目中。它还将配置路由表，它由 ASP.NET Web API。在这种情况下，ASP.NET Web API 应 Url 以匹配格式 `/api/{controller} /{id}`，与 `{id}` 正在可选。

`ProductsApp` 项目包括一个简单的控制器，它继承自 `ApiController` 和公开两个方法：

```

using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}

```

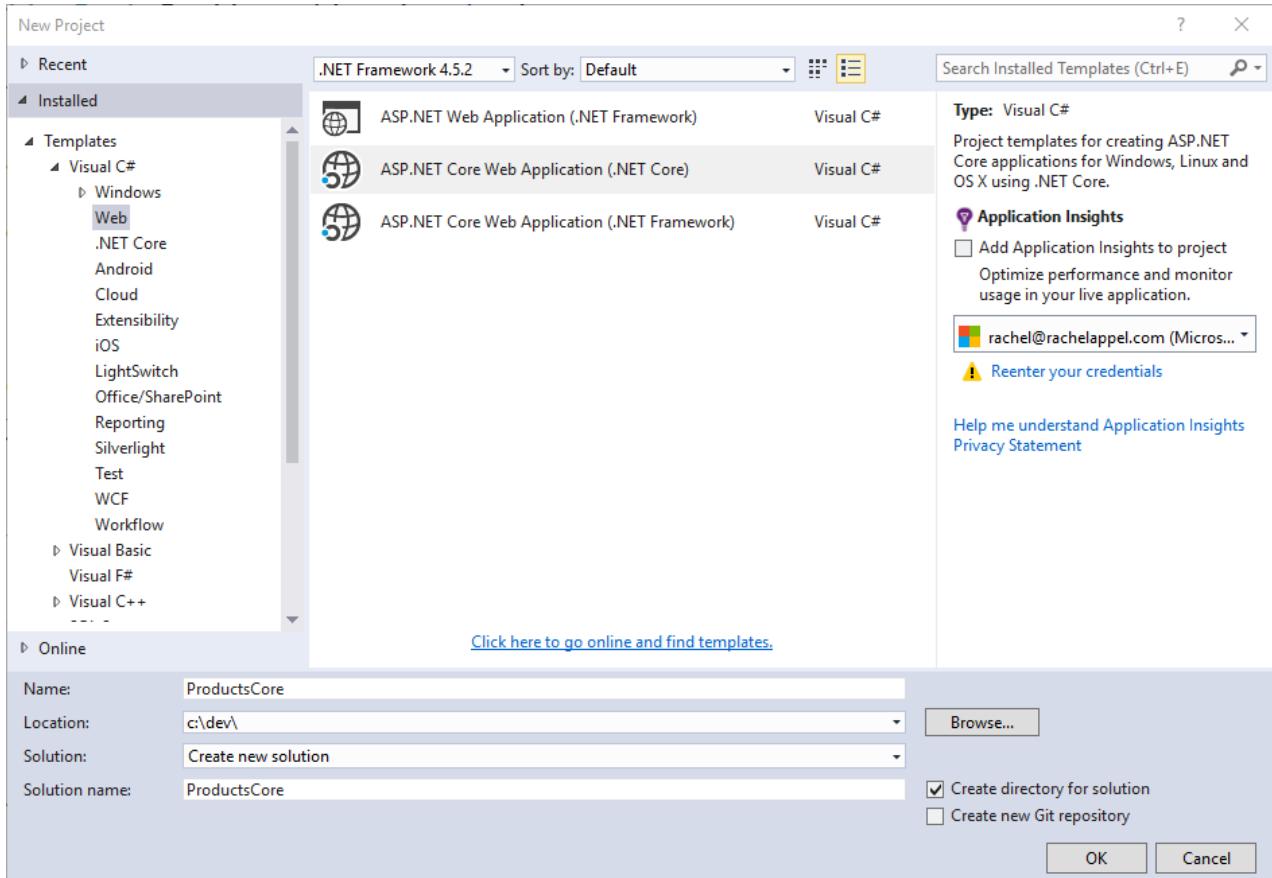
最后，模型产品、所用的*ProductsApp*，是一个简单的类：

```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

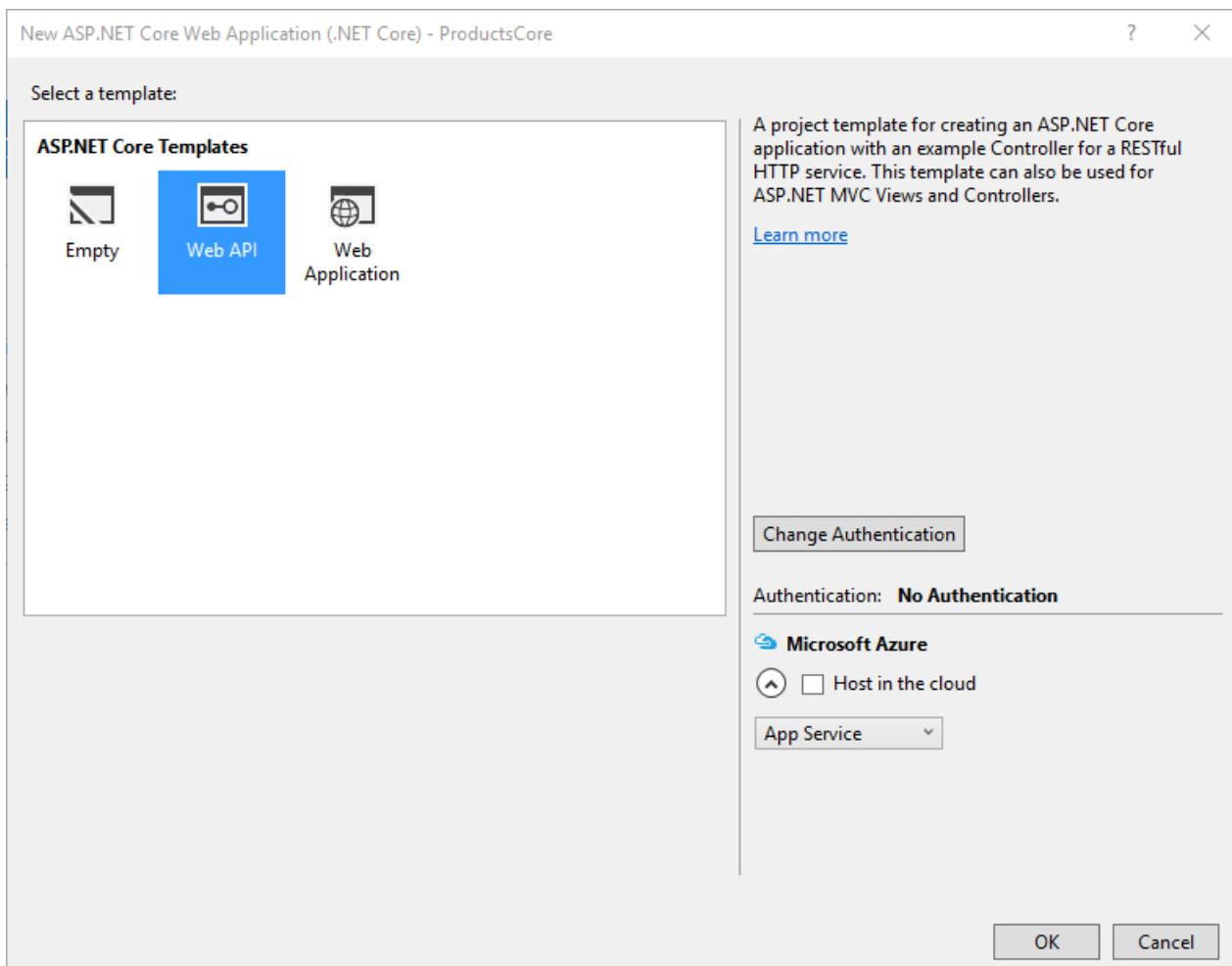
现在，我们已从其开始一个简单的项目，我们可以演示如何将此 Web API 项目迁移到 ASP.NET 核心 MVC。

创建目标项目

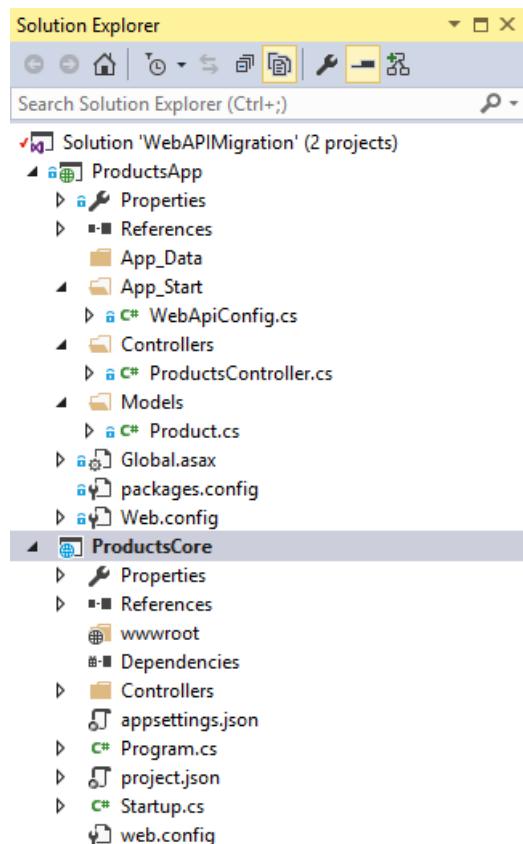
使用 Visual Studio，创建一个新的空解决方案，并将其命名*WebAPIMigration*。添加现有*ProductsApp*到其中，项目，然后将新的 ASP.NET 核心 Web 应用程序项目添加到解决方案。将新项目*ProductsCore*。



接下来，选择 Web API 项目模板。我们会将迁移*ProductsApp*到此新项目的内容。



删除 Project_Readme.html 文件从新项目。你的解决方案现在应如下所示：



迁移配置

ASP.NET 核心不再使用 *Global.asax*, *web.config*, 或 *App_Start* 文件夹。相反, 所有启动任务都在都完成 *Startup.cs* 项目的根目录中 (请参阅[应用程序启动](#))。在 ASP.NET 核心 MVC, 基于属性的路由现在包含默认情况下时 `UseMvc()` 称为; 并且, 这是建议的配置 Web API 的路由的方法 (Web API 初学者项目处理路由的方式)。

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

namespace ProductsCore
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            // Add framework services.
            services.AddMvc();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole(Configuration.GetSection("Logging"));
            loggerFactory.AddDebug();

            app.UseMvc();
        }
    }
}
```

假设你想要使用你今后的项目中的属性路由, 不需进行任何其他配置。只需将应用的属性, 根据需要向你的控制器和操作, 在此示例中的做法一样 `ValuesController` Web API 初学者项目中包含的类:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace ProductsCore.Controllers
{
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        // GET api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public string Get(int id)
        {
            return "value";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody]string value)
        {
        }

        // PUT api/values/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}

```

请注意是否存在 `[控制器]` 第 8 行。基于属性的路由现在支持某些令牌，如 `[控制器]` 和 `[操作]`。这些标记分别被替换为在运行时的控制器或操作名称，已向其应用该属性。此选项可用于减少了大量神奇字符串在项目中，并确保路由将保留其相应的控制器和操作与同步时自动重命名重构应用。

若要迁移的产品的 API 控制器，我们必须首先将复制 `ProductsController` 到新项目。然后只需包括在控制器上的 `route` 属性：

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

你还需要添加 `[HttpGet]` 属性设为两种方法，因为它们都应通过 HTTP Get 调用。有关特性中包括的“`id`”参数的假定条件下 `GetProduct()`：

```
// /api/products  
[HttpGet]  
...  
  
// /api/products/{id}  
[HttpGet("{id}")]
```

此时，路由配置正确；但是，我们无法一尚未将其测试。必须在之前进行其他更改 *ProductsController* 将进行编译。

迁移模型和控制器

此简单的 Web API 项目的迁移过程的最后一步是将复制到控制器，它们使用的所有模型。在这种情况下，只需复制 *Controllers/ProductsController.cs* 从原始到新项目。然后，将整个模型文件夹从原始项目复制到新的一个。调整要与新的项目名称匹配的命名空间 (*ProductsCore*)。此时，可以生成应用程序，并且你将找到大量的编译错误。这些通常应属于以下类别：

- *ApiController* 不存在
- *System.Web.Http* 命名空间不存在
- *IHttpActionResult* 不存在

幸运的是，这些是所有很容易更正：

- 更改 *ApiController* 到 控制器 (可能需要添加 使用 *Microsoft.AspNetCore.Mvc*)
- 删除任何使用语句引用 *System.Web.Http*
- 更改任何方法返回 *IHttpActionResult* 返回 *IActionResult*

一旦后这些更改已生成且未使用 *using* 语句中删除，已迁移 *ProductsController* 类类似如下所示：

```

using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;
using System.Collections.Generic;
using System.Linq;

namespace ProductsCore.Controllers
{
    [Route("api/[controller]")]
    public class ProductsController : Controller
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        // /api/products
        [HttpGet]
        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        // /api/products/1
        [HttpGet("{id}")]
        public IActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}

```

你现在应能够运行已迁移的项目，浏览到 `/api/产品`；而且，你应看到 3 产品的完整列表。浏览到 `/api/products/1`，你应看到第一个产品。

Microsoft.AspNetCore.Mvc.WebApiCompatShim

迁移 ASP.NET Web API 项目到 ASP.NET 核心时的有用工具

是 [Microsoft.AspNetCore.Mvc.WebApiCompatShim](#) 库。兼容性填充码扩展 ASP.NET 核心以允许不同的 Web API 2 约定，要使用的数目。本文档中进行迁移，以前的示例是基本的兼容性填充程序不是所必需的。对于大型项目，使用兼容性填充码可用于临时之间的隔阂 API ASP.NET Core 和 ASP.NET Web API 2。

Web API 兼容性填充码旨在作为临时的度量值用于促进将大型 Web API 项目迁移到 ASP.NET 核心。随着时间推移，应更新项目以使用 ASP.NET Core 模式，而不是依靠兼容性填充码。

Microsoft.AspNetCore.Mvc.WebApiCompatShim 中包含的兼容性功能包括：

- 将添加 `ApiController` 类型，以便控制器的基类型不需要更新。
- 使 Web API 样式模型绑定。类似于 MVC 5，默认情况下，ASP.NET Core MVC 模型绑定功能。兼容性填充码更改模型要更类似于 Web API 2 模型绑定约定绑定。例如，复杂类型自动绑定从请求正文。
- 将扩展模型绑定，以便控制器操作可以采用参数类型 `HttpRequestMessage`。
- 添加消息格式化程序允许操作来返回结果类型 `HttpResponseMessage`。
- 添加 Web API 2 操作可能具有用于为响应提供服务的其他响应方法：
 - `HttpResponseMessage` 生成器：
 - `CreateResponse<T>`

- `CreateErrorResponse`
- 结果的操作方法:
 - `BadResuestErrorMessageResult`
 - `ExceptionResult`
 - `InternalServerErrorResult`
 - `InvalidModelStateResult`
 - `NegotiatedContentResult`
 - `ResponseMessageResult`

- 将的实例添加 `IContentNegotiator` 到应用程序的 DI 容器和使内容从协商相关类型 `Microsoft.AspNet.WebApi.Client` 可用。这包括类型, 如 `DefaultContentNegotiator`, `MediaTypeFormatter`, 等等。

若要使用的兼容性填充码, 你需要:

- 引用 `Microsoft.AspNetCore.Mvc.WebApiCompatShim` NuGet 包。
- 兼容性填充码的服务注册应用程序的 DI 容器通过调用 `services.AddWebApiConventions()` 中应用程序的 `Startup.ConfigureServices` 方法。
- 定义使用的 Web API 的特定路由 `MapWebApiRoute` 上 `IRouteBuilder` 中应用程序的 `IApplicationBuilder.UseMvc` 调用。

总结

将一个简单的 ASP.NET Web API 项目迁移到 ASP.NET 核心 MVC 将非常简单, 感谢到 ASP.NET 核心 MVC 中的 Web API 的内置支持。每个 ASP.NET Web API 项目将需要迁移的主要部分是路由、控制器和模型, 以及更新到控制器和操作由使用的类型。

将配置迁移到 ASP.NET 核心

2018/5/14 • 2 min to read • [Edit Online](#)

作者: [Steve Smith](#) 和 [Scott Addie](#)

在以前的文章中，我们就已着手[将 ASP.NET MVC 项目迁移到 ASP.NET 核心 MVC](#)。在本文中，我们将迁移配置。

[查看或下载示例代码\(如何下载\)](#)

安装程序配置

ASP.NET 核心不再使用 *Global.asax* 和 *web.config* ASP.NET 的早期版本使用的文件。在 ASP.NET 的早期版本中，应用程序的启动逻辑放入 `Application_Startup` 方法内的 *Global.asax*。更高版本，在 ASP.NET MVC *Startup.cs* 项目中的根目录中包含文件和应用程序启动时调用它。ASP.NET 核心已完全采用这种方法，通过将中的所有启动逻辑 *Startup.cs* 文件。

Web.config 还在 ASP.NET Core 替换文件。配置本身现在可以配置，作为应用程序启动过程中所述的一部分 *Startup.cs*。配置仍然可以利用 XML 文件，但通常 ASP.NET 核心项目将置于配置值的 JSON 格式文件，如 *appsettings.json*。ASP.NET 核心配置系统可以方便地访问环境变量，可以提供[更安全、极其可靠的位置](#)特定于环境的值。这是针对如连接字符串和不应签入源代码管理的 API 密钥的机密尤其如此。请参阅[配置](#) 若要了解有关 ASP.NET 核心中配置的详细信息。

有关本文中，我们开始使用从部分已迁移的 ASP.NET Core 项目[上一篇文章](#)。要设置配置中添加以下构造函数和属性 *Startup.cs* 文件位于项目根目录中：

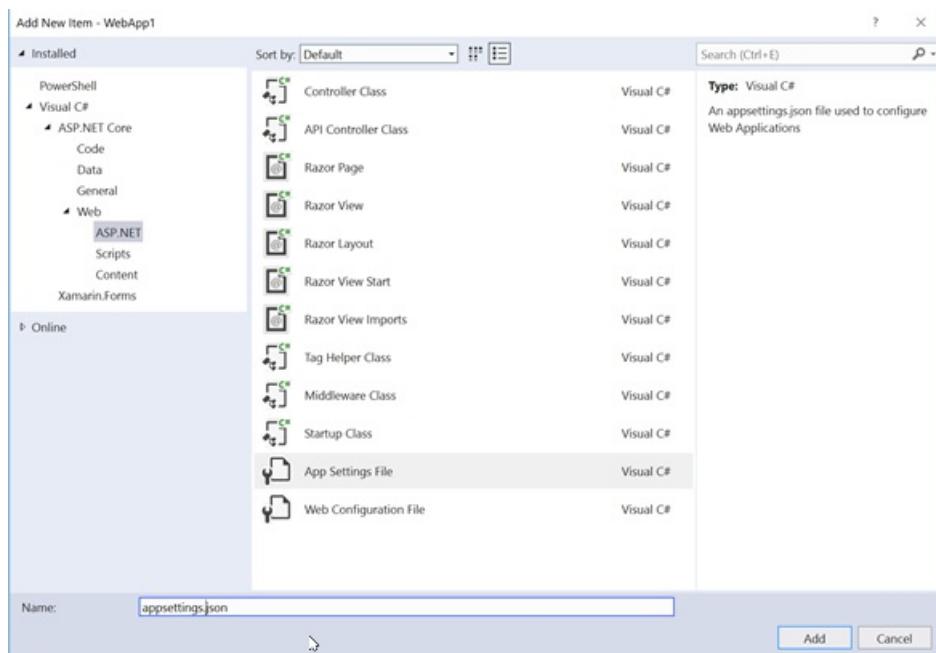
```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

请注意，此时，*Startup.cs* 文件将无法编译，因为我们仍需要将以下内容添加 `using` 语句：

```
using Microsoft.Extensions.Configuration;
```

添加 *appsettings.json* 使用合适的项目模板项目的根目录的文件：



将配置设置迁移从 web.config

我们的 ASP.NET MVC 项目包含中的所需的数据库连接字符串 `web.config` 中 `<connectionStrings>` 元素。在我们 ASP.NET Core 项目中，我们将存储此信息在 `appsettings.json` 文件。打开 `appsettings.json`，并记下它已包含以下：

```
{  
    "Data": {  
        "DefaultConnection": {  
            "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"  
        }  
    }  
}
```

在突出显示的行将上面所示，将更改从数据库的名称 `_CHANGE_ME` 为你的数据库的名称。

总结

ASP.NET 核心将置于单个文件，在其中的必要的服务和依赖项可以定义和配置应用程序的所有启动逻辑。它将替换 `web.config` 与一种灵活的配置功能，可以利用多种文件格式，例如 JSON，以及环境变量的文件。

将身份验证和标识迁移到 ASP.NET 核心

2018/5/14 • 3 min to read • [Edit Online](#)

作者: Steve Smith

在以前的文章中, 我们[配置从 ASP.NET MVC 项目迁移到 ASP.NET 核心 MVC](#)。在本文中, 我们将迁移的注册、登录名和用户管理功能。

配置的标识和成员身份

在 ASP.NET MVC 中, 使用 ASP.NET Identity 中配置身份验证和标识功能 `Startup.Auth.cs` 和 `IdentityConfig.cs` 位于 `App_Start` 文件夹。在 ASP.NET 核心 MVC, 这些功能在中配置 `Startup.cs`。

安装 `Microsoft.AspNetCore.Identity.EntityFrameworkCore` 和 `Microsoft.AspNetCore.Authentication.Cookies` NuGet 包。

然后, 打开 `Startup.cs` 和更新 `Startup.ConfigureServices` 要使用实体框架和标识服务的方法:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity< ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();
}
```

在这种情况下, 有两种在上面的代码中, 我们尚未尚未迁移从 ASP.NET MVC 项目中引用的类型:

`ApplicationDbContext` 和 `ApplicationUser`。创建一个新模型文件夹中 ASP.NET Core 项目, 并将两个类添加到它对应于这些类型。你将找到 ASP.NET MVC 的这些类中的版本 `/Models/IdentityModels.cs`, 但我们将使用每个已迁移的项目中的类的一个文件, 因为它是更清晰。

`ApplicationUser.cs`:

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

namespace NewMvcProject.Models
{
    public class ApplicationUser : IdentityUser
    {
    }
}
```

`ApplicationDbContext.cs`:

```

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Data.Entity;

namespace NewMvcProject.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Identity model and override the defaults if needed.
            // For example, you can rename the ASP.NET Identity table names and more.
            // Add your customizations after calling base.OnModelCreating(builder);
        }
    }
}

```

ASP.NET 核心 MVC 初学者 Web 项目不包括多的自定义的用户，或 `ApplicationDbContext`。当迁移实际应用程序时，你还需要迁移的所有自定义属性和方法的应用程序的用户和 `DbContext` 类，以及你的应用程序使用的任何其他模型类。例如，如果你 `DbContext` 具有 `DbSet<Album>`，你需要迁移 `Album` 类。

与就地情况下，这些文件 `Startup.cs` 文件可以进行编译，方法是更新其 `using` 语句：

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

```

我们的应用程序现在已准备好支持身份验证和标识服务。它只需要具有向用户公开这些功能。

迁移注册和登录名逻辑

使用标识服务对应用程序配置和使用实体框架和 SQL Server 配置的数据访问，我们已准备好将支持注册和登录名添加到应用程序。回想一下，[前面的迁移过程](#) 我们注释掉对的引用 `_LoginPartial` 中 `_Layout.cshtml`。现在，它是返回到该代码中，取消注释它，并在必要的控制器视图和视图，以支持登录功能将添加的时间。

取消注释 `@Html.Partial` 中一行 `_Layout.cshtml`：

```

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
@* @Html.Partial("_LoginPartial") *@
</div>
</div>

```

现在，添加一个新的 Razor 视图调用 `_LoginPartial` 到视图/共享文件夹：

更新 `_LoginPartial.cshtml` 替换为以下代码（替换其内容的所有）：

```
@inject SignInManager< ApplicationUser > SignInManager
@inject UserManager< ApplicationUser > UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="Logout" method="post" id="logoutForm"
class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index" title="Manage">Hello
@UserManager.GetUserName(User) !</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log in</a></li>
    </ul>
}
```

此时，你应能够刷新浏览器中的站点。

总结

ASP.NET 核心引入 ASP.NET 标识功能更改。在本文中，你看到了如何将 ASP.NET 标识的身份验证和用户管理功能迁移到 ASP.NET 核心。

将从 ClaimsPrincipal.Current 迁移

2018/5/14 • 3 min to read • [Edit Online](#)

在 ASP.NET 项目中，常见的做法是使用 `ClaimsPrincipal.Current` 检索当前身份验证的用户的标识和声明。在 ASP.NET 核心，再设置此属性。已根据它的代码需要进行更新，以通过不同方式获取当前经过身份验证的用户的标识。

特定于上下文的数据，而不是静态的数据

使用 ASP.NET 核心，这两者的值时 `ClaimsPrincipal.Current` 和 `Thread.CurrentPrincipal` 未设置。这些属性都表示静态状态，通常避免 ASP.NET Core。相反，ASP.NET 核心体系结构是检索特定于上下文的服务集合（如当前用户的标识）的依赖关系（使用其 [依赖关系注入\(DI\)](#) 模型）。什么是详细，`Thread.CurrentPrincipal` 是线程静态的因此它可能不会持续某些异步方案中的更改（和 `claimsPrincipal.Current` 只调用 `Thread.CurrentPrincipal` 默认情况下）。

若要了解的问题线程类型的静态成员会导致在异步方案中，请考虑下面的代码段：

```
// Create a ClaimsPrincipal and set Thread.CurrentPrincipal
var identity = new ClaimsIdentity();
identity.AddClaim(new Claim(ClaimTypes.Name, "User1"));
Thread.CurrentPrincipal = new ClaimsPrincipal(identity);

// Check the current user
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");

// For the method to complete asynchronously
await Task.Yield();

// Check the current user after
Console.WriteLine($"Current user: {Thread.CurrentPrincipal?.Identity.Name}");
```

前面的示例代码集 `Thread.CurrentPrincipal` 并检查其值之前和之后等待的异步调用。`Thread.CurrentPrincipal` 特定于线程在其上设置，以及该方法有可能继续在另一个线程后的执行。因此，`Thread.CurrentPrincipal` 时的存在时将首先检查，但为 null 的调用后 `await Task.Yield()`。

从应用程序的 DI 服务集合获取当前用户的标识太多可测试性，因为可轻松插入测试标识。

检索当前用户在 ASP.NET Core 应用程序

有几个选项用于检索当前经过身份验证的用户的 `ClaimsPrincipal` 中代替了 ASP.NET Core `ClaimsPrincipal.Current`：

- **ControllerBase.User**。MVC 控制器可以访问当前的身份验证的用户使用其 [用户](#) 属性。
- **HttpContext.User**。有权访问当前组件 `HttpContext`（中间件）可以获取当前用户的 `ClaimsPrincipal` 从 [HttpContext.User](#)。
- **从调用方传入**。无需访问当前库 `HttpContext` 通常称为从控制器或中间件组件，并且可以具有作为参数传递的当前用户的标识。
- **IHttpContextAccessor**。ASP.NET 项目迁移到 ASP.NET 核心可能太大，可轻松地将当前用户的标识传递到所有必要的位置。在这种情况下，[IHttpContextAccessor](#) 可以用作一种解决方法。`IHttpContextAccessor` 能够访问当前 `HttpContext`（如果存在）。将获取尚未尚未更新可以使用 ASP.NET Core DI 驱动体系结构的代码中的当前用户的标识的短期解决方案：

- 请 `IHttpContextAccessor` DI 容器通过调用中可用 `AddHttpContextAccessor` 中 `Startup.ConfigureServices`。
- 获取其实例 `IHttpContextAccessor` 在启动过程并将其存储在静态变量。实例是可用于以前从静态属性检索当前用户的代码。
- 检索当前用户的 `ClaimsPrincipal` 使用 `HttpContextAccessor.HttpContext?.User`。如果此代码使用的 HTTP 请求的上下文之外 `HttpContext` 为 null。

最后一个选项，请使用 `IHttpContextAccessor`，行为是违背 ASP.NET 核心原则（首选静态依赖项的插入依赖关系）。计划最终删除依赖于静态 `IHttpContextAccessor` 帮助器。不过，可以迁移以前使用的大型现有的 ASP.NET 应用时它是一个有用的桥，`ClaimsPrincipal.Current`。

ASP.NET 成员资格身份验证从迁移到 ASP.NET 核心 2.0 标识

2018/5/14 • 7 min to read • [Edit Online](#)

作者 : Isaac Levin

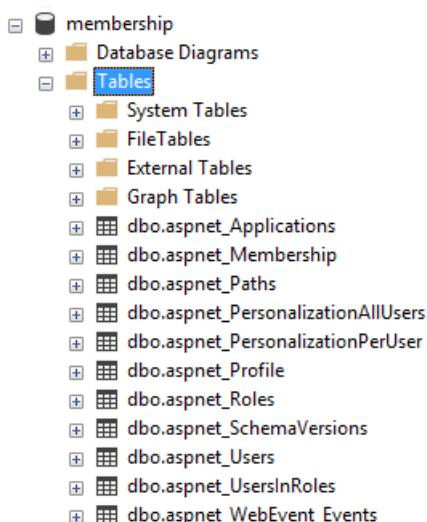
本文演示如何迁移使用成员资格到 ASP.NET 核心 2.0 标识的身份验证的 ASP.NET 应用程序的数据架构。

注意

本文档提供将基于 ASP.NET 成员资格的应用程序的数据架构迁移到用于 ASP.NET 核心标识的数据架构所需的步骤。有关从基于 ASP.NET 成员身份的身份验证迁移到 ASP.NET Identity 的详细信息, 请参阅[将从 SQL 成员资格的现有应用程序迁移到 ASP.NET 标识](#)。有关 ASP.NET 核心标识的详细信息, 请参阅[ASP.NET Core 上的标识简介](#)。

成员资格架构评审

在 ASP.NET 2.0 中之前, 开发人员已执行的任务创建为其应用程序的整个身份验证和授权过程。ASP.NET 2.0 中, 成员身份已引入, 提供处理的 ASP.NET 应用程序内的安全的样本解决方案。开发人员现在已能够启动到使用 SQL Server 数据库的架构[aspnet_regsql.exe](#)命令。运行此命令后下, 表在数据库中创建。



若要将现有应用迁移到 ASP.NET 核心 2.0 标识, 这些表中的数据需要迁移到使用新的标识架构的表。

ASP.NET 核心标识 2.0 架构

ASP.NET 核心 2.0 遵循[标识](#)ASP.NET 4.5 中引入的原则。尽管共享的原则, 框架之间的实现是不同的甚至 ASP.NET 核心的版本之间 (请参阅[将身份验证和标识迁移到 ASP.NET 核心 2.0](#))。

ASP.NET 核心 2.0 标识查看架构的最快方法是创建新的 ASP.NET 核心 2.0 应用程序。请按照下列步骤在 Visual Studio 2017:

- 选择“文件” > “新建” > “项目”。
- 创建一个新**ASP.NET 核心 Web 应用程序**, 并将项目*CoreIdentitySample*。
- 在下拉列表中选择“ASP.NET Core 2.0”, 然后选择“Web 应用程序”。此模板生成**Razor** 页应用。然后单击**确定**, 单击**更改身份验证**。
- 选择单个用户帐户标识模板。最后, 单击**确定**, 然后**确定**。Visual Studio 创建项目时使用 ASP.NET 核心标识模

板。

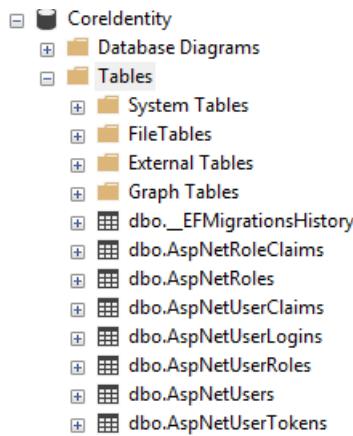
ASP.NET 核心 2.0 标识使用实体框架核心与存储的身份验证数据的数据库进行交互。为了使新创建应用程序处理，需要使用数据库来存储此数据。在创建新的应用程序后，检查数据库环境中的架构的最快方法是创建使用实体框架迁移的数据库。此过程创建数据库，或者本地或其他位置，其中模仿该架构。查看前面的文档以了解更多信息。

若要使用 ASP.NET 核心标识架构中创建数据库，运行 `Update-Database` 命令在 Visual Studio 的程序包管理器控制台(PMC) 窗口一位于工具 > NuGet 包管理器 > 程序包管理器控制台。PMC 支持实体框架中运行命令。

实体框架命令使用连接字符串中指定的数据库为 `appsettings.json`。以下连接字符串上的目标数据库 `localhost` 名为 `asp net 核心标识`。在此设置中，实体框架配置为使用 `DefaultConnection` 连接字符串。

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=localhost;Database=aspnet-core-  
identity;Trusted_Connection=True;MultipleActiveResultSets=true"  
  }  
}
```

此命令将生成指定的架构的数据库和所需的应用程序初始化任何数据。下图描绘了使用前面的步骤创建的表结构。



迁移架构

有细微的差别的表结构和成员身份和 ASP.NET 核心标识字段。模式已显著更改身份验证/授权与 ASP.NET 和 ASP.NET Core 应用程序。仍用于标识的键对象具有 `用户` 和 `角色`。以下是有关映射表 `用户`、`角色` 和 `UserRoles`。

用户

<code>IDENTITY(ASPNETUSERS)</code>		<code>MEMBERSHIP(ASPNET_USERS/A SPNET_MEMBERSHIP)</code>	
字段名称	Type	字段名称	Type
<code>Id</code>	字符串	<code>aspnet_Users.UserId</code>	字符串
<code>UserName</code>	字符串	<code>aspnet_Users.UserName</code>	字符串
<code>Email</code>	字符串	<code>aspnet_Membership.Email</code>	字符串
<code>NormalizedUserName</code>	字符串	<code>aspnet_Users.NormalizedUserName</code>	字符串
<code>NormalizedEmail</code>	字符串	<code>aspnet_Membership.NormalizedEmail</code>	字符串

IDENTITY(ASPNETUSERS)		MEMBERSHIP(ASPNET_USERS/A SPNET_MEMBERSHIP)	
<code>PhoneNumber</code>	字符串	<code>aspnet_Users.MobileAlias</code>	字符串
<code>LockoutEnabled</code>	位	<code>aspnet_Membership.IsLockedOut</code>	位

注意

并非所有字段映射类似都于从成员资格到 ASP.NET 核心标识的一对一关系。前面的表使用默认成员资格用户架构，并将其映射到 ASP.NET 核心标识架构。已用于成员资格的任何其他自定义字段都需要手动映射。在此映射中，密码，没有映射因为密码条件和密码 salt 会迁移两个之间。建议可保留为 `null` 的密码并要求用户重置其密码。在 ASP.NET 核心标识 `LockoutEnd` 应设置为在将来的某个日期中，如果用户被锁定。迁移脚本所示。

角色

IDENTITY(ASPNETROLES)		MEMBERSHIP(ASPNET_ROLES)	
字段名称	Type	字段名称	Type
<code>Id</code>	字符串	<code>RoleId</code>	字符串
<code>Name</code>	字符串	<code>RoleName</code>	字符串
<code>Normalized Name</code>	字符串	<code>LoweredRoleName</code>	字符串

用户角色

IDENTITY(ASPNETUSERROLES)		MEMBERSHIP(ASPNET_USERSI NROLES)	
字段名称	Type	字段名称	Type
<code>RoleId</code>	字符串	<code>RoleId</code>	字符串
<code>UserId</code>	字符串	<code>UserId</code>	字符串

创建用于的迁移脚本时引用前面的映射表`用户`和`角色`。下面的示例假定数据库服务器上有两个数据库。一个数据库包含的现有的 ASP.NET 成员资格架构和数据。使用前面所述的步骤创建另一个数据库。注释是以内联形式包含有关详细信息。

```
-- THIS SCRIPT NEEDS TO RUN FROM THE CONTEXT OF THE MEMBERSHIP DB
BEGIN TRANSACTION MigrateUsersAndRoles
use aspnetdb

-- INSERT USERS
INSERT INTO coreidentity.dbo.aspnetusers
    (id,
     username,
     normalizedusername,
     passwordhash,
     securitystamp,
     emailconfirmed,
     phonenumbers,
     phonenumbersconfirmed,
     twofactorenabled,
```

```

        lockoutend,
        lockoutenabled,
        accessfailedcount,
        email,
        normalizedemail)
SELECT aspnet_users.userid,
       aspnet_users.username,
       aspnet_users.loweredusername,
       --Creates an empty password since passwords don't map between the two schemas
       '',
       --Security Stamp is a token used to verify the state of an account and is subject to change at any
       time. It should be initialized as a new ID.
       NewID(),
       --EmailConfirmed is set when a new user is created and confirmed via email. Users must have this set
       during migration to ensure they're able to reset passwords.
       1,
       aspnet_users.mobilealias,
       CASE
           WHEN aspnet_Users.MobileAlias is null THEN 0
           ELSE 1
       END,
       --2-factor Auth likely wasn't setup in Membership for users, so setting as false.
       0,
       CASE
           --Setting lockout date to time in the future (1000 years)
           WHEN aspnet_membership.islockedout = 1 THEN Dateadd(year, 1000,
                                                               Sysutcdatetime())
           ELSE NULL
       END,
       aspnet_membership.islockedout,
       --AccessFailedAccount is used to track failed logins. This is stored in membership in multiple columns.
       Setting to 0 arbitrarily.
       0,
       aspnet_membership.email,
       aspnet_membership.loweredemail
FROM aspnet_users
LEFT OUTER JOIN aspnet_membership
    ON aspnet_membership.applicationid =
       aspnet_users.applicationid
       AND aspnet_users.userid = aspnet_membership.userid
LEFT OUTER JOIN coreidentity.dbo.aspnetusers
    ON aspnet_membership.userid = aspnetusers.id
WHERE aspnetusers.id IS NULL

-- INSERT ROLES
INSERT INTO coreIdentity.dbo.aspnetroles(id,name)
SELECT roleId,rolename
FROM aspnet_roles;

-- INSERT USER ROLES
INSERT INTO coreidentity.dbo.aspnetuserroles(userid,roleid)
SELECT userid,roleid
FROM aspnet_usersinroles;

IF @@ERROR <> 0
BEGIN
    ROLLBACK TRANSACTION MigrateUsersAndRoles
    RETURN
END

COMMIT TRANSACTION MigrateUsersAndRoles

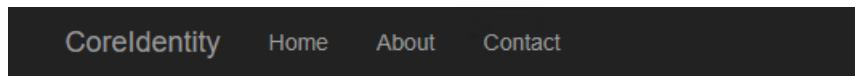
```

完成后此脚本，使用成员资格用户填充前面创建的 ASP.NET 核心标识应用程序。用户需要在中的日志记录之前更改其密码。

注意

如果成员资格系统具有与不匹配其电子邮件地址的用户名的用户，不需要向应用程序创建早期以适应此更改。默认模板需要 `UserName` 和 `Email` 相同。对于它们是不同的情况下，登录过程需要进行修改以使用 `UserName` 而不是 `Email`。

在 `PageModel` 的登录页上，位于 `Pages\Account\Login.cshtml.cs`，删除 `[EmailAddress]` 属性从 `电子邮件` 属性。将其命名为 `用户名`。这需要更改任何地方 `EmailAddress` 提到，在 `视图` 和 `PageModel`。结果如下所示：



Log in

Use a local account to log in.

UserName

Password

, Remember me?

[Forgot your password?](#)

[Register as a new user](#)

后续步骤

在本教程中，您学习了如何移植到 ASP.NET 核心 2.0 标识从 SQL 成员资格用户。有关 ASP.NET 核心标识的详细信息，请参阅[标识简介](#)。

将 HTTP 处理程序和模块迁移到 ASP.NET 核心中间件

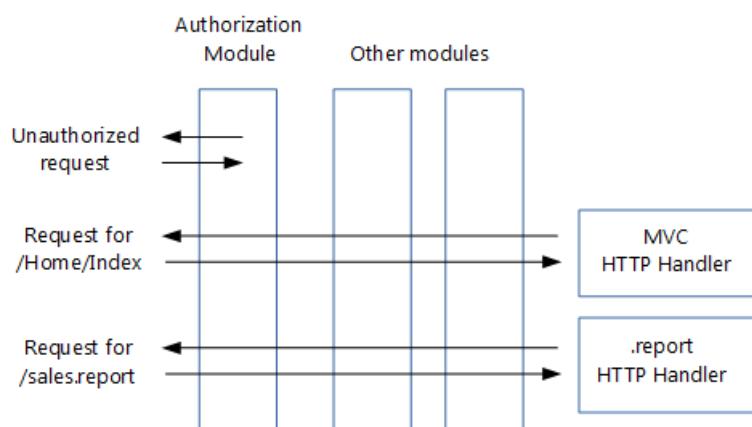
2018/5/4 • 17 min to read • [Edit Online](#)

通过 [Matt Perdeck](#)

这篇文章演示如何迁移现有的 ASP.NET HTTP 模块和处理程序 `system.webserver` 到 ASP.NET 核心中间件。

模块和处理程序重新访问

在继续之前到 ASP.NET 核心中间件，让我们首先会扼要重述 HTTP 模块和处理程序的工作原理：



处理程序：

- 类实现 [IHttpHandler](#)
- 用于使用请求的给定的文件名或扩展，如 `.report`
- [配置](#) 中 `Web.config`

模块为：

- 类实现 [IHttpModule](#)
- 调用为每个请求
- 能够短路（停止进一步处理请求）
- 无法添加到 HTTP 响应中，或创建自己
- [配置](#) 中 `Web.config`

模块顺序处理传入的请求的顺序取决于：

- [应用程序生命周期](#)，这是由 ASP.NET 激发的系列事件：`BeginRequest`, `AuthenticateRequest` 等。每个模块可以创建一个或多个事件处理程序。
- 对于相同的事情，它们在配置中的顺序 `Web.config`。

除模块，还可以添加到生命周期事件的处理程序你 `Global.asax.cs` 文件。在已配置的模块中的处理程序后运行这些处理程序。

从处理程序和到中间件模块

中间件是 HTTP 模块和处理程序比简单得多：

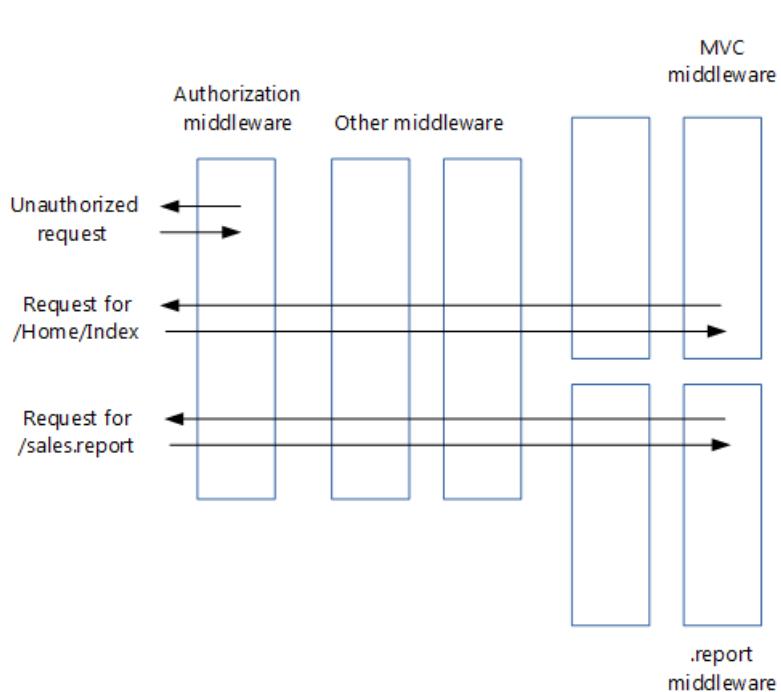
- 模块、处理程序, *Global.asax.cs*, *Web.config* (除外 IIS 配置) 和应用程序生命周期已消失
- 模块和处理程序的角色具有接管中间件
- 中间件配置使用代码而不是在*Web.config*
- [管道分支](#)允许将请求发送到特定的中间件, 基于不仅也上请求标头、查询字符串等的 URL。

中间件是非常类似于模块：

- 在每个请求主体中调用
- 能够通过短路请求[不将请求传递到下一步的中间件](#)
- 能够创建他们自己的 HTTP 响应

中间件和模块按不同顺序处理：

- 中间件的顺序基于在其中插入到请求管道中, 而模块的顺序主要基于的顺序[应用程序生命周期事件](#)
- 响应的中间件顺序是对于请求, 与反向而模块的顺序是相同的请求和响应
- 请参阅[使用 *IApplicationBuilder* 创建中间件管道](#)



请注意如何在上图中, 身份验证中间件 short-circuited 请求。

迁移到中间件模块代码

现有 HTTP 模块看起来类似于此：

```
// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new EventHandler(this.Application_EndRequest));
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}
```

中所示中间件页上，ASP.NET Core 中间件是公开的类 `Invoke` 方法拍摄 `HttpContext` 并返回 `Task`。新中间件将如下所示：

```

// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }

    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}

```

前面的中间件模板摘录自部分[编写中间件](#)。

`MyMiddlewareExtensions`帮助器类，更便于配置中的中间件你 `Startup` 类。`UseMyMiddleware` 方法将您中间件的类添加到请求管道。所需的中间件服务获取注入到中间件的构造函数。

你的模块可能终止请求，例如，如果用户未授权：

```

// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}

```

中间件的方式处理这通过不调用 `Invoke` 在管道中的下一步中间件。请注意这不完全终止请求，因为响应使成为管道上返回到其方法时，仍可以调用以前的中间件。

```
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.

    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

当你迁移到新中间件模块的功能时，你可能会发现你的代码不会编译，因为 `HttpContext` 类中 ASP.NET Core 已显著更改。[更高版本上](#)，你将了解如何将迁移到新的 ASP.NET 核心 `HttpContext`。

迁移模块插入请求管道

HTTP 模块通常会添加到请求管道使用 `Web.config`:

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
    <system.webServer>
        <modules>
            <add name="MyModule" type="MyApp.Modules.MyModule"/>
        </modules>
    </system.webServer>
</configuration>
```

转换这一点[添加新中间件](#)向请求管道中你 `Startup` 类:

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>()
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });
}

app.MapWhen(
    context => context.Request.Path.ToString().EndsWith(".context"),
    appBranch => {
        appBranch.UseHttpContextDemoMiddleware();
    });

app.UseStaticFiles();

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

```

你可以将插入新中间件管道中的确切点取决于它作为模块处理的事件 (`BeginRequest`, `EndRequest` 等) 和在列表中的模块中的顺序 `Web.config`。

如前面所述, 没有任何应用程序生命周期中 ASP.NET 核心, 中间件处理响应的顺序不同于使用模块的顺序。这会使排序决策更具挑战性。

如果排序将为问题, 无法将你的模块分成多个可在单独对其进行排序的中间件组件。

迁移到中间件的处理程序代码

HTTP 处理程序如下所示:

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

在 ASP.NET Core 项目中，你将翻译以下到中间件类似于以下内容：

```

// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {

        // Must have constructor with this signature, otherwise exception at run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}

```

此中间件是非常类似于对应于模块的中间件。唯一的真正的区别是，此处没有不需要调用 `_next.Invoke(context)`。有意义，因为该处理程序末尾的请求管道，因此将没有下一步的中间件来调用。

迁移的处理程序插入到请求管道

配置的 HTTP 处理程序中完成 *Web.config* 和如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report" type="MyApp.HttpHandlers.MyHandler"
resourceType="Unspecified" preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

你无法将其转换通过将新的处理程序中间件添加到请求管道中你 `Startup` 类，类似于从模块转换的中间件。这种方法的问题是，它会将所有请求都发送到新的处理程序中间件。但是，你只想具有给定扩展名的请求来访问中间件。这样，你必须与 HTTP 处理程序的相同功能。

一种解决方案是分支的扩展名为给定的请求管道使用 `MapWhen` 扩展方法。执行此操作在同一 `Configure` 你在其中添加其他中间件的方法：

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>()
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });
}

app.MapWhen(
    context => context.Request.Path.ToString().EndsWith(".context"),
    appBranch => {
        appBranch.UseHttpContextDemoMiddleware();
    });

app.UseStaticFiles();

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

```

`MapWhen` 使用这些参数：

- 采用的 lambda `HttpContext` 并返回 `true` 如果请求应会减少分支。这意味着可以分支请求而不仅仅是基于其扩展，而且取决请求标头、查询字符串参数，等等。
- 采用的 lambda `IApplicationBuilder` 并添加分支的所有中间件。这意味着处理程序中间件的前面添加到分支其他中间件。

中间件将添加到管道，然后将所有请求；调用分支分支将在其上没有任何影响。

加载使用选项模式的中间件选项

一些模块和处理程序已在存储的配置选项 *Web.config*。但是，在 ASP.NET 核心中新的配置模型使用代替了 *Web.config*。

[新配置系统](#)为你提供了这些选项，以解决此问题：

- 直接注入到中间件，选项中所示[下一节](#)。
- 使用[选项模式](#)：

1. 创建一个类来保存中间件的选项，例如：

```
public class MyMiddlewareOptions
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}
```

2. 存储选项的值

配置系统，可存储选项任意位置所需的值。但是，最站点使用 *appsettings.json*，因此我们将采用这种办法：

```
{
    "MyMiddlewareOptionsSection": {
        "Param1": "Param1Value",
        "Param2": "Param2Value"
    }
}
```

*MyMiddlewareOptionsSection*下面是部分名称。它不必是你的选项类别的名称相同。

3. 将选项值与选项类相关联

选项模式使用 ASP.NET 核心依赖关系注入框架将选项类型相关联（如 `MyMiddlewareOptions`）与 `MyMiddlewareOptions` 具有实际选项对象。

更新你 `Startup` 类：

a. 如果你使用 *appsettings.json*，将其添加到中的配置生成器 `Startup` 构造函数：

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

b. 配置选项服务：

```

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}

```

c. 将你的选项与选项类相关联：

```

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}

```

4. 插入到中间件构造函数的选项。这是类似于将注入到控制器的选项。

```

public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}

```

[UseMiddleware](#)将添加到中间件的扩展方法 `IApplicationBuilder` 负责的依赖关系注入。

这已不再局限于 `IOptions` 对象。这种方式，可插入中间件需要的任何其他对象。

加载通过直接注入的中间件选项

选项模式具有创建松散耦合选项值与其使用者之间的优点。一旦你已为实际的选项值其关联选项类别，任何其他

类可获得访问通过依赖关系注入框架的选项。没有无需传递选项值。

这将分解但如果你想要使用相同的中间件两次，使用不同的选项。例如授权中间允许不同的角色的不同分支中使用。不能将两个不同的选项对象与一个选项类相关联。

解决方法是获取使用中的实际选项值的选项对象你 `Startup` 类并将这些直接向中间件的每个实例参数传递。

1. 添加到的第二个键 `appsettings.json`

若要添加另一组选项 `appsettings.json` 文件中，使用新密钥来唯一标识它：

```
{  
  "MyMiddlewareOptionsSection2": {  
    "Param1": "Param1Value2",  
    "Param2": "Param2Value2"  
  },  
  "MyMiddlewareOptionsSection": {  
    "Param1": "Param1Value",  
    "Param2": "Param2Value"  
  }  
}
```

2. 检索选项值，并将它们传递给中间件。`Use...` 扩展方法（其中添加到管道的中间件）是传入选项值的逻辑位置：

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });
}

app.MapWhen(
    context => context.Request.Path.ToString().EndsWith(".context"),
    appBranch => {
        appBranch.UseHttpContextDemoMiddleware();
    });

app.UseStaticFiles();

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

```

3. 启用的中间件来采用选项参数。提供的重载 `Use...` 扩展方法 (它接收 `options` 参数并将其传递给 `UseMiddleware`)。当 `UseMiddleware` 调用具有参数，它将参数传递给中间件构造函数时它实例化的中间件对象。

```
public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>(myMiddlewareOptions));
    }
}
```

请注意这如何包装中的选项对象 `OptionsWrapper` 对象。这将实现 `IOptions`，如下所需的中间件构造函数。

迁移到新 HttpContext

你此前看到的 `Invoke` 中间件中的方法采用一个参数的类型 `HttpContext`：

```
public async Task Invoke(HttpContext context)
```

`HttpContext` 已显著更改 ASP.NET Core 中。本部分说明如何将转换的最常用的属性 `System.Web.HttpContext` 对应项 `Microsoft.AspNetCore.Http.HttpContext`。

HttpContext

`HttpContext.Items` 都会转换为：

```
IDictionary<object, object> items = httpContext.Items;
```

唯一请求 ID (没有 `System.Web.HttpContext` 对应项)

为你的唯一 id 用于每个请求。日志中包括非常有用。

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

`HttpContext.Request.HttpMethod` 都会转换为：

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString

`HttpContext.Request.QueryString` 都会转换为：

```

IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item ("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items ("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();

```

HttpContext.Request.Url和**HttpContext.Request.RawUrl**将转换为:

```

// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();

```

HttpContext.Request.IsSecureConnection都会转换为:

```

var isSecureConnection = httpContext.Request.IsHttps;

```

HttpContext.Request.UserHostAddress都会转换为:

```

var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();

```

HttpContext.Request.Cookies都会转换为:

```

IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no exception)
string knownCookieValue = cookies["cookie1name"]; // will be actual value

```

HttpContext.Request.RequestContext.RouteData都会转换为:

```

var routeValue = httpContext.GetRouteValue("key");

```

HttpContext.Request.Headers都会转换为:

```

// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue = requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and unknownHeaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownHeaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownHeaderValue is the value
IList<string> knownheaderValues = headersDictionary[HeaderNames.AcceptLanguage];
string knownHeaderValue = headersDictionary[HeaderNames.AcceptLanguage].ToString();

```

HttpContext.Request.UserAgent都会转换为:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer都会转换为：

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType都会转换为：

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType.ToString(); // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type.ToString(); // ex. application
string contentSubType = mediaHeaderValue?.SubType.ToString(); // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form都会转换为：

```
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = httpContext.Request.Form; // sync
    // Or
    form = await httpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

警告

仅当内容的子类型为读取窗体值x-响应客户-窗体-urlencoded或窗体数据。

HttpContext.Request.InputStream都会转换为：

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

警告

仅在处理程序类型中间件，末尾的管道中使用此代码。

每个请求的唯一一次如上所示，你可以阅读原始的正文。尝试后第一次读取读取正文的中间件将读取正文为空。

这不适用于读取窗体，如下所示更早版本，因为缓冲区中完成。

HttpContext.Response

HttpContext.Response.Status和**HttpContext.Response.StatusDescription**将转换为：

```
// using Microsoft.AspNetCore.Http;
HttpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding和**HttpContext.Response.ContentType**将转换为：

```
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
HttpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType上其自身还转换为：

```
HttpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output都会转换为：

```
string responseContent = GetResponseContent();
await HttpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

为文件提供服务讨论[此处](#)。

HttpContext.Response.Headers

发送响应标头非常复杂，这一事实，如果任何内容都已写入响应正文将它们设置，它们将不发送。

解决方案是将设置将右之前调用写入响应启动的回调方法。最好的做法是在开始 `Invoke` 中间件中的方法。这是此回调方法，设置响应标头。

下面的代码设置调用的回调方法 `SetHeaders`：

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

`SetHeaders` 回调方法将如下所示：

```

// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.AspNetCore.Http;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1", "headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] = responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] = "http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}

```

HttpContext.Response.Cookies

Cookie 出差到中的浏览器Set-cookie响应标头。因此，发送 cookie 都需要用于发送响应标头为使用相同的回调：

```

public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
}

```

`SetCookies` 回调方法将如下所示：

```

private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5), HttpOnly = true });

    // If you use .Net 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}

```

其他资源

- [HTTP 处理程序和 HTTP 模块概述](#)

- 配置
- 应用程序启动
- 中间件

从 ASP.NET Core 1.x 迁移到 2.0

2018/5/4 • 9 min to read • [Edit Online](#)

作者: Scott Addie

本文将演示如何将现有 ASP.NET Core 1.x 项目更新到 ASP.NET Core 2.0。通过将应用程序迁移到 ASP.NET Core 2.0, 可利用[大量新功能和改进功能](#)。

现有 ASP.NET Core 1.x 应用程序基于版本特定的项目模板。随着 ASP.NET Core 框架不断演变, 其中的项目模板和起始代码也在变化。除了更新 ASP.NET Core 框架外, 还需要为应用程序更新代码。

系统必备

请参阅 [ASP.NET Core 入门](#)。

更新目标框架名字对象 (TFM)

面向 .NET Core 的项目需使用大于或等于 .NET Core 2.0 版本的 TFM。在“.csproj”文件中搜索 `<TargetFramework>` 节点, 并将其内部文本替换为 `netcoreapp2.0` :

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

面向 .NET Framework 的项目需使用大于或等于 .NET Framework 4.6.1 版本的 TFM。在“.csproj”文件中搜索 `<TargetFramework>` 节点, 并将其内部文本替换为 `net461` :

```
<TargetFramework>net461</TargetFramework>
```

注意

相比于 .NET Core 1.x, .NET Core 2.0 提供更多的外围应用。如果仅因为 .NET Core 1.x 中缺少 API 而要面向 .NET Framework, 则定向于 .NET Core 2.0 可能有用。

在 global.json 中更新 .NET Core SDK 版本

如果解决方案依靠 `global.json` 文件来定向于特定 .NET Core SDK 版本, 请更新其 `version` 属性以使用计算机上安装的 2.0 版本:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

更新包引用

1.x 项目中的“.csproj”文件列出了该项目使用的每个 NuGet 包。

在面向 .NET Core 2.0 的 ASP.NET Core 2.0 项目中, “.csproj”文件中的单个 `metapackage` 引用将替换包的集合:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
</ItemGroup>
```

该元包中具备 ASP.NET Core 2.0 和 Entity Framework Core 2.0 的所有功能。

面向 .NET Framework 的 ASP.NET Core 2.0 项目应继续引用单个 NuGet 包。将每个 `<PackageReference />` 节点的 `Version` 特性更新至 2.0.0。

例如，下述列表列出了面向 .NET Framework 的典型 ASP.NET Core 2.0 项目中使用的 `<PackageReference />` 节点：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0"
    PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.0" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
    PrivateAssets="All" />
</ItemGroup>
```

更新 .NET Core CLI 工具

在“.csproj”文件中，将每个 `<DotNetCliToolReference />` 节点的 `Version` 特性更新至 2.0.0。

例如，下述列表列出了面向 .NET Core 2.0 的典型 ASP.NET Core 2.0 项目中使用的 CLI 工具：

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

重命名“包目标回退”属性

1.x 项目的“.csproj”文件使用了 `PackageTargetFallback` 节点和变量：

```
<PackageTargetFallback>$(<PackageTargetFallback>);portable-net45+win8+wp8+wpa81;</PackageTargetFallback>
```

将节点和变量重命名为 `AssetTargetFallback`：

```
<AssetTargetFallback>$(<AssetTargetFallback>);portable-net45+win8+wp8+wpa81;</AssetTargetFallback>
```

更新 Program.cs 中的 Main 方法

在 1.x 项目中，“Program.cs”的 `Main` 方法如下所示：

```

using System.IO;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

            host.Run();
        }
    }
}

```

在 2.0 项目中，简化了“Program.cs”的 `Main` 方法：

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

强烈建议采用新的 2.0 模式，[Entity Framework \(EF\) Core 迁移](#)等产品功能需要此模式才能正常运行。例如，从“包管理器控制台”窗口运行 `Update-Database`，或从命令行（位于转换为 ASP.NET Core 2.0 的项目上）运行 `dotnet ef database update` 时，都会生成以下错误：

```

Unable to create an object of type '<Context>'. Add an implementation of
'IDesignTimeDbContextFactory<Context>' to the project, or see https://go.microsoft.com/fwlink/?linkid=851728
for additional patterns supported at design time.

```

添加配置提供程序

在 1.x 项目中，已通过 `Startup` 构造函数将配置提供程序添加到了某个应用。涉及的步骤包括创建 `ConfigurationBuilder` 实例、加载适用的提供程序（环境变量、应用设置等）以及初始化 `IConfigurationRoot` 的成员。

```

public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }

```

上例使用 `psettings.json` 以及任何与 `IHostingEnvironment.EnvironmentName` 属性匹配的 `appsettings.<EnvironmentName>.json` 文件中的配置设置加载 `Configuration` 成员。这些文件所在位置与 `Startup.cs` 的路径相同。

在 2.0 项目中，样板配置代码会继承在幕后运行的 1.x 代码。例如，启动时就加载环境变量和应用设置。等效的 `Startup.cs` 代码减少到 `IConfiguration` 初始化设置并包括插入的实例：

```

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

public IConfiguration Configuration { get; }

```

若要删除由 `WebHostBuilder.CreateDefaultBuilder` 添加的默认提供程序，请对 `ConfigureAppConfiguration` 内的 `IConfigurationBuilder.Sources` 属性调用 `Clear` 方法。若要添加回提供程序，请使用 `Program.cs` 中的 `ConfigureAppConfiguration` 方法：

```

public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
    {
        // delete all default configuration providers
        config.Sources.Clear();
        config.AddJsonFile("myconfig.json", optional: true);
    })
    .Build();

```

若要查看上一代码片段中 `CreateDefaultBuilder` 方法使用的配置，请参阅[此处](#)。

有关详细信息，请参阅 [ASP.NET Core 中的配置](#)。

移动数据库初始化代码

在 1.x 项目中使用 EF Core 1.x(类似 `dotnet ef migrations add` 的命令)执行下列任务：

1. 实例化 `Startup` 实例
2. 调用 `ConfigureServices` 方法, 为所有服务注册依赖关系注入(包括 `DbContext` 类型)
3. 执行其必要任务

在 2.0 项目中使用 EF Core 2.0, 调用 `Program.BuildWebHost` 以获取应用程序服务。与 1.x 不同, 这有调用 `Startup.Configure` 的副作用。如果 1.x 应用在其 `Configure` 方法中调用了数据库初始化代码, 可能出现意外问题。例如, 如果数据库尚不存在, 种子设定代码将在 EF Core 迁移命令执行前运行。如果数据库尚不存在, 此问题将导致 `dotnet ef migrations list` 命令失败。

请考虑在 `Startup.cs` 的 `Configure` 方法中使用以下 1.x 种子初始化代码。

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

SeedData.Initialize(app.ApplicationServices);
```

在 2.0 项目中, 将 `SeedData.Initialize` 调用移动到 `Program.cs` 的 `Main` 方法:

```
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

从 2.0 开始, `BuildWebHost` 只应用于生成和配置 Web 主机。有关运行应用程序的任何内容都应在 `BuildWebHost` 一外部处理, 通常是在 `Program.cs` 的 `Main` 方法中。

查看 Razor 视图编译设置

加快应用程序启动速度和缩小已发布的捆绑包至关重要。为此, ASP.NET Core 2.0 中默认启用 [Razor 视图编译](#)。

无需再将 `MvcRazorCompileOnPublish` 属性设置为 `true`。若不禁用视图编译, 可能会从“`.csproj`”文件中删除此属性。

以 .NET Framework 为目标时, 仍需显式引用“`.csproj`”文件中的 [Microsoft.AspNetCore.Mvc.Razor.ViewCompilation](#) NuGet 包:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0" PrivateAssets="All"
/>
```

依靠 Application Insights“启动”功能

能够轻松设置应用程序性能检测非常重要。现可依靠 Visual Studio 2017 工具中推出的新的 [Application Insights](#)“启动”功能。

Visual Studio 2017 中创建的 ASP.NET Core 1.1 项目默认添加 Application Insights。若不直接使用 Application Insights SDK，则除了执行“Program.cs”和“Startup.cs”，还请执行以下步骤：

1. 如果定目标到 .NET Core，请从 .csproj 文件中删除以下 `<PackageReference />` 节点：

```
<PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
```

2. 如果定目标到 .NET Core，请从 Program.cs 中删除 `UseApplicationInsights` 扩展方法调用：

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .UseApplicationInsights()
        .Build();

    host.Run();
}
```

3. 从“_Layout.cshtml”中删除 Application Insights 客户端 API 调用。它会比较以下两行代码：

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
@Html.Raw(JavaScriptSnippet.FullScript)
```

若要直接使用 Application Insights SDK，请继续此操作。2.0 [元包](#) 中具备最新版本的 Application Insights，因此如果引用较旧版本，将出现包降级错误。

采用身份验证/标识改进

ASP.NET Core 2.0 具有新的身份验证模型和大量针对 ASP.NET Core 标识的重大更改。如果在启用个人用户帐户的情况下创建了项目，或者已手动添加身份验证或标识，请参阅[将身份验证和标识迁移到 ASP.NET Core 2.0](#)。

其他资源

- [ASP.NET Core 2.0 中的重大更改](#)

将身份验证和标识迁移到 ASP.NET 核心 2.0

2018/5/4 • 9 min to read • [Edit Online](#)

通过[Scott Addie](#)和[Hao 永远](#)

ASP.NET 核心 2.0 具有用于身份验证的新模型和标识。这简化了使用服务的配置。ASP.NET 核心 1.x 应用程序使用身份验证或标识可以更新以使用新的模型，如下所述。

身份验证中间件和服务

在 1.x 项目中，通过中间件配置身份验证。中间件方法调用每个你想要支持的身份验证方案。

下面的 1.x 示例将 Facebook 身份验证配置中标识`Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
    });
}
```

在 2.0 项目中，通过服务配置身份验证。在中注册每个身份验证方案 `ConfigureServices` 方法`Startup.cs`。

`UseIdentity` 方法将替换 `UseAuthentication`。

下面的 2.0 示例将 Facebook 身份验证配置中标识`Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationContext>();

    // If you want to tweak Identity cookies, they're no longer part of IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
    services.AddAuthentication()
        .AddFacebook(options =>
    {
        options.AppId = Configuration["auth:facebook:appid"];
        options.AppSecret = Configuration["auth:facebook:appsecret"];
    });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory) {
    app.UseAuthentication();
}
```

`UseAuthentication` 方法将添加一个单一的身份验证中间件组件，它负责自动身份验证和远程身份验证请求的处理。它将替换所有单独的中间件组件与单个、常见的中间件组件。

以下是每个主要身份验证方案的 2.0 迁移说明。

基于 cookie 的身份验证

选择以下两个选项之一，然后进行必要的更改在`Startup.cs`:

1. 标识与使用 cookie

- 替换 `UseIdentity` 与 `UseAuthentication` 中 `Configure` 方法:

```
app.UseAuthentication();
```

- 调用 `AddIdentity` 中的方法 `ConfigureServices` 方法将添加 cookie 身份验证服务。

- (可选) 调用 `ConfigureApplicationCookie` 或 `ConfigureExternalCookie` 中的方法 `ConfigureServices` 方法来调整标识 cookie 设置。

```
services.AddIdentity< ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
```

2. 使用 cookie, 而无需标识

- 替换 `UseCookieAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddAuthentication` 和 `AddCookie` 中的方法 `ConfigureServices` 方法:

```
// If you don't want the cookie to be automatically authenticated and assigned to
// HttpContext.User,
// remove the CookieAuthenticationDefaults.AuthenticationScheme parameter passed to
// AddAuthentication.
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
{
    options.LoginPath = "/Account/LogIn";
    options.LogoutPath = "/Account/LogOff";
});
```

JWT 持有者身份验证

进行中的以下更改`Startup.cs`:

- 替换 `UseJwtBearerAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddJwtBearer` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
{
    options.Audience = "http://localhost:5001/";
    options.Authority = "http://localhost:5000/";
});
```

此代码段不使用标识，因此应通过将传递设置的默认方案 `JwtBearerDefaults.AuthenticationScheme` 到 `AddAuthentication` 方法。

OpenID 连接 (OIDC) 身份验证

进行中的以下更改 `Startup.cs`:

- 替换 `UseOpenIdConnectAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddOpenIdConnect` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

facebook 身份验证

进行中的以下更改 `Startup.cs`:

- 替换 `UseFacebookAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddFacebook` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
    .AddFacebook(options =>
{
    options.AppId = Configuration["auth:facebook:appid"];
    options.AppSecret = Configuration["auth:facebook:appsecret"];
});
```

Google 身份验证

进行中的以下更改 `Startup.cs`:

- 替换 `UseGoogleAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication`:

```
app.UseAuthentication();
```

- 调用 `AddGoogle` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
    .AddGoogle(options =>
{
    options.ClientId = Configuration["auth:google:clientid"];
    options.ClientSecret = Configuration["auth:google:clientsecret"];
});
```

Microsoft 帐户身份验证

进行中的以下更改`Startup.cs`:

- 替换 `UseMicrosoftAccountAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication` :

```
app.UseAuthentication();
```

- 调用 `AddMicrosoftAccount` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
    .AddMicrosoftAccount(options =>
{
    options.ClientId = Configuration["auth:microsoft:clientid"];
    options.ClientSecret = Configuration["auth:microsoft:clientsecret"];
});
```

Twitter 身份验证

进行中的以下更改`Startup.cs`:

- 替换 `UseTwitterAuthentication` 方法调用 `Configure` 方法替换 `UseAuthentication` :

```
app.UseAuthentication();
```

- 调用 `AddTwitter` 中的方法 `ConfigureServices` 方法:

```
services.AddAuthentication()
    .AddTwitter(options =>
{
    options.ConsumerKey = Configuration["auth:twitter:consumerkey"];
    options.ConsumerSecret = Configuration["auth:twitter:consumersecret"];
});
```

设置默认身份验证方案

在 1.x `AutomaticAuthenticate` 和 `AutomaticChallenge` 属性 `AuthenticationOptions` 基类用于设置在单一身份验证方案。没有良好方法来强制执行此。

在 2.0 中, 这两个属性已删除作为属性对各个 `AuthenticationOptions` 实例。它们可以在中配置 `AddAuthentication` 内的方法调用 `ConfigureServices` 方法`Startup.cs`:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme);
```

在前面的代码段中, 默认方案设置为 `CookieAuthenticationDefaults.AuthenticationScheme ("Cookie")`。

或者, 使用一个重载的版本的 `AddAuthentication` 方法以设置多个属性。在下面的重载的方法示例中, 默认方案设置为 `CookieAuthenticationDefaults.AuthenticationScheme`。或者可能在个人中指定的身份验证方案 `[Authorize]` 属性或授权策略。

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

如果下列条件之一为 true, 请在 2.0 中定义了默认的方案:

- 你希望用户可自动登录
- 你使用 `[Authorize]` 而无需指定方案的属性或授权策略

此规则的唯一例外是 `AddIdentity` 方法。此方法将为您和设置默认值进行身份验证以及与应用程序 cookie 质询方案添加 cookie `IdentityConstants.ApplicationScheme`。此外, 它将默认登录方案设置为外部 cookie `IdentityConstants.ExternalScheme`。

使用 `HttpContext` 身份验证扩展插件

`IAuthenticationManager` 接口是 1.x 身份验证系统的主入口点。它已替换为一组新的 `HttpContext` 中的扩展方法 `Microsoft.AspNetCore.Authentication` 命名空间。

例如, 1.x 项目引用 `Authentication` 属性:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

在 2.0 项目中, 导入 `Microsoft.AspNetCore.Authentication` 命名空间, 并删除 `Authentication` 属性引用:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows 身份验证 (HTTP.sys / IISIntegration)

有两种变体 Windows 身份验证:

1. 主机仅允许经过身份验证的用户
2. 主机允许同时匿名和身份验证的用户

上面所述的第一个变体不受 2.0 的更改。

2.0 更改的情况下, 会影响上面所述的第二个变体。例如, 你可能将以允许匿名用户到你的应用程序在 IIS 或 HTTP.sys 层在控制器级别但授权用户。在此方案中, 设置默认方案为 `IISDefaults.AuthenticationScheme` 中 `ConfigureServices` 方法 `Startup.cs`:

```
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

相应地, 如果设置的默认方案会使质询无法正常工作的授权请求。

IdentityCookieOptions 实例

2.0 更改的副作用是时切换到使用名为而不是 cookie 选项实例的选项。删除自定义标识 cookie 方案名称的能力。

例如, 1.x 项目使用 [构造函数注入](#) 传递 `IdentityCookieOptions` 参数转换 `AccountController.cs`。从提供的实例访问外部 cookie 身份验证方案:

```

public AccountController(
    UserManager< ApplicationUser > userManager,
    SignInManager< ApplicationUser > signInManager,
    IOptions< IdentityCookieOptions > identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme = identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger< AccountController >();
}

```

前面提到的构造函数注入将成为在 2.0 项目中，不必要和 `_externalCookieScheme` 可以删除字段：

```

public AccountController(
    UserManager< ApplicationUser > userManager,
    SignInManager< ApplicationUser > signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger< AccountController >();
}

```

`IdentityConstants.ExternalScheme` 可以直接使用常量：

```

// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

```

添加 POCO `IdentityUser` 导航属性

基的 Entity Framework (EF) 核心导航属性 `IdentityUser` POCO (普通旧 CLR 对象) 已被删除。如果你 1.x 的项目使用这些属性，手动将它们添加回 2.0 项目：

```

/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection< IdentityUserRole< int > > Roles { get; } = new List< IdentityUserRole< int > >();

/// <summary>
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection< IdentityUserClaim< int > > Claims { get; } = new List< IdentityUserClaim< int > >();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection< IdentityUserLogin< int > > Logins { get; } = new List< IdentityUserLogin< int > >();

```

若要防止重复的外键，运行 EF 核心迁移时，将以下代码添加到你 `IdentityDbContext` 类的 `OnModelCreating` 方法 (后 `base.OnModelCreating();` 调用)：

```

protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Identity model and override the defaults if needed.
    // For example, you can rename the ASP.NET Identity table names and more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);
}

```

替换 GetExternalAuthenticationSchemes

同步方法 `GetExternalAuthenticationSchemes` 已删除为支持的异步版本。1.x 项目有以下代码 `ManageController.cs`:

```

var otherLogins = _signInManager.GetExternalAuthenticationSchemes().Where(auth => userLogins.All(ul =>
auth.AuthenticationScheme != ul.LoginProvider)).ToList();

```

此方法将出现在 `Login.cshtml` 太:

```

var loginProviders = SignInManager.GetExternalAuthenticationSchemes().ToList();


@foreach (var provider in loginProviders)
{
    <button type="submit" class="btn btn-default" name="provider"
value="@provider.AuthenticationScheme" title="Log in using your @provider.DisplayName
account">@provider.AuthenticationScheme</button>
}


```

在 2.0 项目中, 使用 `GetExternalAuthenticationSchemesAsync` 方法:

```

var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name != ul.LoginProvider)).ToList();

```

在 `Login.cshtml`、`AuthenticationScheme` 中访问属性 `foreach` 循环更改为 `Name`:

```
var loginProviders = (await SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    <div>
        <p>
            @foreach (var provider in loginProviders)
            {
                <button type="submit" class="btn btn-default" name="provider" value="@provider.Name"
title="Log in using your @provider.DisplayName account">@provider.DisplayName</button>
            }
        </p>
    </div>
</form>
}
```

ManageLoginsViewModel 属性更改

A `ManageLoginsViewModel` 对象将用于 `ManageLogins` 操作 `ManageController.cs`。1.x 项目，该对象中 `OtherLogins` 属性的返回类型是 `IList<AuthenticationDescription>`。此返回类型需要导入 `Microsoft.AspNetCore.Http.Authentication`：

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationDescription> OtherLogins { get; set; }
    }
}
```

在 2.0 项目中，返回类型更改为 `IList<AuthenticationScheme>`。此新的返回类型需要替换 `Microsoft.AspNetCore.Http.Authentication` 使用导入 `Microsoft.AspNetCore.Authentication` 导入。

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

其他资源

有关其他详细信息和讨论，请参阅[针对身份验证 2.0 讨论 GitHub 上的问题](#)。

ASP.NET Core 2.0 中的新增功能

2018/5/17 • 8 min to read • [Edit Online](#)

本文重点介绍 ASP.NET Core 2.0 中最重要的更改，并提供相关文档的链接。

Razor 页面

Razor 页面是 ASP.NET Core MVC 的一个新功能，它可以使基于页面的编码方式更简单高效。

有关详细信息，请参阅相关介绍和教程：

- [Razor 页面介绍](#)
- [Razor 页面入门](#)

ASP.NET Core 元包

新的 ASP.NET Core 元包包含 ASP.NET Core 和 Entity Framework 团队生成和提供支持的所有包及其内部和第三方依赖项。无需再通过包选择单个 ASP.NET Core 功能。[Microsoft.AspNetCore.All](#) 包中包含所有的功能。默认模板使用此包。

有关详细信息，请参阅 [ASP.NET Core 2.0 的 Microsoft.AspNetCore.All 元包](#)。

运行时存储

使用 [Microsoft.AspNetCore.All](#) 元包的应用程序会自动使用新的 .NET Core 运行时存储。此存储包含运行 ASP.NET Core 2.0 应用程序所需的所有运行时资产。使用 [Microsoft.AspNetCore.All](#) 元包时，应用程序不会部署引用的 ASP.NET Core NuGet 包中的任何资产，因为目标系统中已存在这些资产。运行时存储中的资产也已经过预编译，以便缩短应用程序启动时间。

有关详细信息，请参阅 [运行时存储](#)

.NET Standard 2.0

ASP.NET Core 2.0 包面向 .NET Standard 2.0。这些包可以由其他 .NET Standard 2.0 库引用，也可以在兼容 .NET Standard 2.0 的 .NET 实现上运行，其中包括 .NET Core 2.0 和 .NET Framework 4.6.1。

[Microsoft.AspNetCore.All](#) 元包仅面向 .NET Core 2.0，因为它旨在与 .NET Core 2.0 运行时存储一起使用。

配置更新

在 ASP.NET Core 2.0 中，已默认将 [IConfiguration](#) 实例添加到服务容器。服务容器中的 [IConfiguration](#) 可以使应用程序更容易地从容器中检索配置值。

有关已规划文档的状态的信息，请参阅 [GitHub 问题](#)。

日志记录更新

在 ASP.NET Core 2.0 中，已默认将日志记录并入依存关系注入 (DI) 系统。在 Program.cs 文件（而非 Startup.cs 文件）中添加提供程序并配置筛选。此外，默认的 [ILoggerFactory](#) 支持进行筛选，并且你可以使用灵活的方式来跨提供程序筛选和特定于提供程序的筛选。

有关详细信息，请参阅 [日志记录介绍](#)。

身份验证更新

新的身份验证模型简化了使用 DI 为应用程序配置身份验证的过程。

使用 [Azure AD B2C] (<https://azure.microsoft.com/services/active-directory-b2c/>) 为 Web 应用和 Web API 配置身份验证时可使用新模板。

有关已规划文档的状态的信息，请参阅 [GitHub 问题](#)。

标识更新

在 ASP.NET Core 2.0 中，我们简化了使用标识生成安全的 Web API 的过程。可以使用 [Microsoft 身份验证库 \(MSAL\)](#) 获取用于访问 Web API 的访问令牌。

有关 2.0 中的身份验证更改的详细信息，请参阅以下资源：

- [ASP.NET Core 中的帐户确认和密码恢复](#)
- [为 ASP.NET Core 中的验证器应用启用 QR 码生成](#)
- [将身份验证和标识迁移到 ASP.NET Core 2.0](#)

SPA 模板

已提供适用于 Angular、Aurelia、Knockout.js、React.js 及 Redux 的单页应用程序 (SPA) 项目模板。Angular 模板已更新至 Angular 4。默认情况下，Angular 和 React 模板已可用；有关如何获取其他模板的信息，请参阅[新建 SPA 项目](#)。有关如何在 ASP.NET Core 中生成 SPA 的信息，请参阅[使用 JavaScriptServices 创建单页应用程序](#)。

Kestrel 改进

Kestrel Web 服务器包含一项新功能，使其更适合作为面向 Internet 的服务器。在 `KestrelServerOptions` 类的新 `Limits` 属性中添加大量服务器约束配置选项。为以下内容添加限制：

- 客户端最大连接数
- 请求正文最大大小
- 请求正文最小数据速率

有关详细信息，请参阅 [ASP.NET Core 中的 Kestrel Web 服务器实现](#)。

WebListener 已重命名为 HTTP.sys

`Microsoft.AspNetCore.Server.WebListener` 和 `Microsoft.Net.Http.Server` 包已合并为一个新包 `Microsoft.AspNetCore.Server.HttpSys`。命名空间已进行更新以保持一致。

有关详细信息，请参阅 [ASP.NET Core 中的 HTTP.sys Web 服务器实现](#)。

增强了 HTTP 标头支持

使用 MVC 传输 `FileStreamResult` 或 `FileContentResult` 时，现在可以选择对传输的内容设置 `ETag` 或 `LastModified` 日期。可以使用如下所示的代码在返回的内容上设置这些值：

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified: DateTime.UtcNow.AddSeconds(-5), entityTag:
entityTag);
```

返回给访问者的文件将附带 `ETag` 和 `LastModified` 值的适当 HTTP 标头。

如果应用程序访问者使用范围请求标头请求内容，ASP.NET 将识别出这一点，并会处理该标头。如果可以对请求的内容执行部分传输操作，ASP.NET 将适当地跳过一些内容，只返回请求的字节集。不必为了采用或处理此功能而将任何特殊的处理程序写入方法；系统会自动处理。

托管启动和 Application Insights

托管环境现在可以在应用程序启动时插入额外的包依赖项并执行代码，而应用程序无需显式使用依赖项或调用任何方法。可以使用此功能来允许某些环境“启用”该环境特有的功能，而应用程序无需提前获知。

在 ASP.NET Core 2.0 中，如果在 Visual Studio 中调试并且（选择加入后）在 Azure App Services 中运行，将使用此功能自动启用 Application Insights 诊断。因此，默认情况下，项目模板不再添加 Application Insights 包和代码。

有关已规划文档的状态的信息，请参阅 [GitHub 问题](#)。

自动使用防伪标记

默认情况下，ASP.NET Core 始终在帮助对内容进行 HTML 编码，但是在新版本中，还采用了额外的措施来帮助预防跨网站请求伪造（XSRF）攻击。现在在默认情况下，ASP.NET Core 会发出防伪标记，并在窗体 POST 操作和页面上验证它们，且无需其他配置。

有关详细信息，请参阅 [预防跨网站请求伪造（XSRF/CSRF）攻击](#)。

自动预编译

默认情况下，会在发布时启用 Razor 视图预编译，以缩减发布输出大小和应用程序启动时间。

有关详细信息，请参阅 [ASP.NET Core 中的 Razor 视图编译和预编译](#)。

Razor 支持 C# 7.1

Razor 视图引擎已更新为可使用新的 Roslyn 编译器。其中包含对 C# 7.1 功能的支持，例如默认表达式、推断元组名称和泛型模式匹配。若要在项目中使用 C# 7.1，请在项目文件中添加以下属性，然后重新加载解决方案：

```
<LangVersion>latest</LangVersion>
```

有关 C# 7.1 功能的状态的信息，请参阅 [Roslyn GitHub 存储库](#)。

2.0 的其他文档更新

- [用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件](#)
- [密钥管理](#)
- [配置 Facebook 身份验证](#)
- [配置 Twitter 身份验证](#)
- [配置 Google 身份验证](#)
- [配置 Microsoft 帐户身份验证](#)

迁移指南

有关如何将 ASP.NET Core 1.x 应用程序迁移到 ASP.NET Core 2.0 的指南，请参阅以下资源：

- [从 ASP.NET Core 1.x 迁移到 ASP.NET Core 2.0](#)
- [将身份验证和标识迁移到 ASP.NET Core 2.0](#)

其他信息

有关更改的完整列表，请参阅 [ASP.NET Core 2.0 发行说明](#)。

若要实时了解 ASP.NET Core 开发团队的进度和计划，请收看 [ASP.NET Community Standup](#)。

ASP.NET Core 1.1 的新增功能

2018/5/17 • 1 min to read • [Edit Online](#)

ASP.NET Core 1.1 新增了以下功能：

- [URL 重写中间件](#)
- [响应缓存中间件](#)
- [查看组件即标记帮助程序](#)
- [MVC 型中间件筛选器](#)
- [基于 Cookie 的 TempData 提供程序](#)
- [Azure App Service 日志记录提供程序](#)
- [Azure Key Vault 配置提供程序](#)
- [Azure 和 Redis 存储数据保护密钥存储库](#)
- [适用于 Windows 的 WebListener 服务器](#)
- [WebSockets 支持](#)

在 ASP.NET Core 的 1.0 和 1.1 版本之间进行选择

ASP.NET Core 1.1 比 1.0 功能更多。通常情况下，建议使用最新版本。

其他信息

- [ASP.NET Core 1.1.0 发行说明](#)
- 若要实时了解 ASP.NET Core 开发团队的进度和计划，请收看 [ASP.NET Community Standup](#)。