Table of Contents

ASP.NET Core 介绍

作者 Daniel Roth, Rick Anderson, 以及 Shaun Luttin

翻译 江振宇(Kerry Jiang)

校对 许登洋(Seay)、魏美娟(初见)、姚阿勇(Mr.Yao)

ASP.NET Core 是对 ASP.NET 的一次意义重大的重构。本文介绍了 ASP.NET Core 中的一些新概念，并且解释了它们如何帮助你开发现代的 Web 应用程序。

什么是 **ASP.NET Core**?

ASP.NET Core 是一个新的开源和跨平台的框架，用于构建如 Web 应用、物联网（IoT）应用和移动后端应用等连接到互联网的基于云的现代应用程序。ASP.NET Core 应用可运行于 .NET Core 和完整的 .NET Framework 之上。 构建它的目的是为那些部署在云端或者内部运行（on-premises）的应用提供一个优化的开发框架。它由最小开销的模块化的组件构成，因此在构建你的解决方案的同时可以保持灵活性。你可以在 Windows、Mac 和 Linux 上跨平台的开发和运行你的 ASP.NET Core 应用。 ASP.NET Core 开源在 GitHub 上。

为什么构建 **ASP.NET Core**?

ASP.NET 的首个预览版作为 .NET Framework 的一部分发布于15年前。自那以后数百万的开发者用它开发和运行着众多非常棒的 Web 应用，而且在这么多年之间我们也为它增加和改进了很多的功能。

ASP.NET Core 有一些架构上的改变，这些改变会使它成为一个更为精简并且模块化的框架。ASP.NET Core 不再基于 *System.Web.dll* 。当前它基于一系列颗粒化的，并且良好构建的 NuGet 包。这一特点能够让你通过仅仅包含需要的 NuGet 包的方法来优化你的应用。一个更小的应用程序接口通过"只为你需要的功能付出"（pay-for-what-you-use）的模型获得的好处包括更可靠的安全性、简化服务、改进性能和减少成本。

通过使用 ASP.NET Core，你可以获得以下改进：

统一的方式构建 web 界面 和 web APIs

集成 现代的客户端开发框架 以及开发流程

适用于云的，基于环境的 配置系统

内置 依赖注入

新型的轻量级的、模块化 HTTP 请求管道

运行于 IIS 或者自宿主（self-host）于你自己的进程的能力

基于支持真正的 side-by-side 应用程序版本化的 .NET Core 构建

完全以 NuGet 包的形式发布

新的用于简化现代 web 开发的工具

可以在 Windows 、Mac 和 Linux 上构建和运行跨平台的 ASP.NET 应用

开源并且关注社区

使用 **ASP.NET Core MVC** 构建 **web APIs** 以及 **web** 界面

您可以构建出能够适应更广泛的客户端（包括浏览器和移动设备）的HTTP服务。内置 多种数据格式和内容协商 支持 。 ASP.NET Core 基于.NET Core 运行时上构建 Web API 和 RESTful 风格应用的理想平台。参考 构建 web API。

你可以遵循 模型-视图-控制器 (MVC) 模式 来创建可测试以及设计良好的 Web应用程序。参考 MVC 以及 测试。

Razor 提供了一个创造性的语言来编写 Views

Tag Helpers 能使用服务器端代码在 Razor 文件中部分创建以及呈现 HTML 元素

Model 绑定 自动把 HTTP 请求的数据映射到 action 方法参数

Model 验证 执行执行客户端以及服务器端验证

客户端开发

ASP.NET Core 设计上无缝的集成了各种客户端框架， 包括 AngularJS, KnockoutJS 以及 Bootstrap。参考 客户端开发 获取更多细节。

下一步

继续开始学习教程， 参考 ASP.NET Core 指南

需要了解 ASP.NET Core 更深入的概念架构介绍，参考 ASP.NET Core 原理。

ASP.NET Core 应用程序可以使用 .NET Core 或者 .NET Framework 运行时。 更多信息请参考 选择 .NET Core 还是 .NET Framework.

如果你想了解 ASP.NET Core 开发团队的进度和计划， 随时访问 ASP.NET Community Standup.

ASP.NET Core 入门

翻译 娄宇(Lyrics)

校对 刘怡(AlexLEWIS)

安装 .NET Core

创建一个新的 .NET Core 项目：

```
mkdir aspnetcoreapp
cd aspnetcoreapp
dotnet new web
```

注意:

在 macOS 或者 Linux 系统上，打开一个终端窗口。 在 Windows 系统上， 打开命令行窗口。
早期版本的 .NET Core 需要一个 `t` 参数，比如 `dotnet new -t web`。 如果你运行 `dotnet new web` 出错， 安装最新版本的 .NET Core. 输入 `dotnet` (不需要参数) 将会显示 .NET Core 版本。

还原包：

```
dotnet restore
```

运行应用程序（ `dotnet run` 命令在应用程序过期（配置或代码发生变更）时重新生成它）：

```
dotnet run
```

浏览 `http://localhost:5000`

下一步

想学习更多的入门指南， 参考 ASP.NET Core 指南

想了解 ASP.NET Core 概念和架构，参看 ASP.NET Core 介绍 以及 ASP.NET Core 原理.

ASP.NET Core 应用可以使用 .NET Core 或者 .NET Framework 运行时。更多信息，请参考 选择 .NET Core 还是 .NET Framework.

用 Visual Studio Code 在 mac 或者 linux 上创建首个 ASP.NET Core 应用程序

翻译 赵志刚

校对 何镇汐、刘怡(AlexLEWIS)

本节将展示如何在 macOS 或者 Linux 平台上创建首个 ASP.NET Core 应用程序。

配置开发环境

在开发机中下载并安装 .NET Core 以及 Visual Studio Code 并且安装 C# 扩展.

使用 **dotnet new** 来构建程序基架

我们可以使用 `dotnet new` 调用 "Empty Web Template" 模版来生成一个新的 Web 应用程序。 在你的项目中创建一个名为 *firstapp* 的工作目录。跳转到 *firstapp*。

启动 Visual Studio Code 并且代开 *firstapp* 目录。 点击 Ctrl + '`' (反引号字符) 打开 VS Code 内置终端。 你也可以使用独立终端窗口。 运行 `dotnet new` 命令创建一个新闻Web 应用程序， 传递 `mvc` 参数作为模版类型。

```
dotnet new mvc
```

如果你运行 `dotnet new mvc` 时遇到错误，安装最新的 .NET Core。当 CLI 命令运行完毕以后， 会产生以下输出和文件：

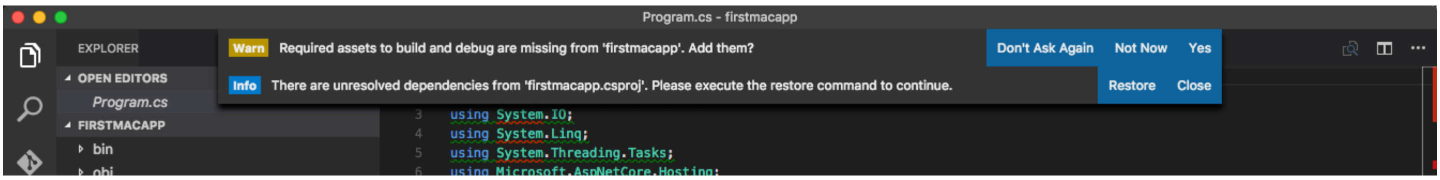
```
Content generation time: 79.6691 ms
The template "Empty ASP.NET Core Web Application" created successfully.
```

Startup.cs : Startup 类 - 配置请求管道的类，处理所有所有由应用程序产生的管道请求。
Program.cs : Program 类 应用程序的主入口点。
firstapp.csproj : Project 文件 ASP.NET Core 应用程序的 MSBuild 格式项目文件。包含项目间的引用， NuGet引用以及其他项目关联项。
appsettings.json / appsettings.Development.json : 基于环境的应用配置文件。 参考配置。
bower.json : 项目 Bower 依赖包。
.bowerrc : 当 Bower 下载文件时用来定义组件安装位置的配置文件。
bundleconfig.json : 设置前端 JavaScript 和 CSS 文件压缩绑定的配置文件。
Views : 包含 Razor 视图，视图是显示应用程序用户界面的组件，用户界面会呈现模型数据。
Controllers : 包含 MVC 控制器， 默认控制器为 *HomeController.cs*。控制器是处理请求的类。
wwwroot : Web 应用程序根目录。

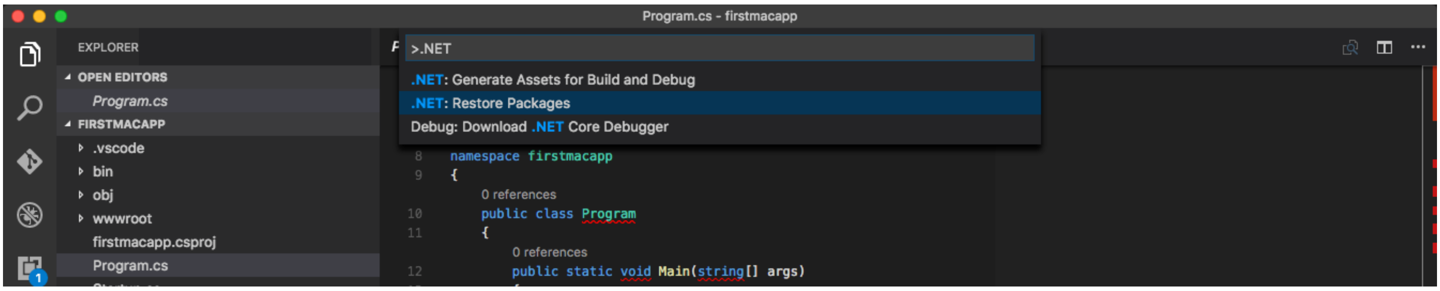
更多信息请参考 The MVC 模式.

使用 **Visual Studio Code** 在 **Mac** 或者 **Linux** 环境下开发 **ASP.NET Core** 应用程序

使用 Visual Studio Code (VS Code) 打开项目目录选择 *Startup.cs* 文件。 VS Code 会弹出需要还原项目依赖以及添加 build/debug 依赖。点击 **Yes** 来添加 build 以及 debug 附件，点击 **Restore** 来还原项目依赖。



除了 **Restore**， 你也可以在终端使用 `dotnet restore` 或者像如下所示在VS Code输入 `⌘⇧P` 或 `Ctrl+Shift+P` 点击 `.NET` ：



VS Code 提供了一个用于处理文件的流式，简洁的界面和高效的编码环境。

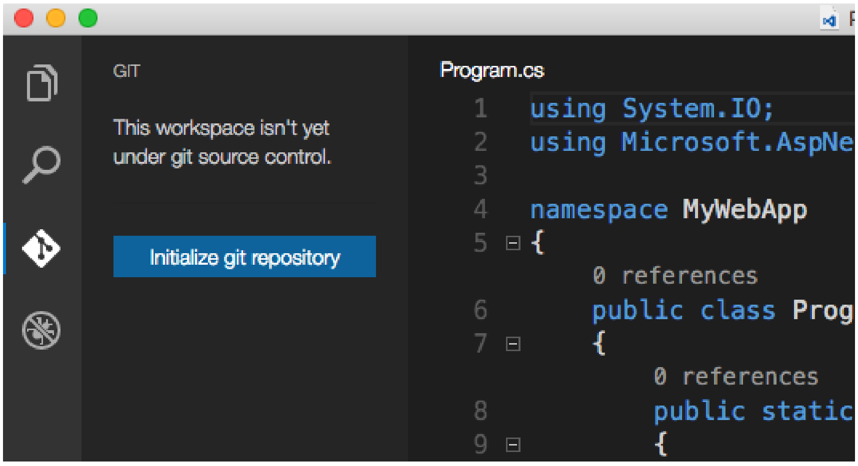
在右侧导航栏，有5个图标，代表5个面板：

浏览
搜索
Git
调试
扩展

浏览面板提供你打开的文件目录导航。如果你有未保存文件会显示一个徽章。你可以在视图中创建新的文件和文件夹. 当鼠标移动到上面的时候你可以选择 **Save All** 菜单。

搜索面板允许你查询当前打开的目录树中的文件。 支持检索文件名和文件内容。

如果你的系统中安装了 GIT *VS Code* 会自动集成。 你可以在 Git 面板中初始化代码仓库，提交修改，推送变更。



调试面板支持应用程序的交互式调试。

VS Code 的编辑器还提供了一些非常棒的特性，比如你会注意到未使用的 using 语句会带有下划线，当出现电灯图标时可使用 `⌘ .` 或者 `Ctrl + .` 自动移除之。类和方法同样可显示在本项目中的引用次数。

更多编辑器请参考 Visual Studio Code.

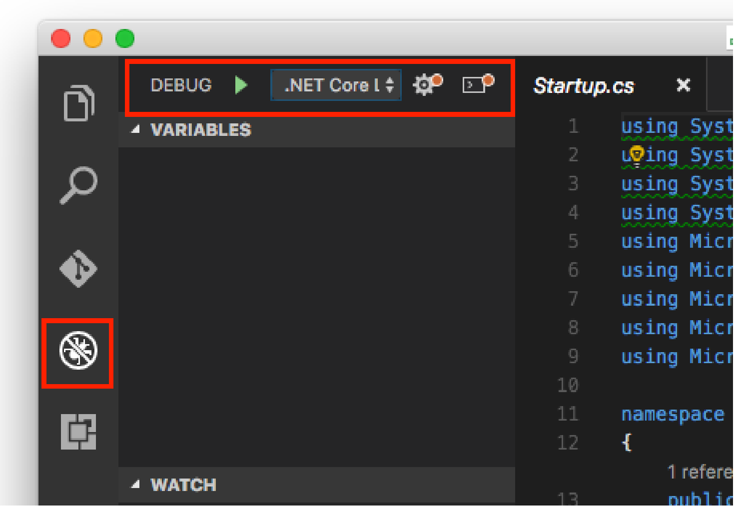
使用 VS Code 调试

本示例配置使用 Kestrel 作为 Web 服务器。

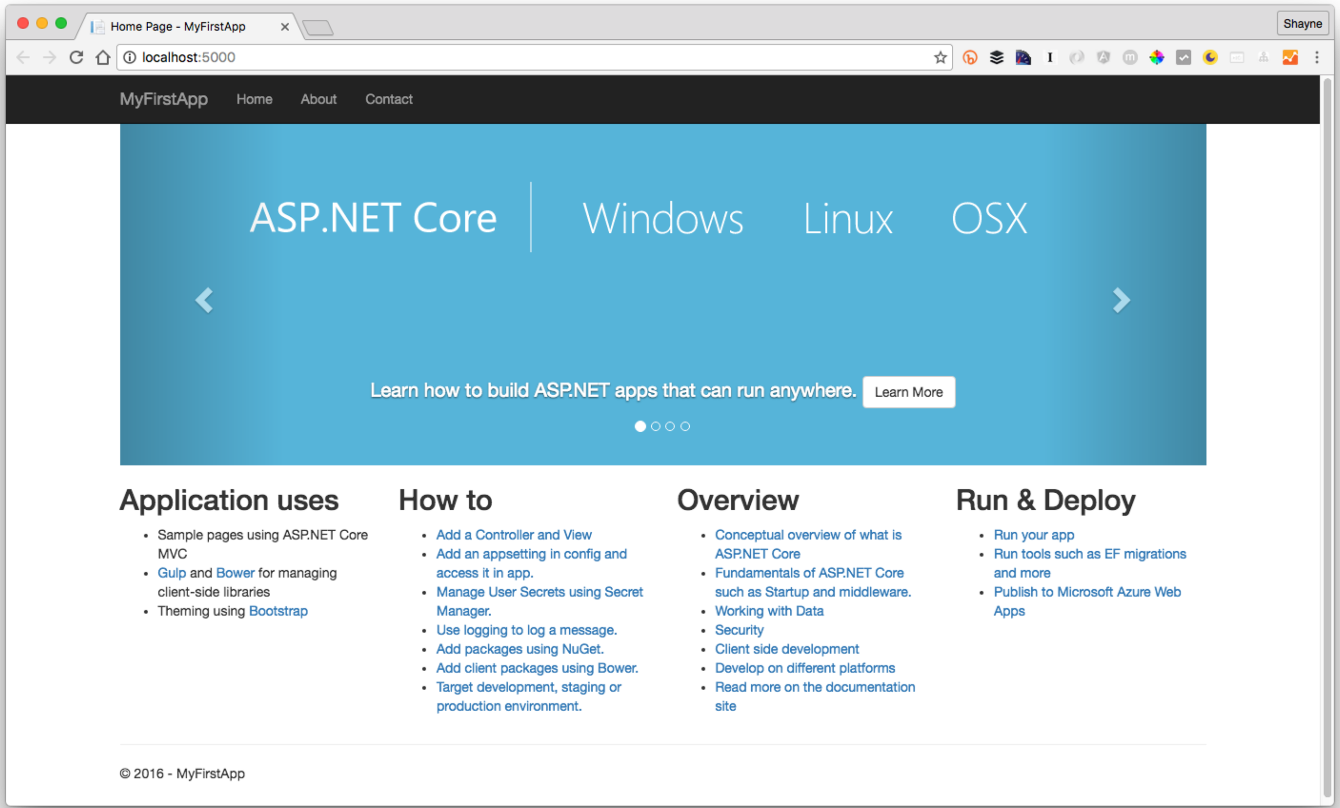
在调试面板中运行应用程序：

在右侧面板视图栏点击调试图标

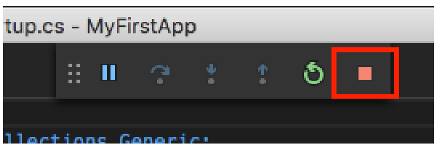
点击 "运行 (F5)" 图标来启动程序



你的浏览器会自动启动并且导航到 `http://localhost:5000`



停止运行程序， 可以关闭浏览器并点击调试栏的 "停止" 图标



使用 dotnet 命令

运行 `dotnet run` 命令从 终端/bash 启动应用程序

浏览 `http://localhost:5000`

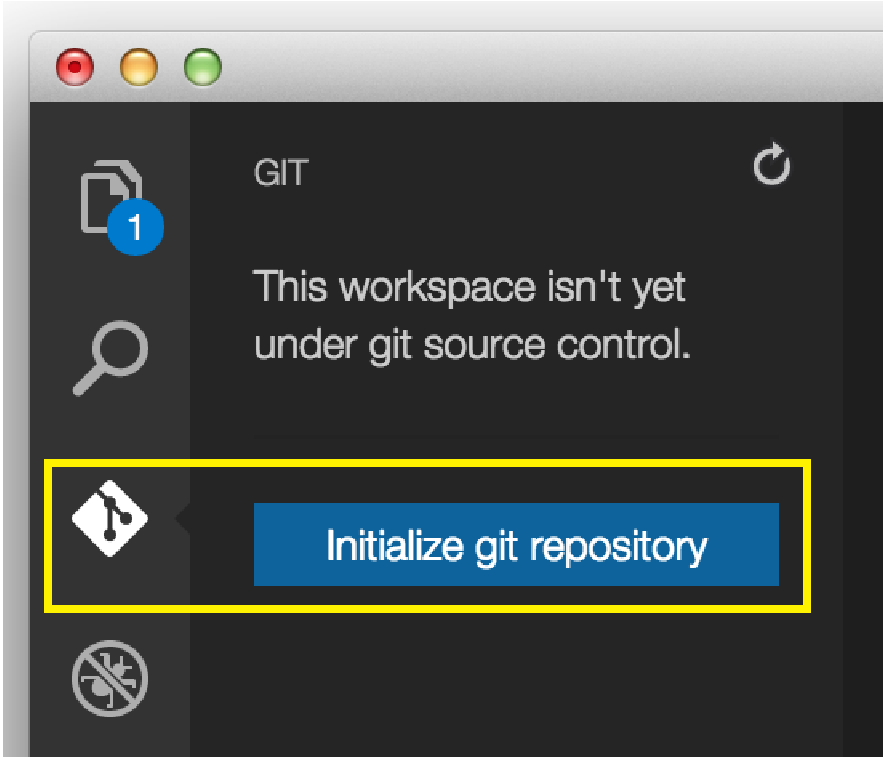
按下 `⌘+C` or `Ctrl+C` 停止 Web 服务。

发布到 Azure

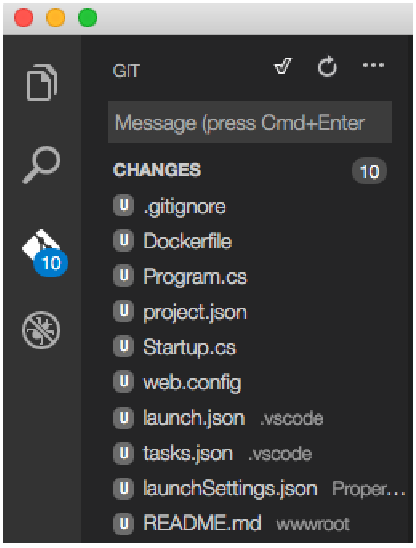
一旦在 Microsoft Azure 部署了你的应用程序，你便可轻松地通过 Visual Studio Code 中集成的 GIT 功能将产品的更新推送到生产环境。

初始化 Git

为你的工作文件夹初始化 GIT。切换到Git视图 然后点击 Initialize Git repository 按钮。



填写提交信息并点击提交，或点击复选框来提交暂存文件。



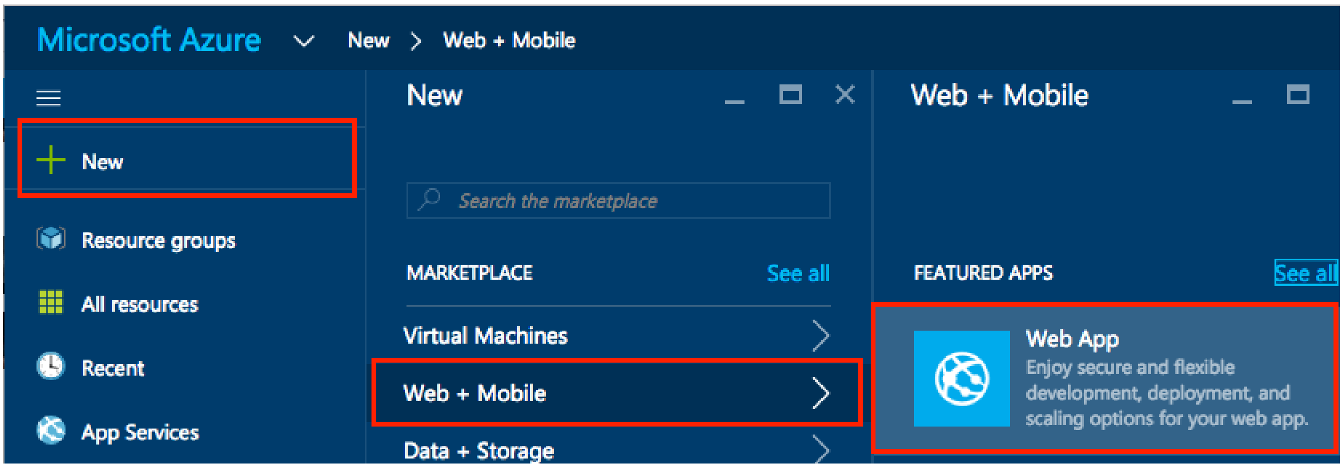
GIT 会跟踪变更，所以如果你更新了文件，Git 面板将显示上次提交之后修改过的文件。

Initialize Azure Website

通过 git 将应用程序直接部署到 Azure。

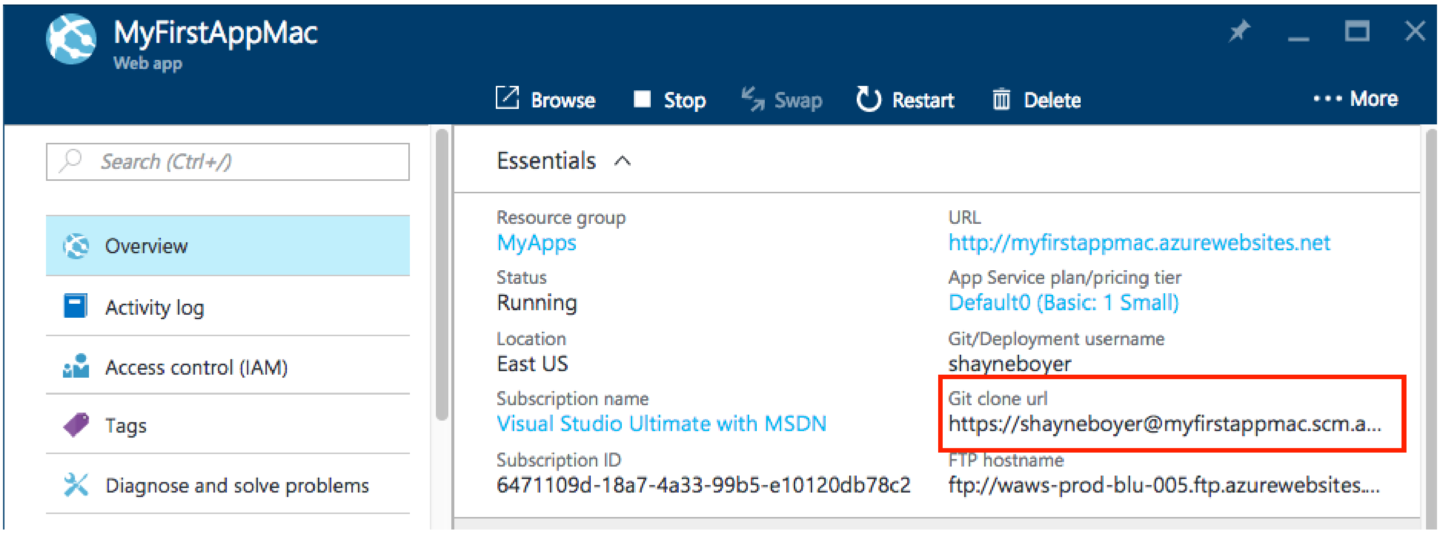
如果你没有 Azure 账号，你可以免费创建一个试用账号。

在 Azure 门户中创建一个 Web 应用来托管你的新的应用程序。



配置 Azure Web 应用程序支持使用 Git 持续部署。

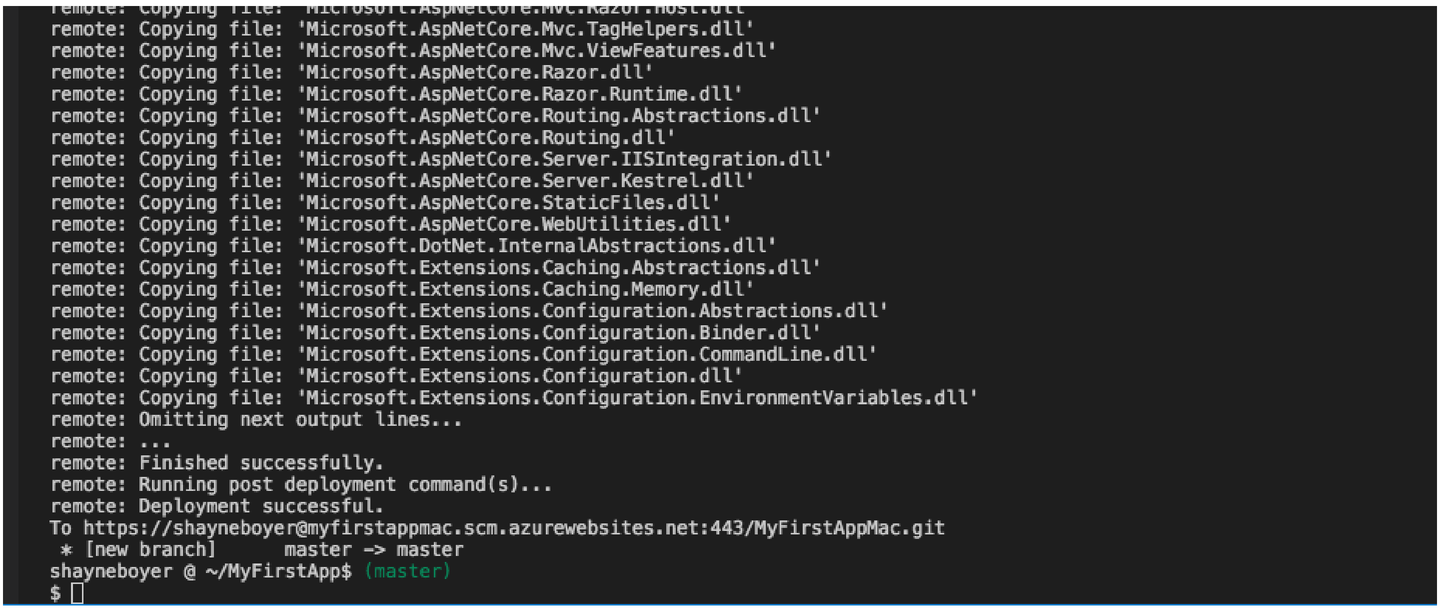
将此 Web 应用程序在 Azure 中的 Git URL 记录下来:



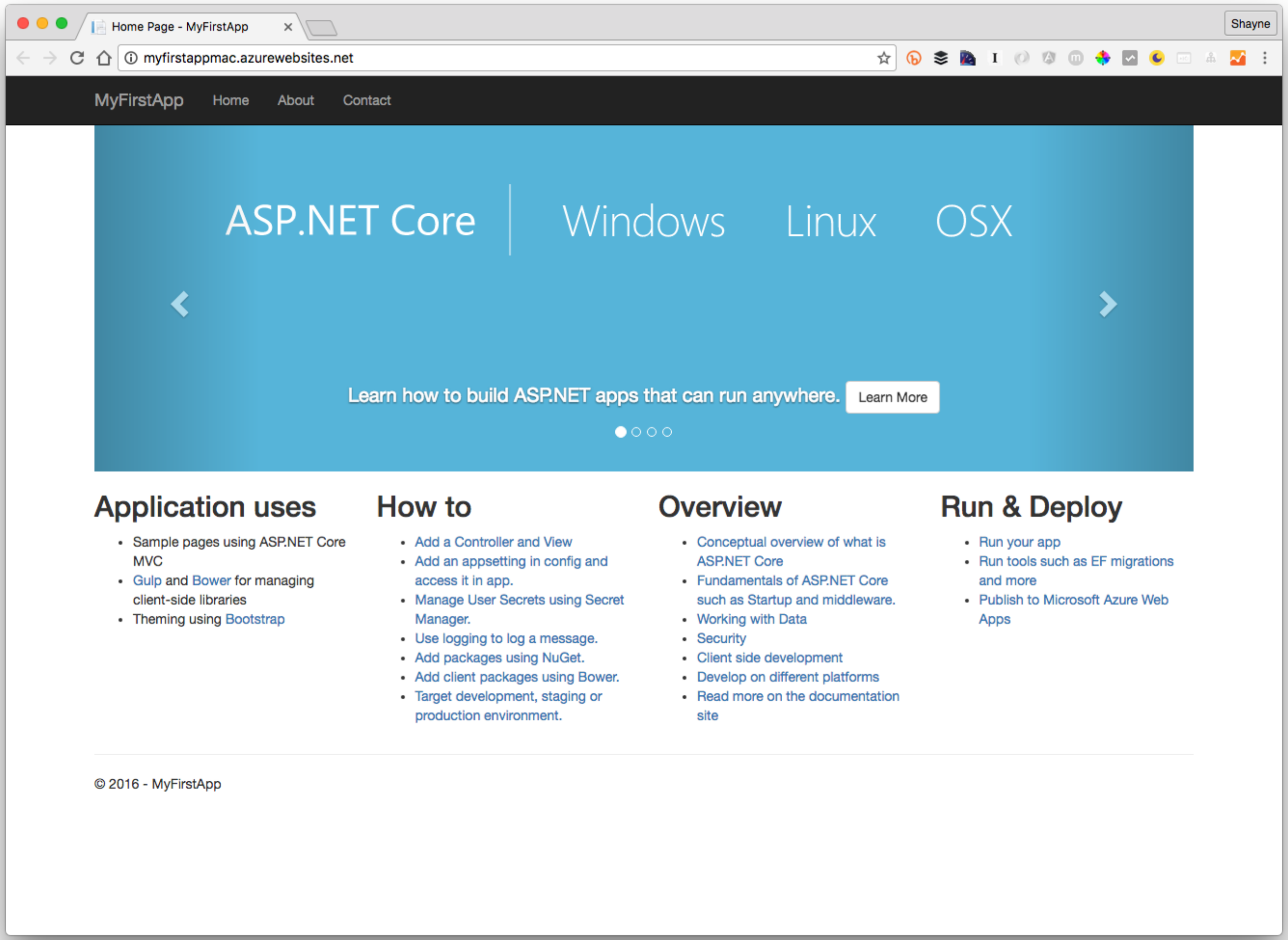
在终端窗口中，用之前记下的 Git URL 新建一个名为 `azure` 的远程主机。

```
git remote add azure https://shayneboyer@myfirstappmac.scm.azurewebsites.net:443/MyFirstAppMac.git
```

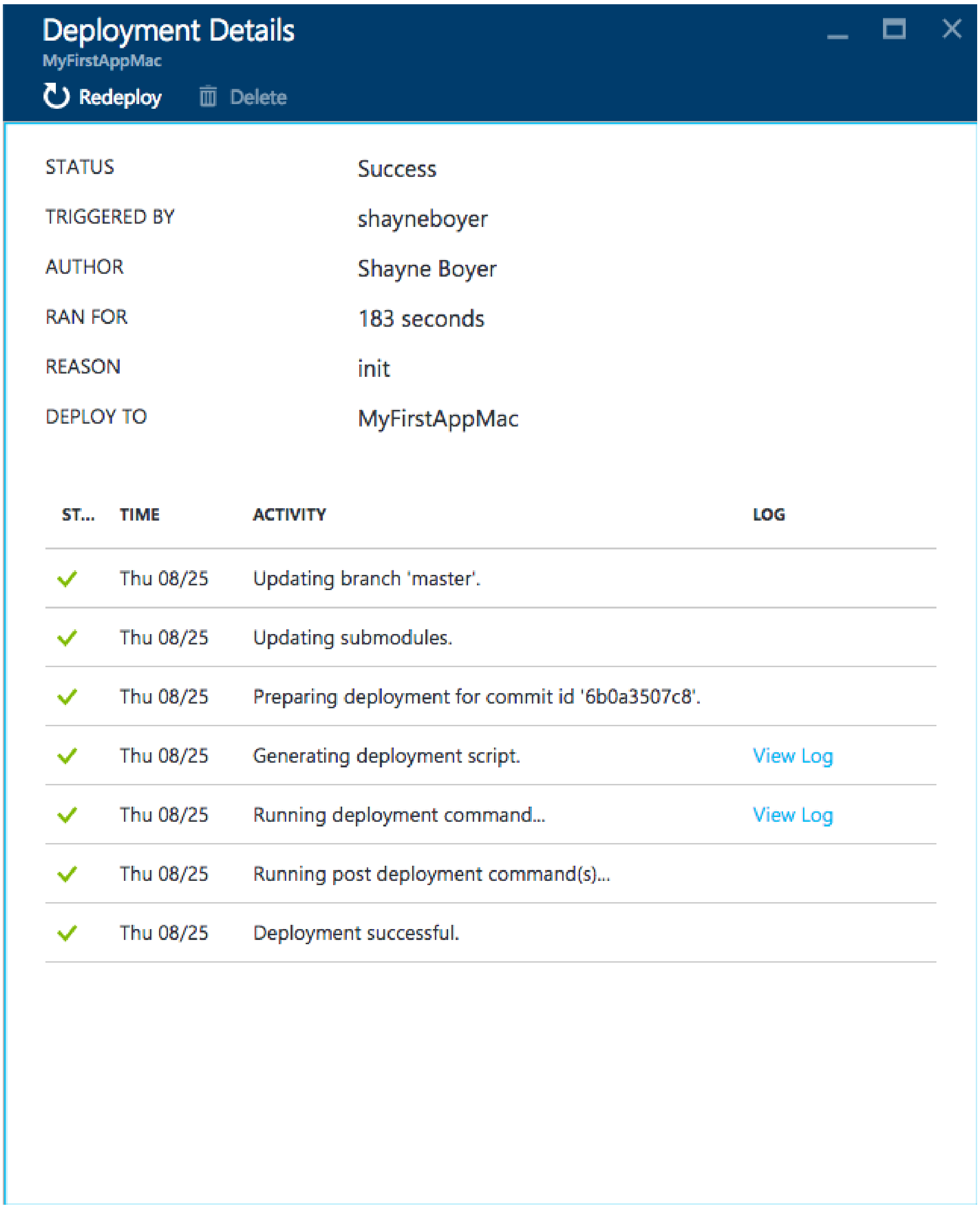
推送到 master 分支。部署： `git push azure master`。



浏览刚才部署的 Web 应用程序。



在 Azure 门户中查看部署细节，你可以看到每个步骤的时间以及分支被提交了一次。



其他资源

[Visual Studio Code](#)

[原理](#)

使用 Visual Studio 创建 ASP.NET Core MVC 应用程序

本系列教程将教会你使用 Visual Studio 创建基本的 ASP.NET Core MVC 应用程序

使用 dotnet watch 开发 ASP.NET Core 应用程序

作者 Rick Anderson 、 Victor Hurdugaci

翻译 谢炀（Kiler)

校对 刘怡(AlexLEWIS)、许登洋(Seay)

`dotnet watch` 是一个开发阶段在源文件发生变动的情况下使用 `dotnet` 命令的工具。 当代码发生变动的时候可以用来执行编译，运行测试，或者发布操作。

在本教程中，我们将使用一个现有的计算两个数字之和以及乘积的 WebApi 应用程序。示例应用程序故意包含一个错误，作为本教程的一部分我们会修复它。

开始下载 示例应用程序。示例程序包含两个项目， `WebApp` （Web 应用程序）以及 `WebAppTests` （Web 应用程序配套的单元测试项目）

在命令行控制台中，进入下载示例程序的目录并且运行下述命令：

`dotnet restore`
`dotnet run`

控制台输出将显示如下信息，表明该应用程序正在运行并等待请求：

```
$ dotnet run
Hosting environment: Production
Content root path: C:/Docs/aspnetcore/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

在 Web 浏览器中，导航到 `http://localhost:5000/api/math/sum?a=4&b=5` 页面你会看到结果 `9` 。

如果你导航到乘法API页面 (`http://localhost:5000/api/math/product?a=4&b=5`) 页面，你期望得到结果 `20`。但是实际上还是返回了 `9`，我们稍后会修复这个问题 。

在项目中添加 `dotnet watch`

添加 `Microsoft.DotNet.Watcher.Tools` 到 *.csproj* 文件：

```
<ItemGroup>
 <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="1.0.0" />
</ItemGroup>
```

运行 `dotnet restore`。

以 `dotnet watch` 的方式运行 `dotnet` 命令

任何与 `dotnet` 有关的命令都可以以 `dotnet watch` 这样的方式运行：例如：

命令	WATCH 方式运行
dotnet run	dotnet watch run
dotnet run -f net451	dotnet watch run -f net451
dotnet run -f net451 -- --arg1	dotnet watch run -f net451 -- --arg1

命令	WATCH 方式运行
dotnet test	dotnet watch test

在 `WebApp` 目录里面运行 `dotnet watch run` 。 控制台输出将显示如下信息，表明 `watch`（监控工作）已经开始了。

在 `dotnet watch` 模式修改代码

确认 `dotnet watch` 模式运行中。

修复 `MathController` 中的 `Product` 方法的 bug ，让它返回乘积结果而不是总和。

```
public static int Product(int a, int b)
{
  return a * b;
}
```

保存文件。 控制台输出将显示如下信息，表明 `dotnet watch` 检测到文件的改变并重启了应用程序。

验证 `http://localhost:5000/api/math/product?a=4&b=5` 链接返回正确的结果。

使用 `dotnet watch` 运行测试

修改 `MathController` 中的 `Product` 方法回到之前的返回总和结果并且保存文件。
在 windows 命令窗口，导航到 `WebAppTests` 目录。
运行 `dotnet restore`

运行 `dotnet watch test`。你会看到输出显示测试失败并且监控器在等待文件改变：

```
Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
Test Run Failed.
```

修复 `Product` 方法代码让他返回乘积结果。保存文件。

`dotnet watch` 侦测到文件改变并且返回测试。 命令行输出显示测试通过。

dotnet-watch GitHub 开源代码

dotnet-watch 是做为 GitHub 上的 DotNetTools repository 开源项目中的一部分。

dotnet-watch 说明 中的 MSBuild 章节 中详细解释了如何把 dotnet-watch 配置到 MSBuild 项目文件中来监控文件的变化。
dotnet-watch 说明 一文包含了所有本教程没有提到的 dotnet-watch 信息。

ASP.NET Core 指南

以下教程将逐步引导您开发一个 ASP.NET Core 应用程序：

构建 Web 应用

在 mac 或者 linux 上使用 Visual Studio Code 开发 ASP.NET Core MVC 应用程序
用 Visual Studio 创建 ASP.NET Core MVC 应用程序
使用 Visual Studio 开发 ASP.NET Core 和 Entity Framework Core 入门
用 Visual Studio Code 在 mac 或者 linux 上创建首个 ASP.NET Core 应用程序
使用 Yeoman 生成项目
制作 Tag Helpers
创建一个简单的视图组件
使用 dotnet watch 开发 ASP.NET Core 应用程序

构建 web APIs

在 Mac 系统中使用 Visual Studio 以及 ASP.NET Core 开发 Web API 项目
在 Windows 系统中使用 Visual Studio 以及 ASP.NET Core 开发 Web API 项目
使用 Visual Studio Code 以及 ASP.NET Core 开发 Web API 项目
使用 Swagger 作为 ASP.NET Web API 文档页面
为原生移动应用程序创建后台服务

处理数据

使用 Visual Studio 开发 ASP.NET Core 和 Entity Framework Core 入门
ASP.NET Core 与 EF Core - 新建数据库
ASP.NET Core 与 EF Core - 现有数据库

验证和授权

允许 Facebook，Google 以及其他外部提供者的验证方式
客户确认和密码重置
短信二次验证

客户端开发

使用 Gulp
使用 Grunt
使用 Bower 管理客户端包
使用 Bootstrap 构建美观、响应式网站

测试

在 .NET Core 中使用 dotnet test 单元测试

发布与部署

使用 Visual Studio 把 ASP.NET Core Web 应用程序发布到
使用持续发布发布到 Azure Web 应用
把 ASP.NET 容器发布到一个远程的 Docker 托管
VSTS 使用持续发布编译发布到 Azure Web 应用
ASP.NET Core 部署到 Nano Server

如何下载示例

下载 ASP.NET 仓库压缩代码。
解压 *Docs-master.zip* 文件。
使用示例链接中的 URL 帮你导航到示例目录。

ASP.NET Core fundamentals overview

An ASP.NET Core app is simply a console app that creates a web server in its `Main` method:

```
using System;
using Microsoft.AspNetCore.Hosting;

namespace aspnetcoreapp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

`Main` uses `WebHostBuilder`, which follows the builder pattern, to create a web application host. The builder has methods that define the web server (for example `UseKestrel`) and the startup class (`UseStartup`). In the example above, the Kestrel web server is used, but other web servers can be specified. We'll show more about `UseStartup` in the next section. `WebHostBuilder` provides many optional methods, including `UseIISIntegration` for hosting in IIS and IIS Express, and `UseContentRoot` for specifying the root content directory. The `Build` and `Run` methods build the `IWebHost` object that will host the app and start it listening for incoming HTTP requests.

Startup

The `UseStartup` method on `WebHostBuilder` specifies the `Startup` class for your app.

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseStartup<Startup>()
            .Build();

        host.Run();
    }
}
```

The `Startup` class is where you define the request handling pipeline and where any services needed by the app are configured. The `Startup` class must be public and contain the following methods:

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

    public void Configure(IApplicationBuilder app)
    {
    }
}
```

`ConfigureServices` defines the services (see Services below) used by your app (such as the ASP.NET MVC Core framework, Entity Framework Core, Identity, etc.)

`Configure` defines the middleware in the request pipeline

For more information, see Application startup.

Services

A service is a component that is intended for common consumption in an application. Services are made available through dependency injection (DI). ASP.NET Core includes a simple built-in inversion of control (IoC) container that supports constructor injection by default. The built-in container can be easily replaced with your container of choice. In addition to its loose coupling benefit, DI makes services available throughout your app. For example, logging is available throughout your app.

For more information, see Dependency injection .

Middleware

In ASP.NET Core you compose your request pipeline using Middleware. ASP.NET Core middleware performs asynchronous logic on an `HttpContext` and then either invokes the next middleware in the sequence or terminates the request directly. You generally "Use" middleware by taking a dependency on a NuGet package and invoking a corresponding `UseXYZ` extension method on the `IApplicationBuilder` in the `Configure` method.

ASP.NET Core comes with a rich set of built-in middleware:

Static files

Routing

Authentication

You can use any OWIN-based middleware with ASP.NET Core, and you can write your own custom middleware.

For more information, see Middleware and Open Web Interface for .NET (OWIN).

Servers

The ASP.NET Core hosting model does not directly listen for requests; rather it relies on an HTTP server implementation to forward the request to the application. The forwarded request is wrapped as a set of feature interfaces that the application then composes into an `HttpContext`. ASP.NET Core includes a managed cross-platform web server, called Kestrel that you would typically run behind a production web server like IIS or nginx.

For more information, see Servers and Hosting.

Content root

The content root is the base path to any content used by the app, such as its views and web content. By default the content root is the same as application base path for the executable hosting the app; an alternative location can be specified with *WebHostBuilder*.

Web root

The web root of your app is the directory in your project for public, static resources like css, js, and image files. The static files middleware will only serve files from the web root directory (and sub-directories) by default. The web root path defaults to */wwwroot*, but you can specify a different location using the *WebHostBuilder*.

Configuration

ASP.NET Core uses a new configuration model for handling simple name-value pairs. The new configuration model is not based on `System.Configuration` or *web.config*; rather, it pulls from an ordered set of configuration providers. The built-in configuration providers support a variety of file formats (XML, JSON, INI) and environment variables to enable environment-based configuration. You can also write your own custom configuration providers.

For more information, see Configuration.

Environments

Environments, like "Development" and "Production", are a first-class notion in ASP.NET Core and can be set using environment variables.

For more information, see Working with Multiple Environments.

.NET Core vs. .NET Framework runtime

An ASP.NET Core app can use the .NET Core or .NET Framework runtime. For more information, see Choosing between .NET Core and .NET Framework.

Additional information

See also the following topics:

Logging
Error Handling
Globalization and localization
File Providers
Managing Application State

Application Startup in ASP.NET Core

By Steve Smith and Tom Dykstra

The `Startup` class configures the request pipeline that handles all requests made to the application.

The Startup class

ASP.NET Core apps require a `Startup` class. By convention, the `Startup` class is named "Startup". You specify the startup class name in the `Main` programs WebHostBuilderExtensions `UseStartup<TStartup>` method.

You can define separate `Startup` classes for different environments, and the appropriate one will be selected at runtime. If you specify `startupAssembly` in the WebHost configuration or options, hosting will load that startup assembly and search for a `Startup` or `Startup[Environment]` type. See FindStartupType in `StartupLoader` and Working with multiple environments. `UseStartup<TStartup>` is the recommended approach.

The `Startup` class constructor can accept dependencies that are provided through dependency injection. You can use `IHostingEnvironment` to set up configuration sources and `ILoggerFactory` to set up logging providers.

The `Startup` class must include a `Configure` method and can optionally include a `ConfigureServices` method, both of which are called when the application starts. The class can also include environment-specific versions of these methods.

Learn about handling exceptions during application startup.

The Configure method

The `Configure` method is used to specify how the ASP.NET application will respond to HTTP requests. The request pipeline is configured by adding middleware components to an `IApplicationBuilder` instance that is provided by dependency injection.

In the following example from the default web site template, several extension methods are used to configure the pipeline with support for BrowserLink, error pages, static files, ASP.NET MVC, and Identity.

```
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?}");
    });
}
```

Each `Use` extension method adds a middleware component to the request pipeline. For instance, the `UseMvc` extension method adds the routing middleware to the request pipeline and configures MVC as the default handler.

For more information about how to use `IApplicationBuilder`, see Middleware.

Additional services, like `IHostingEnvironment` and `ILoggerFactory` may also be specified in the method signature, in which case these services will be injected if they are available.

The ConfigureServices method

The ConfigureServices method is optional; but if used, it's called before the `Configure` method by the runtime (some features are added before they're wired up to the request pipeline). Configuration options are set in this method.

For features that require substantial setup there are `Add[Service]` extension methods on IServiceCollection. This example from the default web site template configures the app to use services for Entity Framework, Identity, and MVC:

```
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>();
}
```

Adding services to the services container makes them available within your application via dependency injection.

Services Available in Startup

ASP.NET Core dependency injection provides application services during an application's startup. You can request these services by including the appropriate interface as a parameter on your `Startup` class's constructor or one of its `Configure` or `ConfigureServices` methods.

Looking at each method in the `Startup` class in the order in which they are called, the following services may be requested as parameters:

In the constructor: `IHostingEnvironment`, `ILoggerFactory`
In the `ConfigureServices` method: `IServiceCollection`
In the `Configure` method: `IApplicationBuilder`, `IHostingEnvironment`, `ILoggerFactory`, `IApplicationLifetime`

Additional Resources

Working with Multiple Environments
Middleware
Logging
Configuration

ASP.NET Core Middleware Fundamentals

By Rick Anderson and Steve Smith

View or download sample code

What is middleware

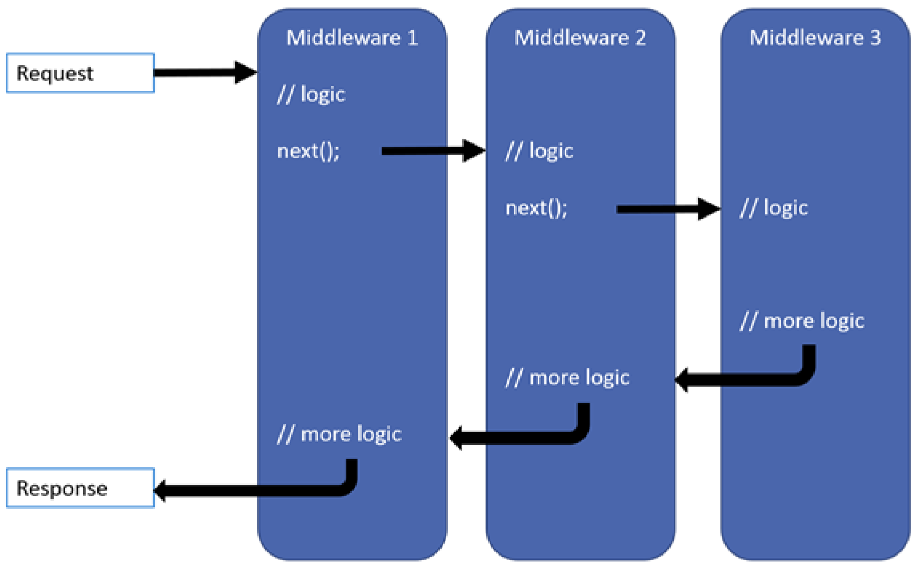
Middleware is software that is assembled into an application pipeline to handle requests and responses. Each component chooses whether to pass the request on to the next component in the pipeline, and can perform certain actions before and after the next component is invoked in the pipeline. Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods on the IApplicationBuilder instance that is passed into the `Configure` method in the `Startup` class. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, or *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline, or short-circuiting the chain if appropriate.

Migrating HTTP Modules to Middleware explains the difference between request pipelines in ASP.NET Core and the previous versions and provides more middleware samples.

Creating a middleware pipeline with IApplicationBuilder

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other, as this diagram shows (the thread of execution follows the black arrows):



Each delegate can perform operations before and after the next delegate. A delegate can also decide to not pass a request to the next delegate, which is called short-circuiting the request pipeline. Short-circuiting is often desirable because it allows unnecessary work to be avoided. For example, the static file middleware can return a request for a static file and short-circuit the rest of the pipeline. Exception-handling delegates need to be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

```
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!");
        });
    }
}
```

The first app.Run delegate terminates the pipeline.

You can chain multiple request delegates together with app.Use. The `next` parameter represents the next delegate in the pipeline. (Remember that you can short-circuit the pipeline by *not* calling the *next* parameter.) You can typically perform actions both before and after the next delegate, as this example demonstrates:

```
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.");
        });
    }
}
```

▯ Warning

Do not call `next.Invoke` after the response has been sent to the client. Changes to `HttpResponse` after the response has started will throw an exception. For example, changes such as setting headers, status code, etc, will throw an exception. Writing to the response body after calling `next`:

May cause a protocol violation. For example, writing more than the stated `content-length`.
May corrupt the body format. For example, writing an HTML footer to a CSS file.

HttpResponse.HasStarted is a useful hint to indicate if headers have been sent and/or the body has been written to.

Ordering

The order that middleware components are added in the `Configure` method defines the order in which they are invoked on requests, and the reverse order for the response. This ordering is critical for security, performance, and functionality.

The Configure method (shown below) adds the following middleware components:

Exception/error handling
Static file server
Authentication

MVC

```
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.UseIdentity();                     // Authenticate before you access
                                           // secure resources.

    app.UseMvcWithDefaultRoute();          // Add MVC to the request pipeline.
}
```

In the code above, `UseExceptionHandler` is the first middleware component added to the pipeline—therefore, it catches any exceptions that occur in later calls.

The static file middleware is called early in the pipeline so it can handle requests and short-circuit without going through the remaining components. The static file middleware provides **no** authorization checks. Any files served by it, including those under *wwwroot*, are publicly available. See Working with static files for an approach to secure static files.

If the request is not handled by the static file middleware, it's passed on to the Identity middleware (`app.UseIdentity`), which performs authentication. Identity does not short-circuit unauthenticated requests. Although Identity authenticates requests, authorization (and rejection) occurs only after MVC selects a specific controller and action.

The following example demonstrates a middleware ordering where requests for static files are handled by the static file middleware before the response compression middleware. Static files are not compressed with this ordering of the middleware. The MVC responses from UseMvcWithDefaultRoute can be compressed.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();         // Static files not compressed
                                  // by middleware.
    app.UseResponseCompression();
    app.UseMvcWithDefaultRoute();
}
```

Run, Map, and Use

You configure the HTTP pipeline using `Run`, `Map`, and `Use`. The `Run` method short-circuits the pipeline (that is, it does not call a `next` request delegate). `Run` is a convention, and some middleware components may expose `Run[Middleware]` methods that run at the end of the pipeline.

`Map*` extensions are used as a convention for branching the pipeline. Map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

```
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>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segment(s) are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

MapWhen branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

```
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>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=master	Branch used = master

`Map` supports nesting, for example:

```
app.Map("/level1", level1App => {
       level1App.Map("/level2a", level2AApp => {
           // "/level1/level2a"
           //...
       });
       level1App.Map("/level2b", level2BApp => {
           // "/level1/level2b"
           //...
       });
   });
```

`Map` can also match multiple segments at once, for example:

```
app.Map("/level1/level2", HandleMultiSeg);
```

Built-in middleware

ASP.NET Core ships with the following middleware components:

MIDDLEWARE	DESCRIPTION
Authentication	Provides authentication support.
CORS	Configures Cross-Origin Resource Sharing.

MIDDLEWARE	DESCRIPTION
Response Caching	Provides support for caching responses.
Response Compression	Provides support for compressing responses.
Routing	Defines and constrains request routes.
Session	Provides support for managing user sessions.
Static Files	Provides support for serving static files and directory browsing.
URL Rewriting Middleware	Provides support for rewriting URLs and redirecting requests.

Writing middleware

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following middleware, which sets the culture for the current request from the query string:

```
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}");
        });

    }
}
```

Note: The sample code above is used to demonstrate creating a middleware component. See Globalization and localization for ASP.NET Core's built-in localization support.

You can test the middleware by passing in the culture, for example `http://localhost:7997/?culture=no`.

The following code moves the middleware delegate to a class:

```
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 Invoke(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);
        }
    }
}
```

The following extension method exposes the middleware through IApplicationBuilder:

```
using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}
```

The following code calls the middleware from `Configure`:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });

    }
}
```

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*. See *Per-request dependencies* below if you need to share services with middleware within a request.

Middleware components can resolve their dependencies from dependency injection through constructor parameters. `UseMiddleware<T>` can also accept additional parameters directly.

Per-request dependencies

Because middleware is constructed at app startup, not per-request, *scoped* lifetime services used by middleware constructors are not shared with other dependency-injected types during each request. If you must share a *scoped* service between your middleware and other types, add these services to the `Invoke` method's signature. The `Invoke` method can accept additional parameters that are populated by dependency injection. For example:

```
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);
    }
}
```

Resources

Sample code used in this doc
Migrating HTTP Modules to Middleware
Application Startup
Request Features

Introduction to working with static files in ASP.NET Core

By Rick Anderson

Static files, such as HTML, CSS, image, and JavaScript, are assets that an ASP.NET Core app can serve directly to clients.

View or download sample code

Serving static files

Static files are typically located in the `web root` (*<content-root>/wwwroot*) folder. See Content root and Web root for more information. You generally set the content root to be the current directory so that your project's `web root` will be found while in development.

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();

    host.Run();
}
```

Static files can be stored in any folder under the `web root` and accessed with a relative path to that root. For example, when you create a default Web application project using Visual Studio, there are several folders created within the *wwwroot* folder - *css*, *images*, and *js*. The URI to access an image in the *images* subfolder:

`http://<app>/images/<imageFileName>`
`http://localhost:9189/images/banner3.svg`

In order for static files to be served, you must configure the Middleware to add static files to the pipeline. The static file middleware can be configured by adding a dependency on the *Microsoft.AspNetCore.StaticFiles* package to your project and then calling the `UseStaticFiles` extension method from `Startup.Configure`:

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

`app.UseStaticFiles();` makes the files in `web root` (*wwwroot* by default) servable. Later I'll show how to make other directory contents servable with `UseStaticFiles`.

You must include the NuGet package "Microsoft.AspNetCore.StaticFiles".

Note

`web root` defaults to the *wwwroot* directory, but you can set the `web root` directory with `UseWebRoot`.

Suppose you have a project hierarchy where the static files you wish to serve are outside the `web root`. For example:

wwwroot
css
images
...

MyStaticFiles

test.png

For a request to access *test.png*, configure the static files middleware as follows:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles")
    });
}
```

A request to `http://<app>/StaticFiles/test.png` will serve the *test.png* file.

`StaticFileOptions()` can set response headers. For example, the code below sets up static file serving from the *wwwroot* folder and sets the `Cache-Control` header to make them publicly cacheable for 10 minutes (600 seconds):

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions()
    {
        OnPrepareResponse = ctx =>
        {
            ctx.Context.Response.Headers.Append("Cache-Control", "public,max-age=600");
        }
    });
}
```

Static file authorization

The static file module provides **no** authorization checks. Any files served by it, including those under *wwwroot* are publicly available. To serve files based on authorization:

Store them outside of *wwwroot* and any directory accessible to the static file middleware **and**

Serve them through a controller action, returning a `FileResult` where authorization is applied

Enabling directory browsing

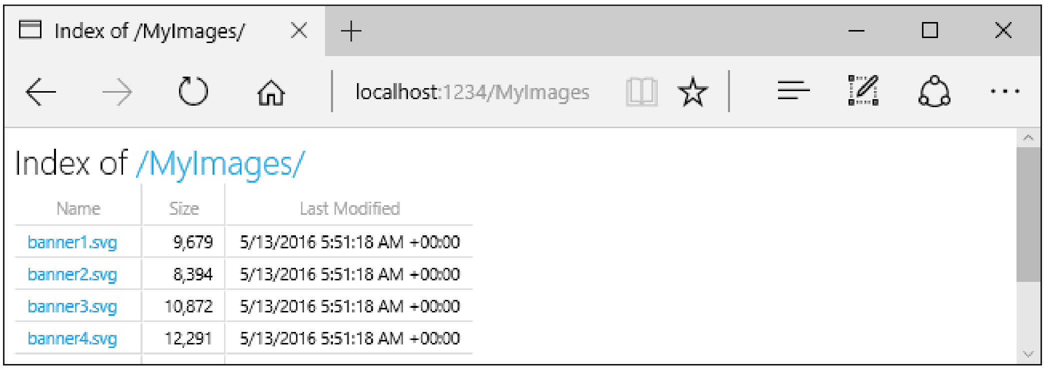
Directory browsing allows the user of your web app to see a list of directories and files within a specified directory. Directory browsing is disabled by default for security reasons (see Considerations). To enable directory browsing, call the `UseDirectoryBrowser` extension method from `Startup.Configure`:

```
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 = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

And add required services by calling `AddDirectoryBrowser` extension method from `Startup.ConfigureServices`:

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

The code above allows directory browsing of the *wwwroot/images* folder using the URL http://<app>/MyImages, with links to each file and folder:



See Considerations on the security risks when enabling browsing.

Note the two `app.UseStaticFiles` calls. The first one is required to serve the CSS, images and JavaScript in the *wwwroot* folder, and the second call for directory browsing of the *wwwroot/images* folder using the URL http://<app>/MyImages:

```
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 = new PathString("/MyImages")
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

Serving a default document

Setting a default home page gives site visitors a place to start when visiting your site. In order for your Web app to serve a default page without the user having to fully qualify the URI, call the `UseDefaultFiles` extension method from `Startup.Configure` as follows.

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

Note

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL re-writer that doesn't actually serve the file. You must enable the static file middleware (`UseStaticFiles`) to serve the file.

With `UseDefaultFiles`, requests to a folder will search for:

default.htm
default.html
index.htm
index.html

The first file found from the list will be served as if the request was the fully qualified URI (although the browser URL will continue to show the URI requested).

The following code shows how to change the default file name to *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` combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and `UseDirectoryBrowser`.

The following code enables static files and the default file to be served, but does not allow directory browsing:

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

The following code enables static files, default files and directory browsing:

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

See Considerations on the security risks when enabling browsing. As with `UseStaticFiles`, `UseDefaultFiles`, and `UseDirectoryBrowser`, if you wish to serve files that exist outside the `web root`, you instantiate and configure an `FileServerOptions` object that you pass as a parameter to `UseFileServer`. For example, given the following directory hierarchy in your Web app:

wwwroot

css

images

...

MyStaticFiles

test.png

default.html

Using the hierarchy example above, you might want to enable static files, default files, and browsing for the `MyStaticFiles` directory. In the following code snippet, that is accomplished with a single call to `FileServerOptions`.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseFileServer(new FileServerOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"MyStaticFiles")),
        RequestPath = new PathString("/StaticFiles"),
        EnableDirectoryBrowsing = true
    });
}
```

If `enableDirectoryBrowsing` is set to `true` you are required to call `AddDirectoryBrowser` extension method from `Startup.ConfigureServices`:

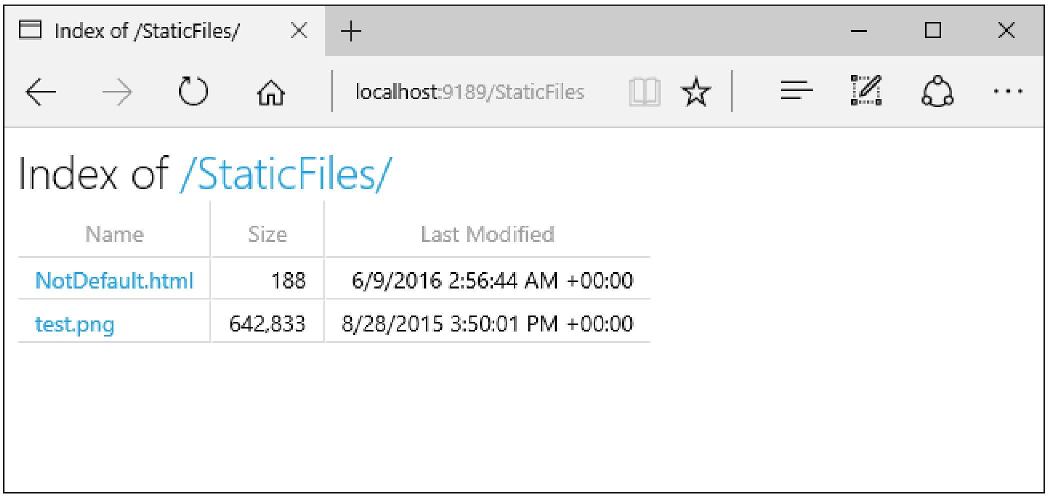
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

Using the file hierarchy and code above:

URI	RESPONSE
`http://<app>/StaticFiles/test.png`	MyStaticFiles/test.png

URI	RESPONSE
`http://<app>/StaticFiles`	MyStaticFiles/default.html

If no default named files are in the *MyStaticFiles* directory, http://<app>/StaticFiles returns the directory listing with clickable links:



Note

`UseDefaultFiles` and `UseDirectoryBrowser` will take the url http://<app>/StaticFiles without the trailing slash and cause a client side redirect to http://<app>/StaticFiles/ (adding the trailing slash). Without the trailing slash relative URLs within the documents would be incorrect.

FileExtensionContentTypeProvider

The `FileExtensionContentTypeProvider` class contains a collection that maps file extensions to MIME content types. In the following sample, several file extensions are registered to known MIME types, the ".rtf" is replaced, and ".mp4" is removed.

```
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 = new PathString("/MyImages"),
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions()
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), @"wwwroot", "images")),
        RequestPath = new PathString("/MyImages")
    });
}
```

See MIME content types.

Non-standard content types

The ASP.NET static file middleware understands almost 400 known file content types. If the user requests a file of an unknown file type, the static file middleware returns a HTTP 404 (Not found) response. If directory browsing is enabled, a link to the file will be displayed, but the URI will return an HTTP 404 error.

The following code enables serving unknown types and will render the unknown file as an image.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions()
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}
```

With the code above, a request for a file with an unknown content type will be returned as an image.

⚠ Warning

Enabling `ServeUnknownFileTypes` is a security risk and using it is discouraged. `FileExtensionContentTypeProvider` (explained above) provides a safer alternative to serving files with non-standard extensions.

Considerations

⚠ Warning

`UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. We recommend that you **not** enable directory browsing in production. Be careful about which directories you enable with `UseStaticFiles` or `UseDirectoryBrowser` as the entire directory and all sub-directories will be accessible. We recommend keeping public content in its own directory such as *<content root>/wwwroot*, away from application views, configuration files, etc.

The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of their underlying file system. For example, Windows is case insensitive, but Mac and Linux are not.

ASP.NET Core applications hosted in IIS use the ASP.NET Core Module to forward all requests to the application including requests for static files. The IIS static file handler is not used because it doesn't get a chance to handle requests before they are handled by the ASP.NET Core Module.

To remove the IIS static file handler (at the server or website level):

Navigate to the **Modules** feature

Select **StaticFileModule** in the list

Tap **Remove** in the **Actions** sidebar

⚠ Warning

If the IIS static file handler is enabled **and** the ASP.NET Core Module (ANCM) is not correctly configured (for example if *web.config* was not deployed), static files will be served.

Code files (including c# and Razor) should be placed outside of the app project's `web root` (*wwwroot* by default). This creates a clean separation between your app's client side content and server side source code, which prevents server side code from being leaked.

Additional Resources

Middleware

Introduction to ASP.NET Core

Routing in ASP.NET Core

By Ryan Nowak, Steve Smith, and Rick Anderson

Routing functionality is responsible for mapping an incoming request to a route handler. Routes are defined in the ASP.NET app and configured when the app starts up. A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing. Using route information from the ASP.NET app, the routing functionality is also able to generate URLs that map to route handlers. Therefore, routing can find a route handler based on a URL, or the URL corresponding to a given route handler based on route handler information.

Important

This document covers the low level ASP.NET Core routing. For ASP.NET Core MVC routing, see Routing to Controller Actions

View or download sample code

Routing basics

Routing uses *routes* (implementations of IRouter) to:

map incoming requests to *route handlers*

generate URLs used in responses

Generally, an app has a single collection of routes. When a request arrives, the route collection is processed in order. The incoming request looks for a route that matches the request URL by calling the `RouteAsync` method on each available route in the route collection. By contrast, a response can use routing to generate URLs (for example, for redirection or links) based on route information, and thus avoid having to hard-code URLs, which helps maintainability.

Routing is connected to the middleware pipeline by the `RouterMiddleware` class. ASP.NET MVC adds routing to the middleware pipeline as part of its configuration. To learn about using routing as a standalone component, see using-routing-middleware.

URL matching

URL matching is the process by which routing dispatches an incoming request to a *handler*. This process is generally based on data in the URL path, but can be extended to consider any data in the request. The ability to dispatch requests to separate handlers is key to scaling the size and complexity of an application.

Incoming requests enter the `RouterMiddleware`, which calls the `RouteAsync` method on each route in sequence. The `IRouter` instance chooses whether to *handle* the request by setting the `RouteContext` `Handler` to a non-null `RequestDelegate`. If a route sets a handler for the request, route processing stops and the handler will be invoked to process the request. If all routes are tried and no handler is found for the request, the middleware calls *next* and the next middleware in the request pipeline is invoked.

The primary input to `RouteAsync` is the `RouteContext` `HttpContext` associated with the current request. The `RouteContext.Handler` and `RouteContext` `RouteData` are outputs that will be set after a route matches.

A match during `RouteAsync` will also set the properties of the `RouteContext.RouteData` to appropriate values based on the request processing done so far. If a route matches a request, the `RouteContext.RouteData` will contain important state information about the *result*.

`RouteData` `Values` is a dictionary of *route values* produced from the route. These values are usually determined by tokenizing the URL, and can be used to accept user input, or to make further dispatching decisions inside the application.

`RouteData` `DataTokens` is a property bag of additional data related to the matched route. `DataTokens` are provided to support associating state data with each route so the application can make decisions later based on which route matched. These values are developer-defined and do **not** affect the behavior of routing in any way. Additionally, values stashed in data tokens can be of any type, in contrast to route values, which must be easily convertible to and from strings.

`RouteData` `Routers` is a list of the routes that took part in successfully matching the request. Routes can be nested inside one another, and the `Routers` property reflects the path through the logical tree of routes that resulted in a match. Generally the first item in `Routers` is the route collection, and should be used for URL generation. The last item in `Routers` is the route handler that matched.

URL generation

URL generation is the process by which routing can create a URL path based on a set of route values. This allows for a logical separation between your handlers and the URLs that access them.

URL generation follows a similar iterative process, but starts with user or framework code calling into the `GetVirtualPath` method of the route collection. Each *route* will then have its `GetVirtualPath` method called in sequence until a non-null `VirtualPathData` is returned.

The primary inputs to `GetVirtualPath` are:

`VirtualPathContext` `HttpContext`

`VirtualPathContext` `Values`

`VirtualPathContext` `AmbientValues`

Routes primarily use the route values provided by the `Values` and `AmbientValues` to decide where it is possible to generate a URL and what values to include. The `AmbientValues` are the set of route values that were produced from matching the current request with the routing system. In contrast, `Values` are the route values that specify how to generate the desired URL for the current operation. The `HttpContext` is provided in case a route needs to get services or additional data associated with the current context.

Tip: Think of `Values` as being a set of overrides for the `AmbientValues`. URL generation tries to reuse route values from the current request to make it easy to generate URLs for links using the same route or route values.

The output of `GetVirtualPath` is a `VirtualPathData`. `VirtualPathData` is a parallel of `RouteData`; it contains the `VirtualPath` for the output URL as well as the some additional properties that should be set by the route.

The `VirtualPathData` `VirtualPath` property contains the *virtual path* produced by the route. Depending on your needs you may need to process the path further. For instance, if you want to render the generated URL in HTML you need to prepend the base path of the application.

The `VirtualPathData` `Router` is a reference to the route that successfully generated the URL.

The `VirtualPathData` `DataTokens` properties is a dictionary of additional data related to the route that generated the URL. This is the parallel of `RouteData.DataTokens`.

Creating routes

Routing provides the `Route` class as the standard implementation of `IRouter`. `Route` uses the *route template* syntax to define patterns that will match against the URL path when `RouteAsync` is called. `Route` will use the same route template to generate a URL when `GetVirtualPath` is called.

Most applications will create routes by calling `MapRoute` or one of the similar extension methods defined on `IRouteBuilder`. All of these methods will create an instance of `Route` and add it to the route collection.

Note: `MapRoute` doesn't take a route handler parameter - it only adds routes that will be handled by the `DefaultHandler`. Since the default handler is an `IRouter`, it may decide not to handle the request. For example, ASP.NET MVC is typically configured as a default handler that only handles requests that match an available controller and action. To learn more about routing to MVC, see Routing to Controller Actions.

This is an example of a `MapRoute` call used by a typical ASP.NET MVC route definition:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template will match a URL path like `/Products/Details/17` and extract the route values `{ controller = Products, action = Details, id = 17 }`. The route values are determined by splitting the URL path into segments, and matching each segment with the *route parameter* name in the route template. Route parameters are named. They are defined by enclosing the parameter name in braces `{ }`.

The template above could also match the URL path `/` and would produce the values `{ controller = Home, action = Index }`. This happens because the `{controller}` and `{action}` route parameters have default values, and the `id` route parameter is optional. An equals `=` sign followed by a value after the route parameter name defines a default value for the parameter. A question mark `?` after the route parameter name defines the parameter as optional. Route parameters with a default value *always* produce a route value when the route matches - optional parameters will not produce a route value if there was no corresponding URL path segment.

See route-template-reference for a thorough description of route template features and syntax.

This example includes a *route constraint*:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id:int}");
```

This template will match a URL path like `/Products/Details/17`, but not `/Products/Details/Apples`. The route parameter definition `{id:int}` defines a *route constraint* for the `id` route parameter. Route constraints implement `IRouteConstraint` and inspect route values to verify them. In this example the route value `id` must be convertible to an integer. See route-constraint-reference for a more detailed explanation of route constraints that are provided by the framework.

Additional overloads of `MapRoute` accept values for `constraints`, `dataTokens`, and `defaults`. These additional parameters of `MapRoute` are defined as type `object`. The typical usage of these parameters is to pass an anonymously typed object, where the property names of the anonymous type match route parameter names.

The following two examples create equivalent routes:

```
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?}");
```

Tip: The inline syntax for defining constraints and defaults can be more convenient for simple routes. However, there are features such as data tokens which are not supported by inline syntax.

This example demonstrates a few more features:

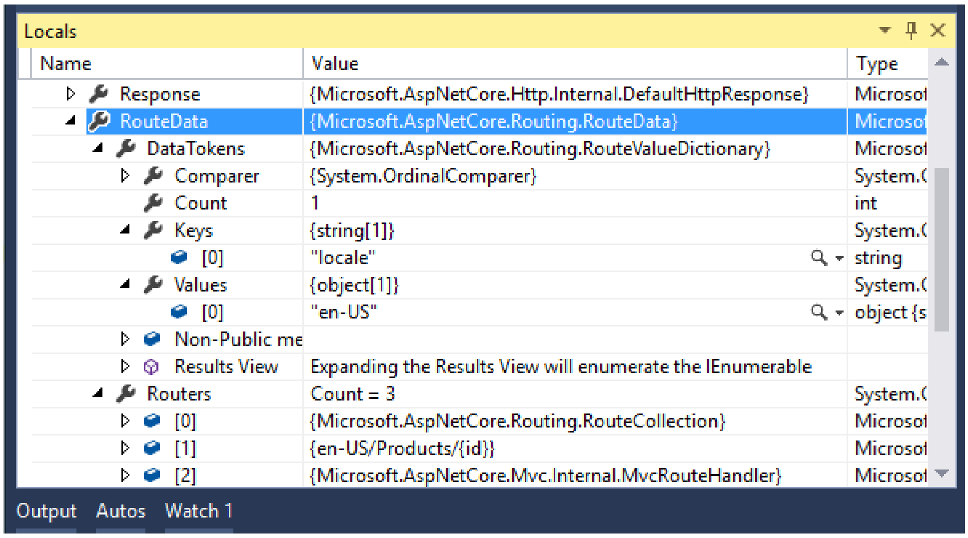
```
routes.MapRoute(
  name: "blog",
  template: "Blog/{*article}",
  defaults: new { controller = "Blog", action = "ReadArticle" });
```

This template will match a URL path like `/Blog/All-About-Routing/Introduction` and will extract the values `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`. The default route values for `controller` and `action` are produced by the route even though there are no corresponding route parameters in the template. Default values can be specified in the route template. The `article` route parameter is defined as a *catch-all* by the appearance of an asterix `*` before the route parameter name. Catch-all route parameters capture the remainder of the URL path, and can also match the empty string.

This example adds route constraints and data tokens:

```
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" });
```

This template will match a URL path like `/en-US/Products/5` and will extract the values `{ controller = Products, action = Details, id = 5 }` and the data tokens `{ locale = en-US }`.



URL generation

The `Route` class can also perform URL generation by combining a set of route values with its route template. This is logically the reverse process of matching the URL path.

Tip: To better understand URL generation, imagine what URL you want to generate and then think about how a route template would match that URL. What values would be produced? This is the rough equivalent of how URL generation works in the `Route` class.

This example uses a basic ASP.NET MVC style route:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

With the route values `{ controller = Products, action = List }`, this route will generate the URL `/Products/List`. The route values are substituted for the corresponding route parameters to form the URL path. Since `id` is an optional route parameter, it's no problem that it doesn't have a value.

With the route values `{ controller = Home, action = Index }`, this route will generate the URL `/`. The route values that were provided match the default values so the segments corresponding to those values can be safely omitted. Note that both URLs generated would round-trip with this route definition and produce the same route values that were used to generate the URL.

Tip: An app using ASP.NET MVC should use `UrlHelper` to generate URLs instead of calling into routing directly.

For more details about the URL generation process, see url-generation-reference.

Using Routing Middleware

Add the NuGet package "Microsoft.AspNetCore.Routing".

Add routing to the service container in *Startup.cs*:

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

Routes must configured in the `Configure` method in the `Startup` class. The sample below uses these APIs:

`RouteBuilder`
`Build`
`MapGet` Matches only HTTP GET requests
`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);
}
```

The table below shows the responses with the given URIs.

URI	RESPONSE
/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/	<Fall through, no match>
GET /hello/Joe	Hi, Joe!

URI	RESPONSE
POST /hello/Joe	<Fall through, matches HTTP GET only>
GET /hello/Joe/Smith	<Fall through, no match>

If you are configuring a single route, call `app.UseRouter` passing in an `IRouter` instance. You won't need to call `RouteBuilder`.

The framework provides a set of extension methods for creating routes such as:

`MapRoute`
`MapGet`
`MapPost`
`MapPut`
`MapDelete`
`MapVerb`

Some of these methods such as `MapGet` require a `RequestDelegate` to be provided. The `RequestDelegate` will be used as the *route handler* when the route matches. Other methods in this family allow configuring a middleware pipeline which will be used as the route handler. If the *Map* method doesn't accept a handler, such as `MapRoute`, then it will use the `DefaultHandler`.

The `Map[Verb]` methods use constraints to limit the route to the HTTP Verb in the method name. For example, see MapGet and MapVerb.

Route Template Reference

Tokens within curly braces (`{ }`) define *route parameters* which will be bound if the route is matched. You can define more than one route parameter in a route segment, but they must be separated by a literal value. For example `{controller=Home}{action=Index}` would not be a valid route, since there is no literal value between `{controller}` and `{action}`. These route parameters must have a name, and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URLs path. To match the literal route parameter delimiter `{` or `}`, escape it by repeating the character (`{{` or `}}`).

URL patterns that attempt to capture a filename with an optional file extension have additional considerations. For example, using the template `files/{filename}.{ext?}` - When both `filename` and `ext` exist, both values will be populated. If only `filename` exists in the URL, the route matches because the trailing period `.` is optional. The following URLs would match this route:

`/files/myFile.txt`
`/files/myFile.`
`/files/myFile`

You can use the `*` character as a prefix to a route parameter to bind to the rest of the URI - this is called a *catch-all* parameter. For example, `blog/{*slug}` would match any URI that started with `/blog` and had any value following it (which would be assigned to the `slug` route value). Catch-all parameters can also match the empty string.

Route parameters may have *default values*, designated by specifying the default after the parameter name, separated by an `=`. For example, `{controller=Home}` would define `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. In addition to default values, route parameters may be optional (specified by appending a `?` to the end of the parameter name, as in `id?`). The difference between optional and "has default" is that a route parameter with a default value always produces a value; an optional parameter has a value only when one is provided.

Route parameters may also have constraints, which must match the route value bound from the URL. Adding a colon `:` and constraint name after the route parameter name specifies an *inline constraint* on a route parameter. If the constraint requires arguments those are provided enclosed in parentheses `( )` after the constraint name. Multiple inline constraints can be specified by appending another colon `:` and constraint name. The constraint name is passed to the `IInlineConstraintResolver` service to create an instance of `IRouteConstraint` to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies the `minlength` constraint with the argument `10`. For more description route constraints, and a listing of the constraints provided by the framework, see route-constraint-reference.

The following table demonstrates some route templates and their behavior.

ROUTE TEMPLATE	EXAMPLE MATCHING URL	NOTES
hello	/hello	Only matches the single path `/hello`
{Page=Home}	/	Matches and sets `Page` to `Home`
{Page=Home}	/Contact	Matches and sets `Page` to `Contact`
{controller}/{action}/{id?}	/Products/List	Maps to `Products` controller and `List` action
{controller}/{action}/{id?}	/Products/Details/123	Maps to `Products` controller and `Details` action. `id` set to 123
{controller=Home}/{action=Index}/{id?}	/	Maps to `Home` controller and `Index` method; `id` is ignored.

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

Tip: Enable Logging to see how the built in routing implementations, such as `Route`, match requests.

Route Constraint Reference

Route constraints execute when a `Route` has matched the syntax of the incoming URL and tokenized the URL path into route values. Route constraints generally inspect the route value associated via the route template and make a simple yes/no decision about whether or not the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the `HttpMethodRouteConstraint` can accept or reject a request based on its HTTP verb.

 Warning

Avoid using constraints for **input validation**, because doing so means that invalid input will result in a 404 (Not Found) instead of a 400 with an appropriate error message. Route constraints should be used to **disambiguate** between similar routes, not to validate the inputs for a particular route.

The following table demonstrates some route constraints and their expected behavior.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any integer
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` or `false` (case-insensitive)

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime` value (in the invariant culture - see warning)
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal` value (in the invariant culture - see warning)
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid `double` value (in the invariant culture - see warning)
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid `float` value (in the invariant culture - see warning)
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	Matches a valid `Guid` value
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid `long` value
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be at least 4 characters
`maxlength(value)`	`{filename:maxlength(8)}`	`Richard`	String must be no more than 8 characters
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be exactly 12 characters long
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be at least 8 and no more than 16 characters long
`min(value)`	`{age:min(18)}`	`19`	Integer value must be at least 18
`max(value)`	`{age:max(120)}`	`91`	Integer value must be no more than 120
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value must be at least 18 but no more than 120
`alpha`	`{name:alpha}`	`Rick`	String must consist of one or more alphabetical characters (`a`-`z`, case-insensitive)
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must match the regular expression (see tips about defining a regular expression)

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`required`	`{name:required}`	`Rick`	Used to enforce that a non-parameter value is present during URL generation

▯ Warning

Route constraints that verify the URL can be converted to a CLR type (such as `int` or `DateTime`) always use the invariant culture - they assume the URL is non-localizable. The framework-provided route constraints do not modify the values stored in route values. All route values parsed from the URL will be stored as strings. For example, the Float route constraint will attempt to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions

The ASP.NET Core framework adds `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant` to the regular expression constructor. See RegexOptions Enumeration for a description of these members.

Regular expressions use delimiters and tokens similar to those used by Routing and the C# language. Regular expression tokens must be escaped. For example, to use the regular expression `^\d{3}-\d{2}-\d{4}$` in Routing, it needs to have the `\` characters typed in as `\\` in the C# source file to escape the `\` string escape character (unless using verbatim string literals). The `{` and `}` characters need to be escaped by doubling them to escape the Routing parameter delimiter characters. The table below shows a regular expression and the escaped version.

EXPRESSION	NOTE
`^\d{3}-\d{2}-\d{4}$`	Regular expression
`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`	Escaped

Regular expressions used in routing will often start with the `^` character (match starting position of the string) and end with the `$` character (match ending position of the string). The `^` and `$` characters ensure that the regular expression match the entire route parameter value. Without the `^` and `$` characters the regular expression will match any sub-string within the string, which is often not what you want. The table below shows some examples and explains why they match or fail to match.

EXPRESSION	STRING	MATCH	COMMENT
`[a-z]{2}`	hello	yes	substring matches
`[a-z]{2}`	123abc456	yes	substring matches
`[a-z]{2}`	mz	yes	matches expression
`[a-z]{2}`	MZ	yes	not case sensitive
`^[a-z]{2}$`	hello	no	see `^` and `$` above
`^[a-z]{2}$`	123abc456	no	see `^` and `$` above

Refer to .NET Framework Regular Expressions for more information on regular expression syntax.

To constrain a parameter to a known set of possible values, use a regular expression. For example `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string "^(list|get|create)$" would be equivalent. Constraints that are passed in the constraints dictionary (not inline within a template) that don't match one of the known constraints are also treated as regular expressions.

URL Generation Reference

The example below shows how to generate a link to a route given a dictionary of route values and a `RouteCollection`.

```
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/>");
});
```

The `VirtualPath` generated at the end of the sample above is `/package/create/123`.

The second parameter to the `VirtualPathContext` constructor is a collection of *ambient values*. Ambient values provide convenience by limiting the number of values a developer must specify within a certain request context. The current route values of the current request are considered ambient values for link generation. For example, in an ASP.NET MVC app if you are in the `About` action of the `HomeController`, you don't need to specify the controller route value to link to the `Index` action (the ambient value of `Home` will be used).

Ambient values that don't match a parameter are ignored, and ambient values are also ignored when an explicitly-provided value overrides it, going from left to right in the URL.

Values that are explicitly provided but which don't match anything are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller="Home"	action="About"	`/Home/About`
controller="Home"	controller="Order",action="About"	`/Order/About`
controller="Home",color="Red"	action="About"	`/Home/About`
controller="Home"	action="About",color="Red"	`/Home/About?color=Red`

If a route has a default value that doesn't correspond to a parameter and that value is explicitly provided, it must match the default value. For example:

```
routes.MapRoute("blog_route", "blog/{*slug}",
  defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation would only generate a link for this route when the matching values for controller and action are provided.

URL Rewriting Middleware in ASP.NET Core

By Luke Latham and Mikael Mengistu

View or download sample code

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses are not tightly linked. There are several scenarios where URL rewriting is valuable:

Moving or replacing server resources temporarily or permanently while maintaining stable locators for those resources
Splitting request processing across different applications or across areas of one application
Removing, adding, or reorganizing URL segments on incoming requests
Optimizing public URLs for Search Engine Optimization (SEO)
Permitting the use of friendly public URLs to help people predict the content they will find by following a link
Redirecting insecure requests to secure endpoints
Preventing image hotlinking

You can define rules for changing the URL in several ways, including regular expression (regex) matching rules, rules based on the Apache mod_rewrite module, rules based on the IIS Rewrite Module, and with your own method and class rule logic. This document introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core applications.

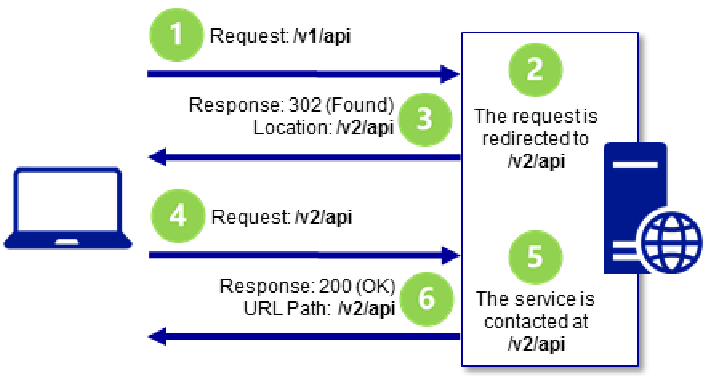
Note

URL rewriting can reduce the performance of an application. Where feasible, you should limit the number and complexity of rules.

URL redirect and URL rewrite

The difference in wording between *URL redirect* and *URL rewrite* may seem subtle at first but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting Middleware is capable of meeting the need for both.

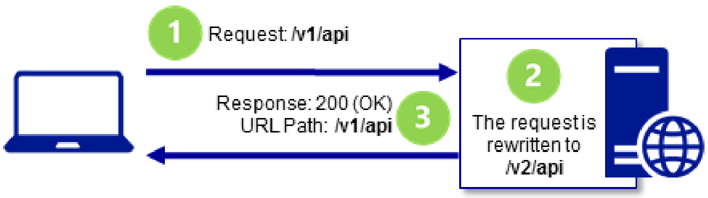
A *URL redirect* is a client-side operation, where the client is instructed to access a resource at another address. This requires a round-trip to the server, and the redirect URL returned to the client will appear in the browser's address bar when the client makes a new request for the resource. If `/resource` is *redirected* to `/different-resource`, the client will request `/resource`, and the server will respond that the client should obtain the resource at `/different-resource` with a status code indicating that the redirect is either temporary or permanent. The client will execute a new request for the resource at the redirect URL.



When redirecting requests to a different URL, you will indicate whether the redirect is permanent or temporary. The 301 (Moved Permanently) status code is used where the resource has a new, permanent URL and you wish to instruct the client that all future requests for the resource should use the new URL. The client will cache the response when a 301 status code is received. The 302 (Found) status code is used where the redirection is temporary or generally subject to change, such that the client should not store and reuse the redirect URL in the future. For more information, see RFC 2616: Status Code Definitions.

A *URL rewrite* is a server-side operation to provide a resource from a different resource address. Rewriting a URL doesn't require a round-trip to the server. The rewritten URL is not returned to the client and won't appear in a browser's address bar. When

`/resource` is *rewritten* to `/different-resource`, the client will request `/resource`, and the server will *internally* fetch the resource at `/different-resource`. Although the client might be able to retrieve the resource at the rewritten URL, the client won't be informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample application

You can explore the features of the URL Rewriting Middleware with the URL rewriting sample application. The application applies rewrite and redirect rules and shows the resultant rewritten or redirected URL.

When to use URL Rewriting Middleware

Use URL Rewriting Middleware when you are unable to use the URL Rewrite module in IIS on Windows Server, the Apache mod_rewrite module on Apache Server, URL rewriting on Nginx, or your application is hosted on WebListener server. The main reasons to use the server-based URL rewriting technologies in IIS, Apache, or Nginx are that the middleware doesn't support the full features of these modules and the performance of the middleware probably won't match that of the modules. However, there are some features of the server modules that don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, you can use the middleware instead.

Package

To include the middleware in your project, add a reference to the `Microsoft.AspNetCore.Rewrite` package. The middleware depends on .NET Framework 4.5.1 or .NET Standard 1.3 or later. This feature is available for apps that target ASP.NET Core 1.1.0 or later.

Extension and options

Establish your URL rewrite and redirect rules by creating an instance of the `RewriteOptions` class with extension methods for each of your rules. Chain multiple rules in the order that you would like them processed. The `RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with `app.UseRewriter(options);`.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

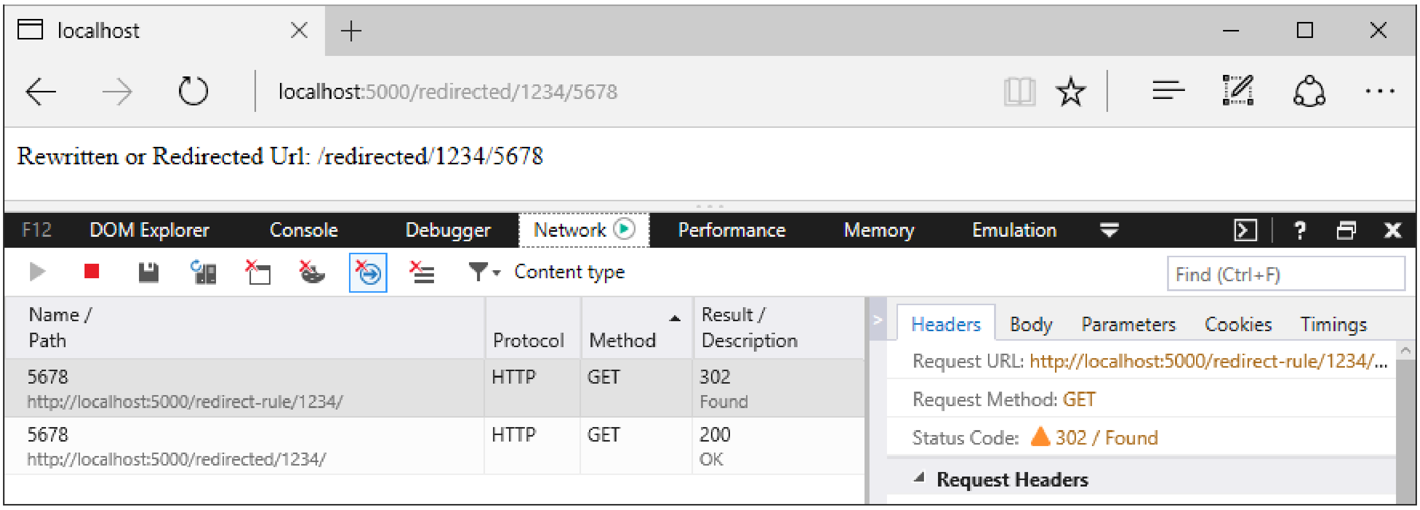
URL redirect

Use `AddRedirect()` to redirect requests. The first parameter will contain your regex for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If you don't specify the status code, it defaults to 302 (Found), which indicates that the resource has been temporarily moved or replaced.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

In a browser with developer tools enabled, make a request to the sample application with the path `/redirect-rule/1234/5678`. The regex matches the request path on `redirect-rule/(.*)`, and the path is replaced with `/redirected/1234/5678`. The redirect URL is sent back to the client with a 302 (Found) status code. The browser makes a new request at the redirect URL, which will appear in the browser's address bar. Since no rules in the sample application match on the redirect URL, the second request receives a 200 (OK) response from the application and the body of the response shows the redirect URL. A complete roundtrip is made to the server when a URL is *redirected*.

Original Request: `/redirect-rule/1234/5678`



The part of the expression contained by parentheses is called a *capture group*. The dot (`.`) of the expression means *match any character*, and the asterisk (`*`) signifies to *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678`, are captured by capture group `(.*)`. Any value you provide in the request URL after `redirect-rule/` will be captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1`, the second with `$2`, and they continue in sequence for the capture groups in your regex. There is only one captured group in the redirect rule regex in the sample application, so there is only one injected group in the replacement string, which is `$1`. When the rule is applied, the URL becomes `/redirected/1234/5678`.

URL redirect to a secure endpoint

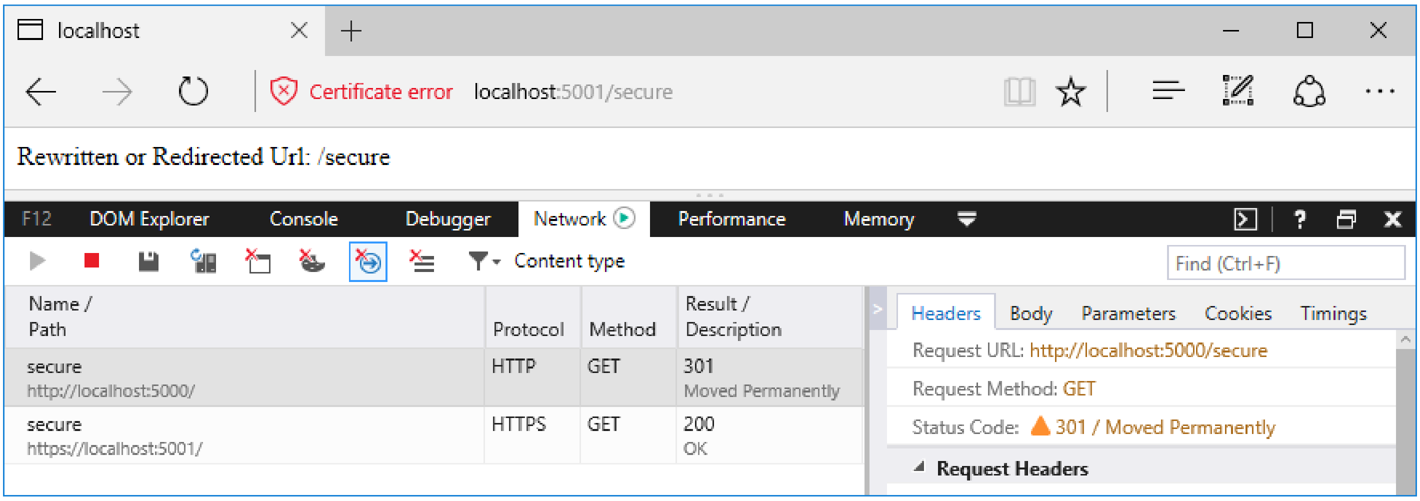
Use `AddRedirectToHttps()` to redirect insecure requests to the same host and path with secure HTTPS protocol (`https://`) with the flexibility to choose the status code and port. If the status code is not supplied, the middleware will default to 302 (Found). If the port is not supplied, the middleware will default to `null`, which means the protocol will change to `https://` and the client will access the resource on port 443. The example shows how to set the status code to 301 (Moved Permanently) and change the port to 5001.

```
var options = new RewriteOptions()
    .AddRedirectToHttps(301, 5001);

app.UseRewriter(options);
```

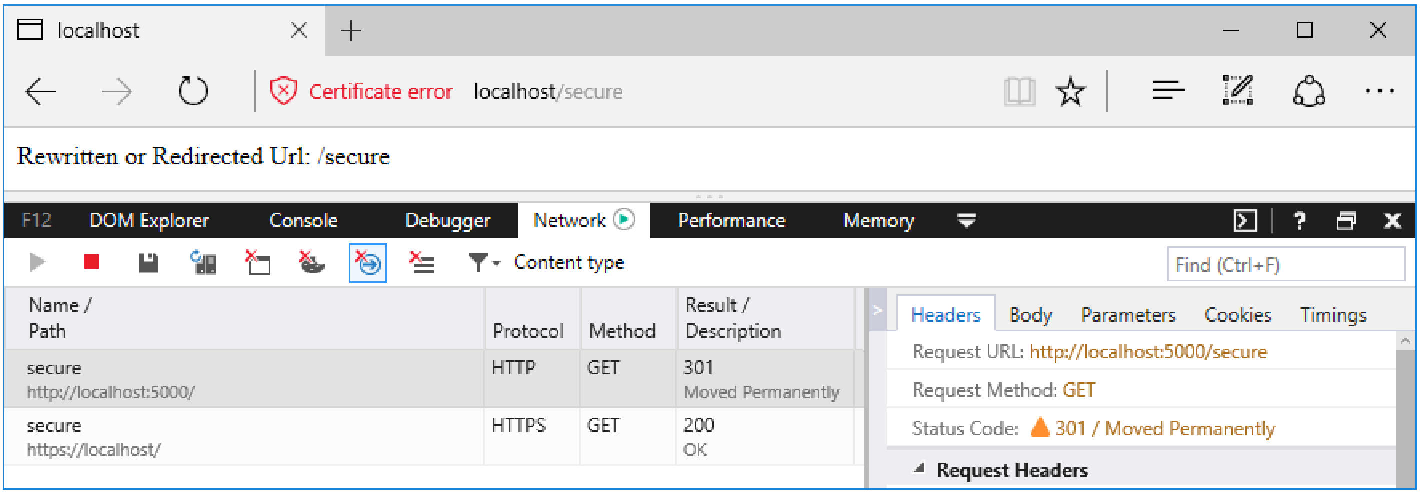
Use `AddRedirectToHttpsPermanent()` to redirect insecure requests to the same host and path with secure HTTPS protocol (`https://` on port 443). The middleware will set the status code to 301 (Moved Permanently).

The sample application is capable of demonstrating how to use `AddRedirectToHttps()` or `AddRedirectToHttpsPermanent()`. Add the extension method to the `RewriteOptions()`. Make an insecure request to the application at any URL. In order to see the response in a browser, you will probably need to dismiss a browser security warning that the self-signed certificate is untrusted.

Original Request using `AddRedirectToHttps(301, 5001)`: `/secure`



Original Request using `AddRedirectToHttpsPermanent()`: `/secure`

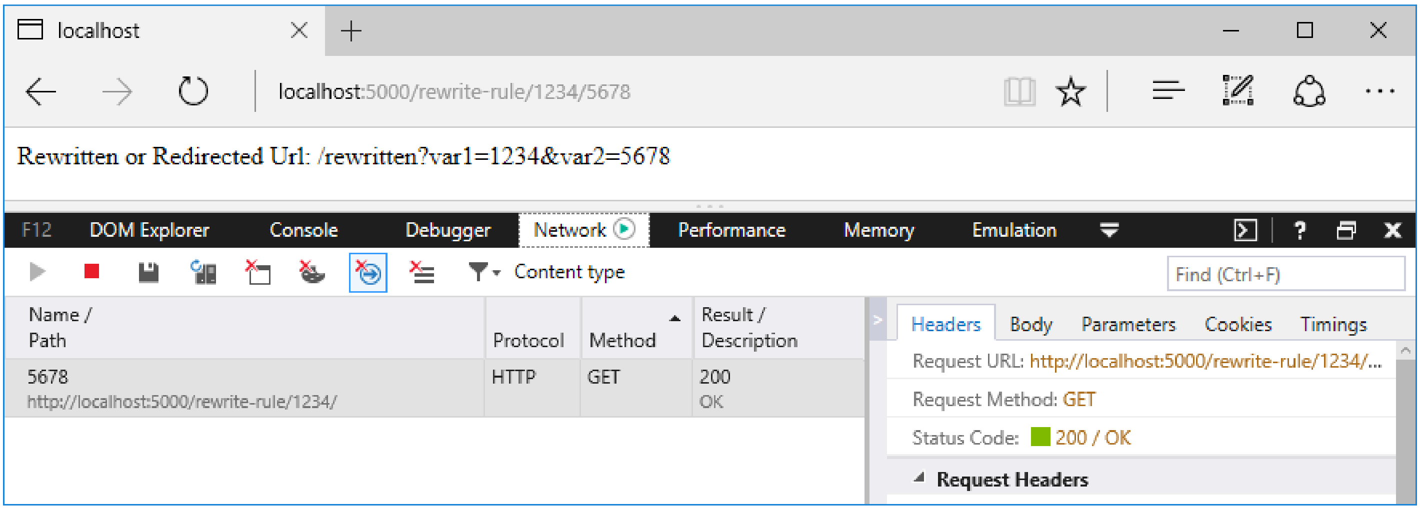


URL rewrite

Use `AddRewrite()` to create a rules for rewriting URLs. The first parameter will contain your regex for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, will indicate to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

Original Request: `/rewrite-rule/1234/5678`



The first thing you will notice in the regex is the carat (`^`) at the beginning of the expression. This means that matching should be attempted starting at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there is no carat at the start of the regex; therefore, any characters may precede `redirect-rule/` in the path for a successful match.

PATH	MATCH
`/redirect-rule/1234/5678`	Yes
`/my-cool-redirect-rule/1234/5678`	Yes
`/anotherredirect-rule/1234/5678`	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, will only match paths if they start with `rewrite-rule/`. Notice the difference in matching between the rewrite rule below and the redirect rule above.

PATH	MATCH
`/rewrite-rule/1234/5678`	Yes
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the resultant rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the querystring. The requested path of `/rewrite-rule/1234/5678` is rewritten to obtain the resource at `/rewritten?var1=1234&var2=5678`. If a querystring is present on the original request, it's preserved when the URL is rewritten.

There is no roundtrip to the server to obtain the resource. If the resource exists, it's fetched and returned to the client with a 200 (OK) status code. Because the client isn't redirected, the URL in the browser address bar doesn't change. As far as the client is concerned, the URL rewrite operation never occurred.

Note

`skipRemainingRules: true` should be used whenever possible, because matching rules is an expensive process and slows down application response time. For the fastest application response, order your rewrite rules from most frequently matched to least frequently matched and skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

Apache mod_rewrite

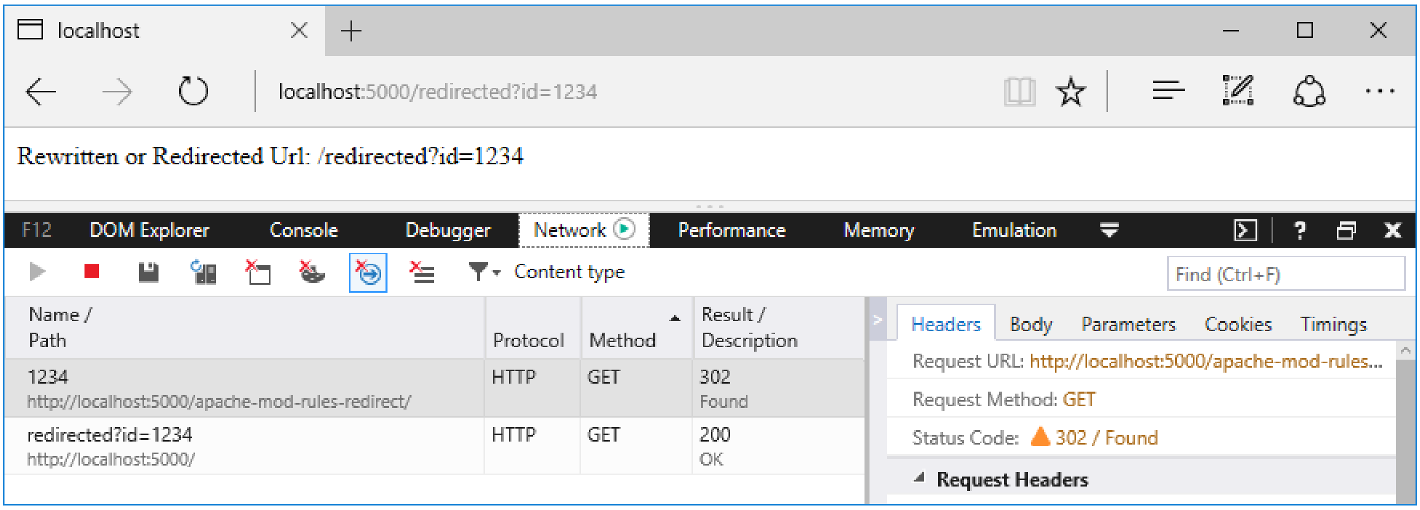
You can apply Apache mod_rewrite rules with `AddApacheModRewrite()`. The first parameter takes an `IFileProvider`, which is provided in the sample application via Dependency Injection by injecting the `IHostingEnvironment` and using it to provide the `ContentRootFileProvider`. The second parameter is the path to your rules file, which is *ApacheModRewrite.txt* in the sample application. You must make sure that the rules file is deployed with the application. For more information and examples of mod_rewrite rules, see Apache mod_rewrite.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

The sample application will redirect requests from `/apache-mod-rules-redirect/(.\*)` to `/redirected?id=$1`. The response status code is 302 (Found).

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Original Request: `/apache-mod-rules-redirect/1234`



Supported server variables

The middleware supports the following Apache mod_rewrite server variables:

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 Rewrite Module rules

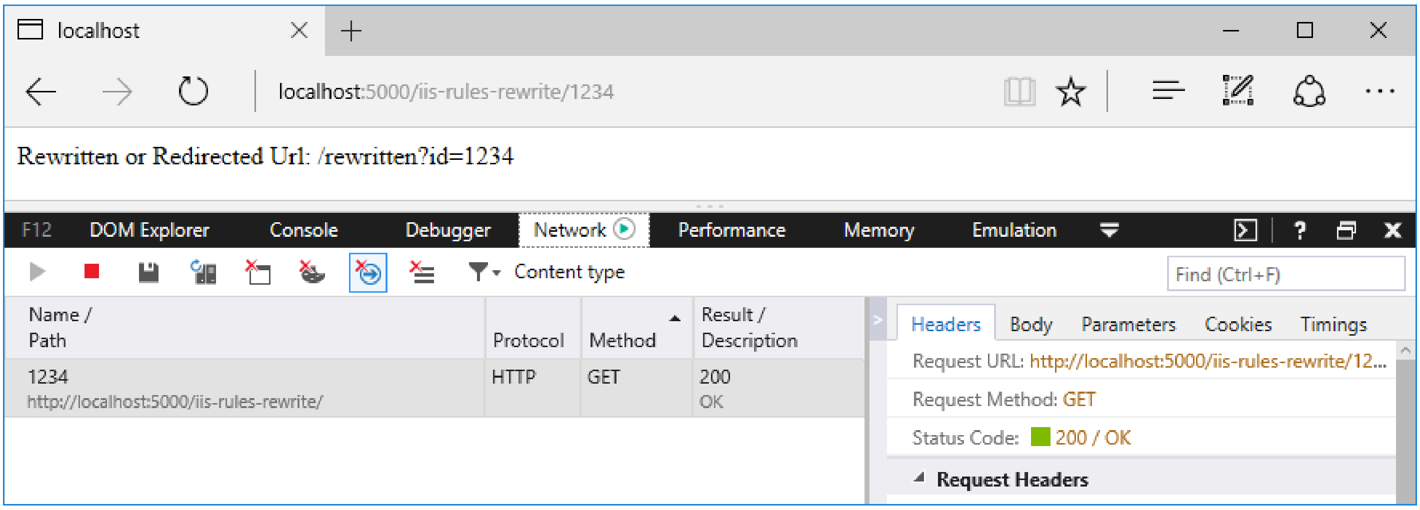
To use rules that would normally apply to the IIS URL Rewrite Module, use `AddIISUrlRewrite()`. The first parameter takes an `IFileProvider`, while the second parameter is the path to your XML rules file, which is *IISUrlRewrite.xml* in the sample application. You must make sure that the rules file is deployed with the application. Don't point this at your *web.config* file, as these rules should be stored outside of your *web.config* to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see Using Url Rewrite Module 2.0 and URL Rewrite Module Configuration Reference.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

The sample application will rewrite requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a 200 (OK) status code.

```
<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>
```

Original Request: `/iis-rules-rewrite/1234`



If you have an active IIS Rewrite Module with server-level rules configured that would impact your application in undesirable ways, you can disable the IIS Rewrite Module for an application. For more information, see Disabling IIS modules.

Unsupported features

The middleware does not support the following IIS URL Rewrite Module features:

Global rules (Basic Middleware #169)
Rewrite Maps (Basic Middleware #168)
CustomResponse action (Basic Middleware #135)
Custom Server Variables (Basic Middleware #183)
trackAllCaptures (Basic Middleware #178)
Wildcards
LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

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

Note

You can also obtain an `IFileProvider` via a `PhysicalFileProvider`. This approach may provide greater flexibility for the location of your rewrite rules files. Make sure that your rewrite rules files are deployed to the server at the path you provide.

```
PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
```

Method-based rule

Use `Add(Action<RewriteContext> applyRule)` to implement your own rule logic in a method. The `RewriteContext` exposes the `HttpContext` for use in your method. The `context.Result` determines how additional pipeline processing is handled.

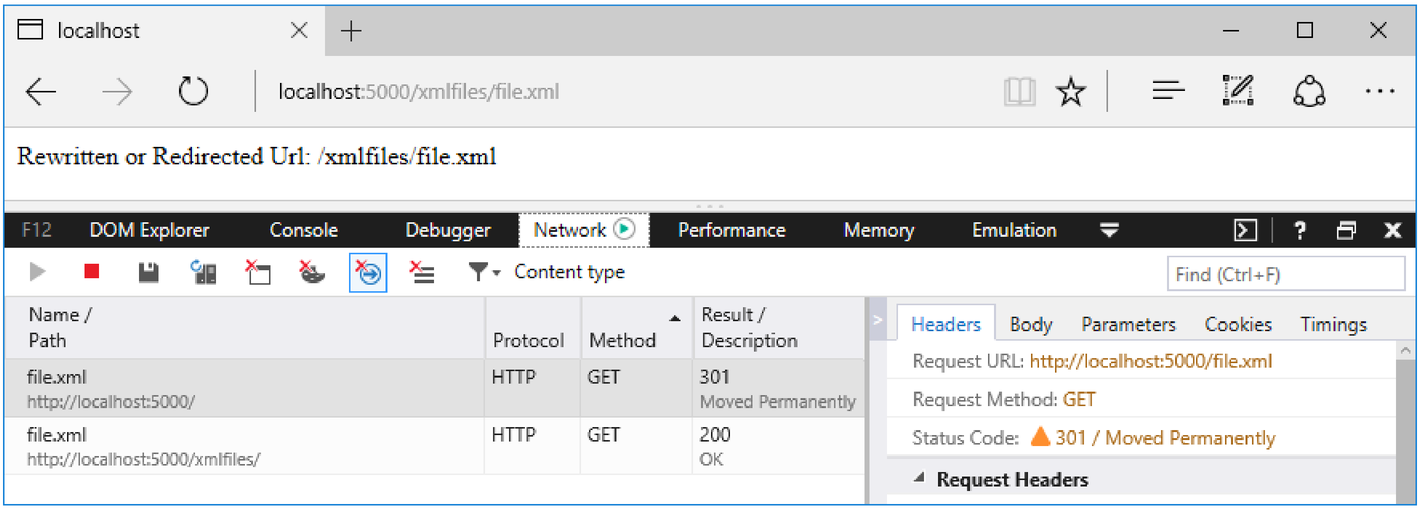
`CONTEXT.RESULT`	ACTION
`RuleResult.ContinueRules` (default)	Continue applying rules
`RuleResult.EndResponse`	Stop applying rules and send the response
`RuleResult.SkipRemainingRules`	Stop applying rules and send the context to the next middleware

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

The sample application demonstrates a method that redirects requests for paths that end with `.xml`. If you make a request for `/file.xml`, it's redirected to `/xmlfiles/file.xml`. The status code is set to 301 (Moved Permanently). For a redirect, you must explicitly set the status code of the response; otherwise, a 200 (OK) status code will be returned and the redirect won't occur on the client.

```
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;
    }
}
```

Original Request: `/file.xml`



IRule-based rule

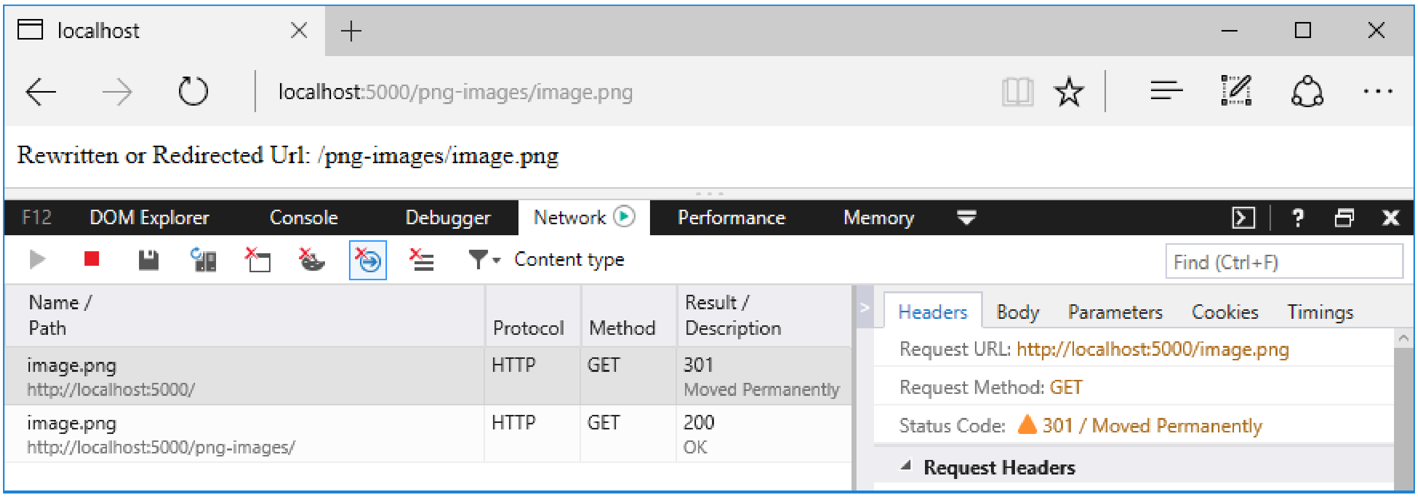
Use `Add(IRule)` to implement your own rule logic in a class that derives from `IRule`. Using an `IRule` provides greater flexibility over using the method-based rule approach. Your derived class may include a constructor, where you can pass in parameters for the `ApplyRule` method.

```
var options = new RewriteOptions()
    .AddRedirect("redirect-rule/(.*)", "redirected/$1")
    .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2", skipRemainingRules: true)
    .AddApacheModRewrite(env.ContentRootFileProvider, "ApacheModRewrite.txt")
    .AddIISUrlRewrite(env.ContentRootFileProvider, "IISUrlRewrite.xml")
    .Add(RedirectXMLRequests)
    .Add(new RedirectImageRequests(".png", "/png-images"))
    .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

app.UseRewriter(options);
```

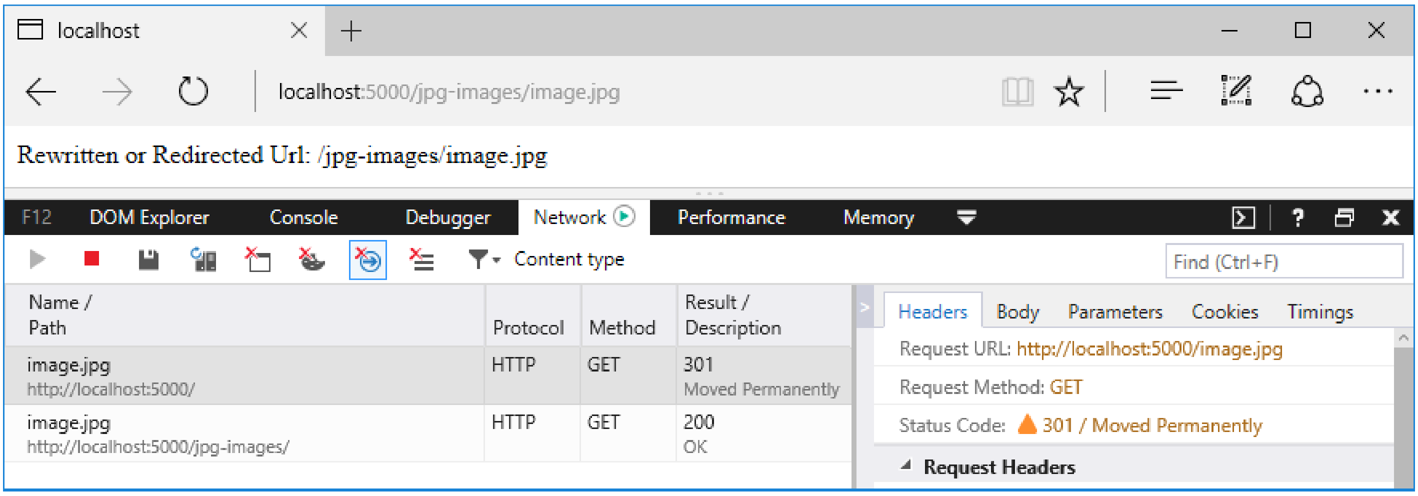
The values of the parameters in the sample application for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be *.png*, *.jpg*, or *.gif*. If the `newPath` isn't valid, an `ArgumentException` is thrown. If you make a request for *image.png*, it's redirected to `/png-images/image.png`. If you make a request for *image.jpg*, it's redirected to `/jpg-images/image.jpg`. The status code is set to 301 (Moved Permanently), and the `context.Result` is set to stop processing rules and send the response.

```
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("The extension is not valid. The extension must be .png, .jpg, or
.gif.", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("The path is not valid. Provide an alphanumeric path that starts with
a forward slash.", 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(new PathString(_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;
        }
    }
}
```

Original Request: `/image.png`



Original Request: `/image.jpg`



Regex examples

GOAL	REGEX STRING & MATCH EXAMPLE	REPLACEMENT STRING & OUTPUT EXAMPLE
Rewrite path into querystring	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
Strip trailing slash	`(.*)/$` `/path/`	`$1` `/path`
Enforce trailing slash	`(.*[^/])$` `/path`	`$1/` `/path/`

Avoid rewriting specific requests	`(.*[^(\.axd)])$`
	Yes: `/resource.htm`
	No: `/resource.axd`
		`rewritten/$1`
		`/rewritten/resource.htm`
		`/resource.axd`

Rearrange URL segments	`path/(.*)/(.*)/(.*)` `path/1/2/3`	`path/$3/$2/$1` `path/3/2/1`
Replace a URL segment	`^(.*)/segment2/(.*)` `/segment1/segment2/segment3`	`$1/replaced/$2` `/segment1/replaced/segment3`

Resources

- Application Startup
- Middleware
- Regular expressions in .NET
- Regular expression language - quick reference
- Apache mod_rewrite
- Using Url Rewrite Module 2.0 (for IIS)
- URL Rewrite Module Configuration Reference
- IIS URL Rewrite Module Forum
- Keep a simple URL structure
- 10 URL Rewriting Tips and Tricks
- To slash or not to slash

Introduction to Error Handling in ASP.NET Core

By Steve Smith and Tom Dykstra

When errors occur in your ASP.NET Core app, you can handle them in a variety of ways, as described in this article.

View or download sample code

The developer exception page

To configure an app to display a page that shows detailed information about exceptions, install the `Microsoft.AspNetCore.Diagnostics` NuGet package and add a line to the Configure method in the Startup class:

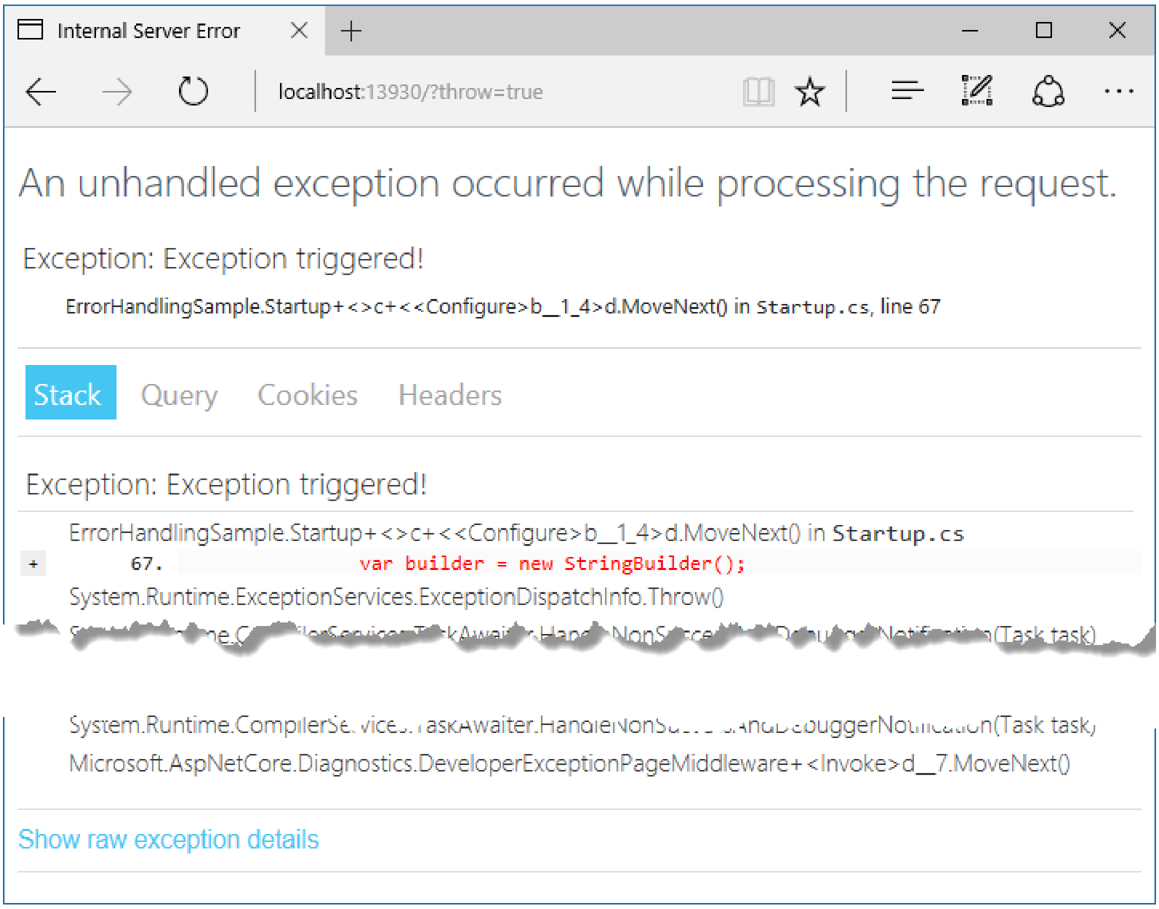
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
```

Put `UseDeveloperExceptionPage` before any middleware you want to catch exceptions in, such as `app.UseMvc`.

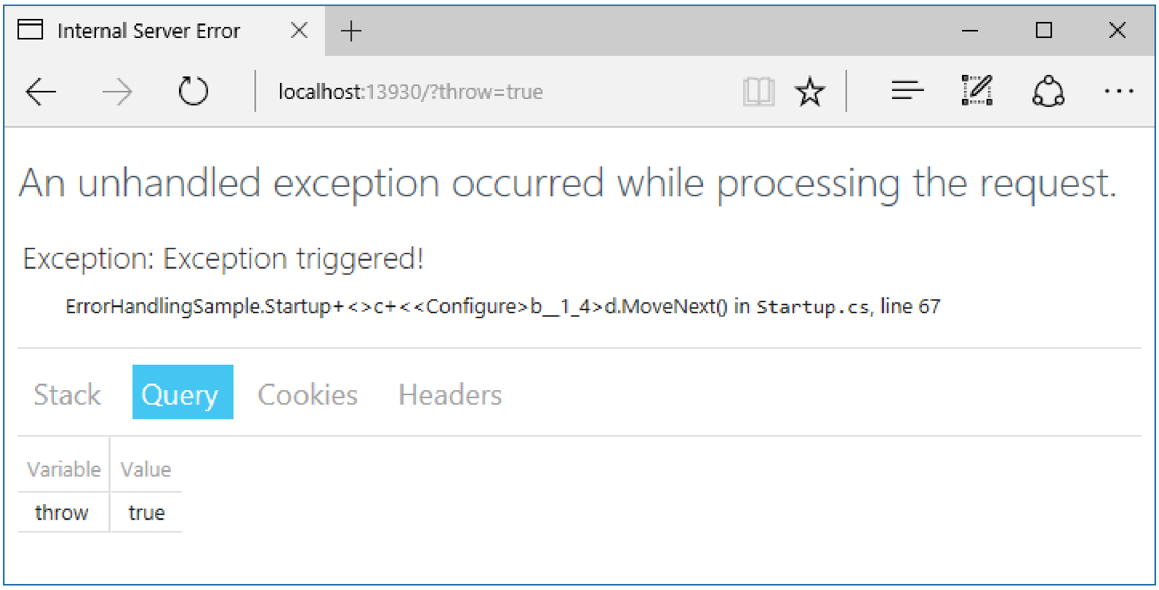
 Warning

Enable the developer exception page **only when the app is running in the Development environment**. You don't want to share detailed exception information publicly when the app runs in production. Learn more about configuring environments.

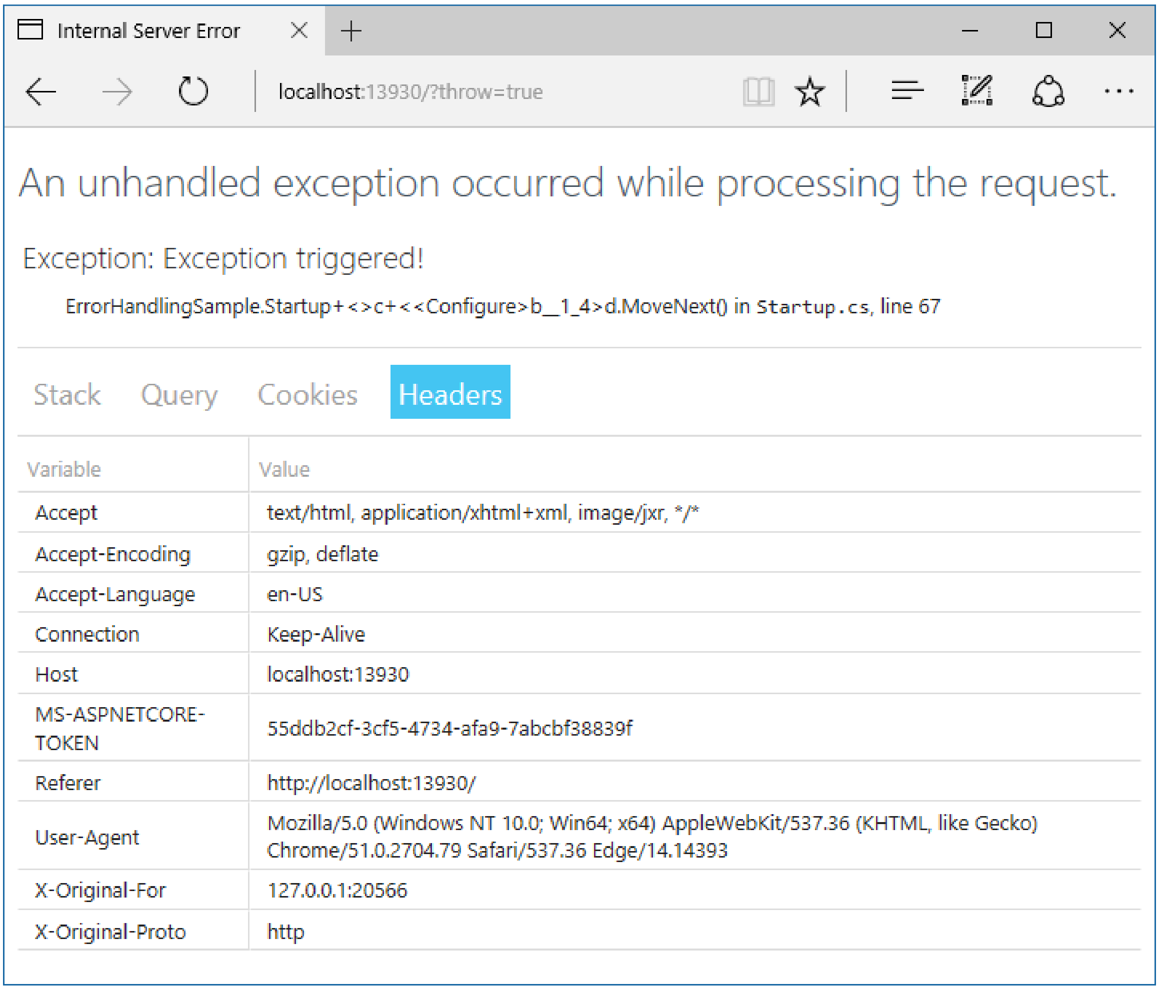
To see the developer exception page, run the sample application with the environment set to `Development`, and add `?throw=true` to the base URL of the app. The page includes several tabs with information about the exception and the request. The first tab includes a stack trace.



The next tab shows the query string parameters, if any.



This request didn't have any cookies, but if it did, they would appear on the **Cookies** tab. You can see the headers that were passed in the last tab.



Configuring a custom exception handling page

It's a good idea to configure an exception handler page to use when the app is not running in the `Development` environment.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    env.EnvironmentName = EnvironmentName.Production;
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
```

In an MVC app, don't explicitly decorate the error handler action method with HTTP method attributes, such as `HttpGet`. Using explicit verbs could prevent some requests from reaching the method.

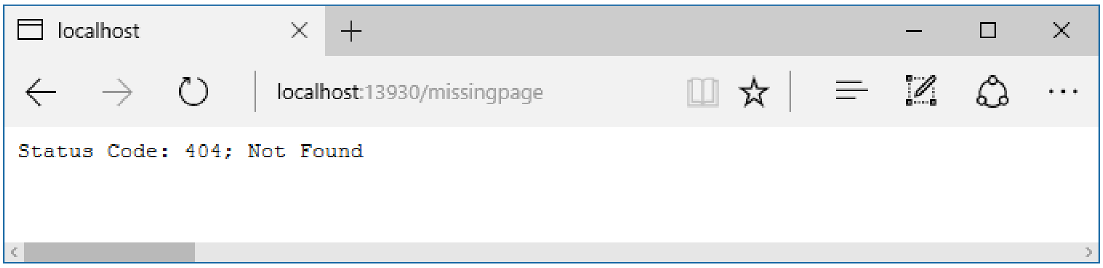
```
[Route("/Error")]
public IActionResult Index()
{
    // Handle error here
}
```

Configuring status code pages

By default, your app will not provide a rich status code page for HTTP status codes such as 500 (Internal Server Error) or 404 (Not Found). You can configure the `StatusCodePagesMiddleware` by adding a line to the `Configure` method:

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

By default, this middleware adds simple, text-only handlers for common status codes, such as 404:



The middleware supports several different extension methods. One takes a lambda expression, another takes a content type and format string.

```
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";
    await context.HttpContext.Response.WriteAsync(
        "Status code page, status code: " +
        context.HttpContext.Response.StatusCode);
});
```

```
app.UseStatusCodePages("text/plain", "Status code page, status code: {0}");
```

There are also redirect extension methods. One sends a 302 status code to the client, and one returns the original status code to the client but also executes the handler for the redirect URL.

```
app.UseStatusCodePagesWithRedirects("/error/{0}");
```

```
app.UseStatusCodePagesWithReExecute("/error/{0}");
```

If you need to disable status code pages for certain requests, you can do so:

```
var statusCodePagesFeature = context.Features.Get<IStatusCodePagesFeature>();
if (statusCodePagesFeature != null)
{
  statusCodePagesFeature.Enabled = false;
}
```

Exception-handling code

Code in exception handling pages can throw exceptions. It's often a good idea for production error pages to consist of purely static content.

Also, be aware that once the headers for a response have been sent, you can't change the response's status code, nor can any exception pages or handlers run. The response must be completed or the connection aborted.

Server exception handling

In addition to the exception handling logic in your app, the server hosting your app will perform some exception handling. If the server catches an exception before the headers have been sent it sends a 500 Internal Server Error response with no body. If it catches an exception after the headers have been sent, it closes the connection. Requests that are not handled by your app will be handled by the server, and any exception that occurs will be handled by the server's exception handling. Any custom error pages or exception handling middleware or filters you have configured for your app will not affect this behavior.

Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. Exceptions that occur during app startup can impact server behavior. For example, if an exception happens before you call `KestrelServerOptions.UseHttps`, the hosting layer catches the exception, starts the server, and displays an error page on the non-SSL port. If an exception happens after that line executes, the error page is served over HTTPS instead.

You can configure how the host will behave in response to errors during startup using `CaptureStartupErrors` and the `detailedErrors` key.

ASP.NET MVC error handling

MVC apps have some additional options for handling errors, such as configuring exception filters and performing model validation.

Exception Filters

Exception filters can be configured globally or on a per-controller or per-action basis in an MVC app. These filters handle any unhandled exception that occurs during the execution of a controller action or another filter, and are not called otherwise. Learn more about exception filters in Filters.

> 汯 Tip
>
> Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen.

Handling Model State Errors

Model validation occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect `ModelState.IsValid` and react appropriately.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a filter may be an appropriate place to implement such a policy. You should test how your actions behave with invalid model states. Learn more in Testing controller logic.

Introduction to WebSockets in ASP.NET Core

By Tom Dykstra and Andrew Stanton-Nurse

This article explains how to get started with WebSockets in ASP.NET Core. WebSocket is a protocol that enables two-way persistent communication channels over TCP connections. It is used for applications such as chat, stock tickers, games, anywhere you want real-time functionality in a web application.

View or download sample code. See the Next Steps section for more information.

Prerequisites

ASP.NET Core 1.1 (does not run on 1.0)

Any OS that ASP.NET Core runs on:

Windows 7 / Windows Server 2008 and later
Linux

macOS

Exception: If your app runs on Windows with IIS, or with WebListener, you must use Windows 8 / Windows Server 2012 or later

For supported browsers, see http://caniuse.com/#feat=websockets.

When to use it

Use WebSockets when you need to work directly with a socket connection. For example, you might need the best possible performance for a real-time game.

ASP.NET SignalR provides a richer application model for real-time functionality, but it runs only on ASP.NET, not ASP.NET Core. A Core version of SignalR is under development; to follow its progress, see the GitHub repository for SignalR Core.

If you don't want to wait for SignalR Core, you can use WebSockets directly now. But you might have to develop features that SignalR would provide, such as:

Support for a broader range of browser versions by using automatic fallback to alternative transport methods.
Automatic reconnection when a connection drops.
Support for clients calling methods on the server or vice versa.
Support for scaling to multiple servers.

How to use it

Install the Microsoft.AspNetCore.WebSockets package.
Configure the middleware.
Accept WebSocket requests.
Send and receive messages.

Configure the middleware

Add the WebSockets middleware in the `Configure` method of the `Startup` class.

```
app.UseWebSockets();
```

The following settings can be configured:

`KeepAliveInterval` - How frequently to send "ping" frames to the client, to ensure proxies keep the connection open.

`ReceiveBufferSize` - The size of the buffer used to receive data. Only advanced users would need to change this, for performance tuning based on the size of their data.

```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
app.UseWebSockets(webSocketOptions);
```

Accept WebSocket requests

Somewhere later in the request life cycle (later in the `Configure` method or in an MVC action, for example) check if it's a WebSocket request and accept the WebSocket request.

This example is from later in the `Configure` method.

```
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();
    }

});
```

A WebSocket request could come in on any URL, but this sample code only accepts requests for `/ws`.

Send and receive messages

The `AcceptWebSocketAsync` method upgrades the TCP connection to a WebSocket connection and gives you a WebSocket object. Use the WebSocket object to send and receive messages.

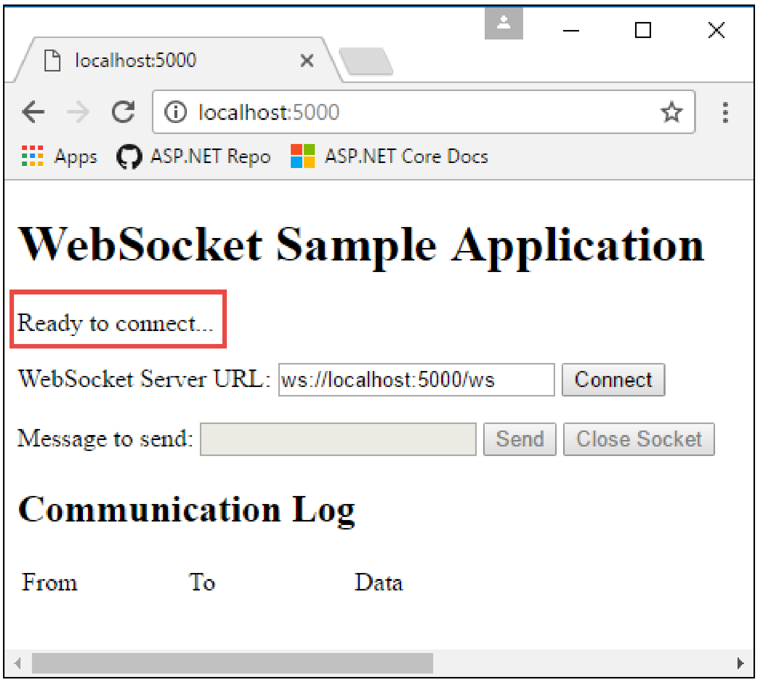
The code shown earlier that accepts the WebSocket request passes the `WebSocket` object to an `Echo` method; here's the `Echo` method. The code receives a message and immediately sends back the same message. It stays in a loop doing that until the client closes the connection.

```
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);
}
```

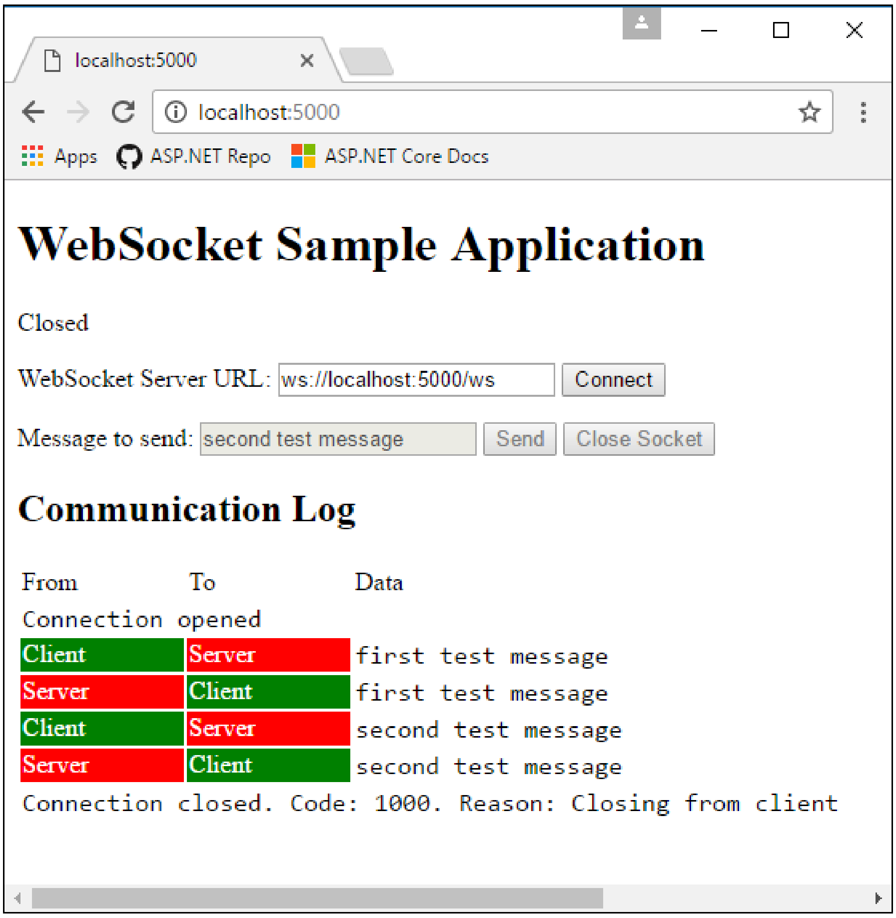
When you accept the WebSocket before beginning this loop, the middleware pipeline ends. Upon closing the socket, the pipeline unwinds. That is, the request stops moving forward in the pipeline when you accept a WebSocket, just as it would when you hit an MVC action, for example. But when you finish this loop and close the socket, the request proceeds back up the pipeline.

Next steps

The sample application that accompanies this article is a simple echo application. It has a web page that makes WebSocket connections, and the server just resends back to the client any messages it receives. Run it from a command prompt (it's not set up to run from Visual Studio with IIS Express) and navigate to http://localhost:5000. The web page shows connection status at the upper left:



Select **Connect** to send a WebSocket request to the URL shown. Enter a test message and select **Send**. When done, select **Close Socket**. The **Communication Log** section reports each open, send, and close action as it happens.


localhost:5000
localhost:5000
Apps
ASP.NET Repo
ASP.NET Core Docs
WebSocket Sample Application
Closed
WebSocket Server URL:
ws://localhost:5000/ws
Connect
Message to send:
second test message
Send
Close Socket
Communication Log
From
To
Data
Connection opened
Client
Server
first test message
Server
Client
first test message
Client
Server
second test message
Server
Client
second test message
Connection closed. Code: 1000. Reason: Closing from client

Globalization and localization

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

Creating a multilingual website with ASP.NET Core will allow your site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

Make the app's content localizable

Provide localized resources for the languages and cultures you support

Implement a strategy to select the language/culture for each request

Make the app's content localizable

Introduced in ASP.NET Core, `IStringLocalizer` and `IStringLocalizer<T>` were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The simple interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require you to store the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
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"];
        }
    }
}
```

In the code above, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" is not found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but not the resource string. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
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();
        }
```

Note: You generally want to only localize text and not HTML.

At the lowest level, you can get `IStringLocalizerFactory` out of [Dependency Injection](#):

```
{
    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);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization.StarterWeb
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

```
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;
    }
```

View localization

The `IViewLocalizer` service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@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>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There is no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

KEY	VALUE
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

Notes:

View localization requires the "Localization.AspNetCore.TagHelpers" NuGet package.
You generally want to only localize text and not HTML.

To use a shared resource file in a view, inject `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 localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option `ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

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 e-mail 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; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

Provide localized resources for the languages and cultures you support

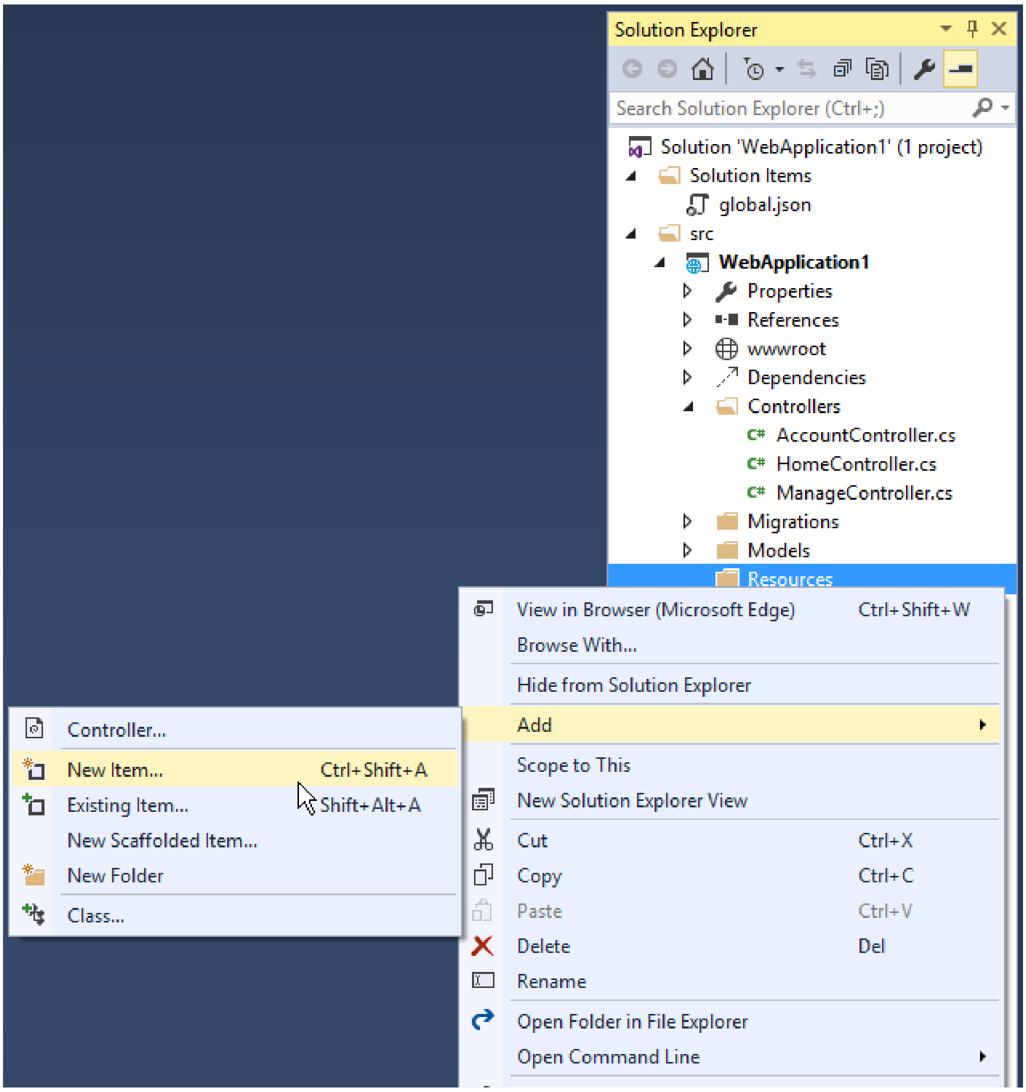
SupportedCultures and SupportedUICultures

ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See CultureInfo.CurrentCulture for more info on how the server gets the Culture. The `SupportedUICultures` determines which translates strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that is determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

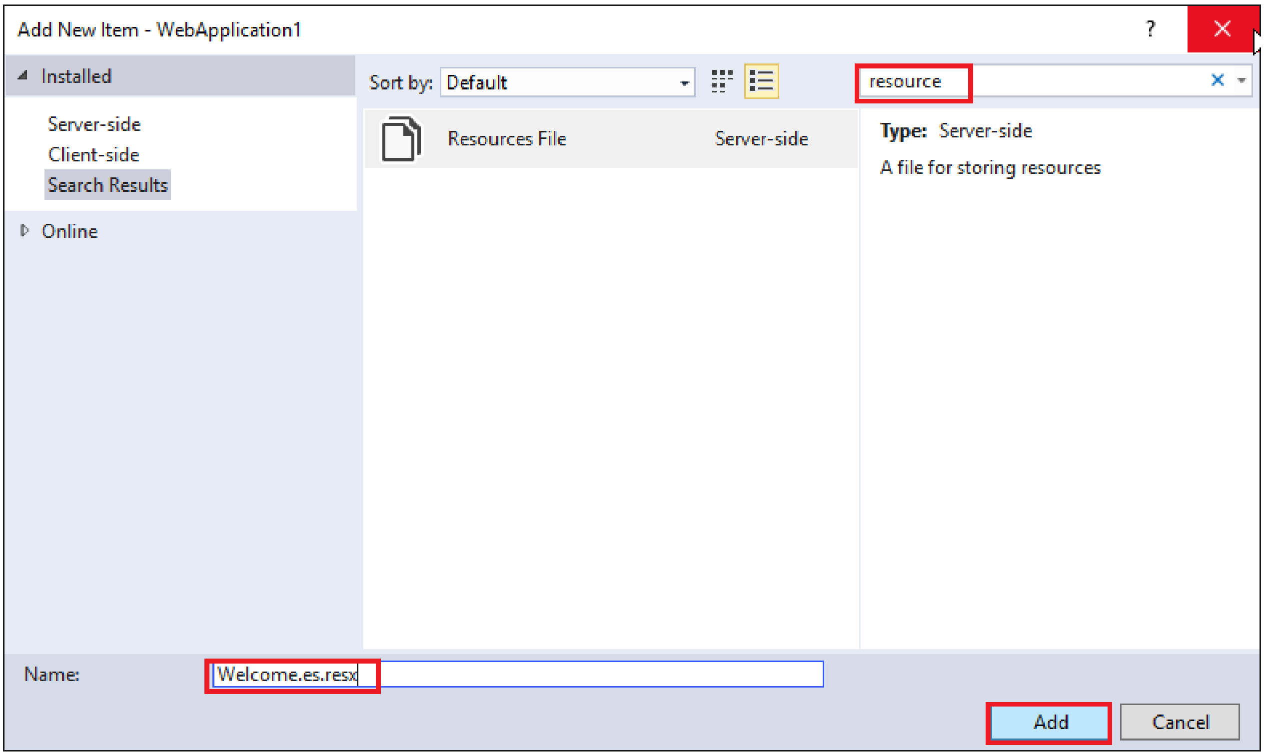
Working with resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated *.resx* resource files. For example, you might want to create Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

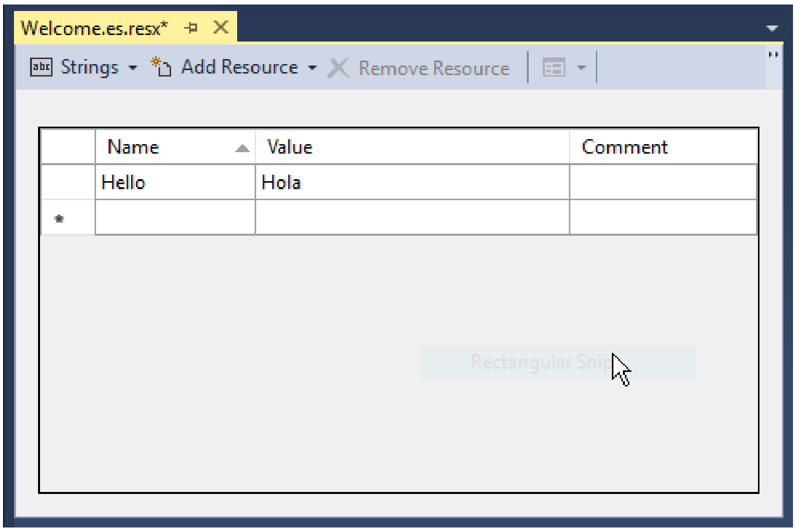
In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



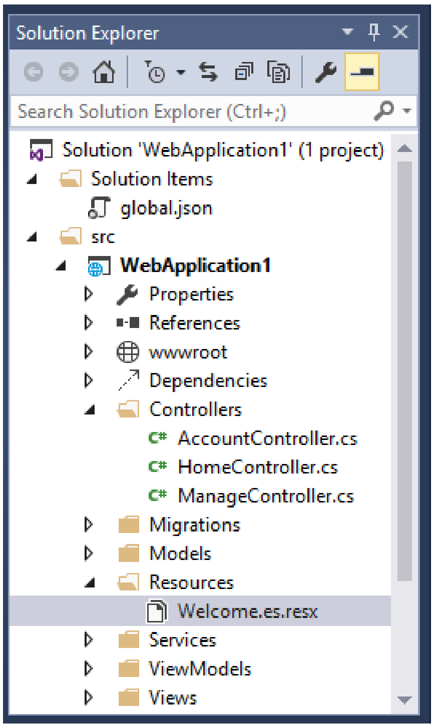
In the **Search installed templates** box, enter "resource" and name the file.



Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace is not the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

RESOURCE NAME	DOT OR PATH NAMING
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Home/About.cshtml* view could be either of the following:

Resources/Views/Home/About.fr.resx

Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the view.

If you remove the ".fr" culture designator AND you have the culture set to French (via cookie or other mechanism), the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource, when nothing meets your requested culture you're served the *.resx file without a culture designator. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generating resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core; you typically don't have a default *.resx* resource file (A *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio will not generate the class file. We anticipate that many developers will **not** create a default language resource file.

Adding Other Cultures

Each language and culture combination (other than the default language) requires a unique resource file. You can create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the .resx file name extension, as in *Welcome.es-MX.resx* (Spanish/Mexico). To specify a culturally neutral language, you would eliminate the country code, such as *Welcome.fr.resx* for the French language.

Implement a strategy to select the language/culture for each request

Configuring localization

Localization is configured in the `ConfigureServices` method:

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
  .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
  .AddDataAnnotationsLocalization();
```

`AddLocalization` Adds the localization services to the services container. The code above also sets the resources path to "Resources".

`AddViewLocalization` Adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.

`AddDataAnnotationsLocalization` Adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in the `Configure` method of *Startup.cs* file. Note, the localization middleware must be configured before any middleware which might check the request culture (for example, `app.UseMvc()`).

```
            var supportedCultures = new[]
            {
                new CultureInfo("en-US"),
                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("en-US"),
                // Formatting numbers, dates, etc.
                SupportedCultures = supportedCultures,
                // UI strings that we have localized.
                SupportedUICultures = supportedCultures
            });

            app.UseStaticFiles();

            app.UseIdentity();

            // To configure external authentication please see http://go.microsoft.com/fwlink/?LinkID=532715

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

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

`QueryStringRequestCultureProvider`
`CookieRequestCultureProvider`
`AcceptLanguageHeaderRequestCultureProvider`

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the culture and UI culture. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

`http://localhost:5000/?culture=es-MX`

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider` `DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is ".AspNetCore.Culture".

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

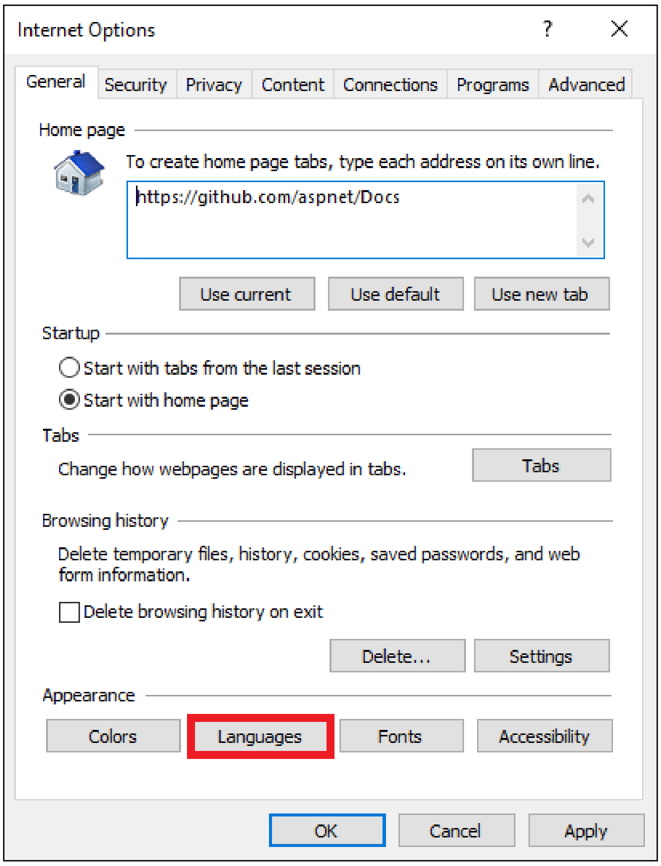
The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request is not an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Setting the Accept-Language HTTP header in IE

From the gear icon, tap **Internet Options**.

Tap **Languages**.



Tap **Set Language Preferences**.

Tap **Add a language**.

Add the language.

Tap the language, then tap **Move Up**.

Using a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```
services.Configure<RequestLocalizationOptions>(options =>
   {
       var supportedCultures = new[]
       {
           new CultureInfo("en-US"),
           new CultureInfo("fr")
       };

       options.DefaultRequestCulture = new RequestCulture(culture: "en-US", uiCulture: "en-US");
       options.SupportedCultures = supportedCultures;
       options.SupportedUICultures = supportedCultures;

       options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
       {
         // My custom request culture logic
         return new ProviderCultureResult("en");
       }));
   });
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Setting the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The *Views/Shared/_SelectLanguagePartial.cshtml* file allows you to select the culture from the list of supported cultures:

```
@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.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@Context.Request.Path"
          method="post" class="form-horizontal" role="form">
        @Localizer["Language:"] <select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

The *Views/Shared/_SelectLanguagePartial.cshtml* file is added to the `footer` section of the layout file so it will be available to all views:

```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; 2015 - Localization.StarterWeb</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture 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);
}
```

You can't plug in the *_SelectLanguagePartial.cshtml* to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is "-", where is the language code and is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See Language Culture Name.

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

Globalization (G11N): The process of making an app support different languages and regions.
Localization (L10N): The process of customizing an app for a given language and region.
Internationalization (I18N): Describes both globalization and localization.
Culture: It is a language and, optionally, a region.
Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")
Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")
Locale: A locale is the same as a culture.

Additional Resources

Localization.StarterWeb project used in the article.

Resource Files in Visual Studio

Resources in .resx Files

Configuration in ASP.NET Core

Rick Anderson, Mark Michaelis, Steve Smith, Daniel Roth

The configuration API provides a way of configuring an app based on a list of name-value pairs that can be read at runtime from multiple sources. The name-value pairs can be grouped into a multi-level hierarchy. There are configuration providers for:

File formats (INI, JSON, and XML)
Command-line arguments
Environment variables
In-memory .NET objects
An encrypted user store
Azure Key Vault
Custom providers, which you install or create

Each configuration value maps to a string key. There's built-in binding support to deserialize settings into a custom POCO object (a simple .NET class with properties).

View or download sample code

Simple configuration

The following console app uses the JSON configuration provider:

```
using Microsoft.Extensions.Configuration;
using System;
using System.IO;

// Add NuGet <package id="Microsoft.Extensions.Configuration" and
// <package id="Microsoft.Extensions.Configuration.Json"
// .NET Framework 4.x use the following path:
//.SetBasePath(Path.Combine(Directory.GetCurrentDirectory(), @"..\.."))

public class Program
{
    public static IConfigurationRoot 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($"{Configuration["wizards:0:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:0:Age"]}");
        Console.Write($"{Configuration["wizards:1:Name"]}, ");
        Console.WriteLine($"age {Configuration["wizards:1:Age"]}");
    }
}
```

The app reads and displays the following configuration settings:

```
{
  "option1": "value1_from_json",
  "option2": 2,

  "subsection": {
    "suboption1": "subvalue1_from_json"
  },
  "wizards": [
    {
      "Name": "Gandalf",
      "Age": "1000"
    },
    {
      "Name": "Harry",
      "Age": "17"
    }
  ]
}
```

Configuration consists of a hierarchical list of name-value pairs in which the nodes are separated by a colon. To retrieve a value, you access the `Configuration` indexer with the corresponding item's key:

```
Console.WriteLine($"option1 = {Configuration["subsection:suboption1"]}");
```

To work with arrays in JSON-formatted configuration sources, use a zero-based array index as part of the colon-separated string used to retrieve the value. For instance, to get the name of the first item in the `wizards` array shown above, use the following code:

```
Console.Write($"{Configuration["wizards:0:Name"]}, ");
```

Name/value pairs written to the built in `Configuration` providers are **not** persisted, however, you can create a custom provider that saves values. See custom configuration provider.

The sample above uses the configuration indexer to read values. When the `Configuration` object is available, the indexer is a convenient way to access setting. To flow information from configuration into other parts of the app outside of `Startup`, we recommend using the options pattern. We'll show that later in this document.

It's typical to have different configuration settings for different environments, for example, development, test and production. The following highlighted code hooks up two configuration providers to three sources:

JSON provider, reading *appsettings.json*
JSON provider, reading *appsettings.<EnvironmentName>.json*
Environment variables provider

```
public class Startup
{
    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();
    }
```

See AddJsonFile for an explanation of the parameters. `reloadOnChange` is only supported in ASP.NET Core 1.1 and higher.

Configuration sources are read in the order they are specified. In the code above, the environment variables are read last. Any configuration values set through the environment would replace those set in the two previous providers.

The environment is typically set to one of `Development`, `Staging`, or `Production`. See Working with Multiple Environments for more information.

Configuration considerations:

`IOptionsSnapshot` can reload configuration data when it changes. Use `IOptionsSnapshot` if you need to reload configuration data. See IOptionsSnapshot for more information.
Configuration keys are case insensitive.
A best practice is to specify environment variables last, so that the local environment can override anything set in deployed configuration files.
Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. You also shouldn't use production secrets in your development or test environments. Instead, specify secrets outside the project tree, so they cannot be accidentally committed into your repository. Learn more about Working with Multiple Environments and managing Safe storage of app secrets during development.
If `:` cannot be used in environment variables in your system, replace `:` with `__` (double underscore).

Using Options and configuration objects

The options pattern uses custom options classes to represent a group of related settings. We recommended that you create decoupled classes for each feature within your app. Decoupled classes follow:

The Interface Segregation Principle (ISP) : Classes depend only on the configuration settings they use.
Separation of Concerns : Settings for different parts of your app are not dependent or coupled with one another.

The options class must be non-abstract with a public parameterless constructor. For example:

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }
    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

In the following code, the JSON configuration provider is enabled. The `MyOptions` class is added to the service container and bound to configuration.

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        // Set up configuration sources.
        var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);

        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; set; }

    public void ConfigureServices(IServiceCollection services)
    {
        // Adds services required for using options.
        services.AddOptions();

        // Register the IConfiguration instance which MyOptions binds against.
        services.Configure<MyOptions>(Configuration);

        // Add framework services.
        services.AddMvc();
    }
```

The following controller uses Dependency Injection on `IOptions<TOptions>` to access settings:

```
public class HomeController : Controller
{
    private readonly MyOptions _options;

    public HomeController(IOptions<MyOptions> optionsAccessor)
    {
        _options = optionsAccessor.Value;
    }

    public IActionResult Index()
    {
        var option1 = _options.Option1;
        var option2 = _options.Option2;
        return Content($"option1 = {option1}, option2 = {option2}");
    }
}
```

With the following *appsettings.json* file:

```
{
  "option1": "value1_from_json",
  "option2": 2
}
```

The `HomeController.Index` method returns `option1 = value1_from_json, option2 = 2`.

Typical apps won't bind the entire configuration to a single options file. Later on I'll show how to use `GetSection` to bind to a section.

In the following code, a second `IConfigureOptions<TOptions>` service is added to the service container. It uses a delegate to configure the binding with `MyOptions`.

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds services required for using options.
    services.AddOptions();

    // Register the ConfigurationBuilder instance which MyOptions binds against.
    services.Configure<MyOptions>(Configuration);

    // Registers the following lambda used to configure options.
    services.Configure<MyOptions>( myOptions =>
    {
        myOptions.Option1 = "value1_from_action";
    });

    // Add framework services.
    services.AddMvc();
}
```

You can add multiple configuration providers. Configuration providers are available in NuGet packages. They are applied in order they are registered.

Each call to `Configure<TOptions>` adds an `IConfigureOptions<TOptions>` service to the service container. In the example above, the values of `Option1` and `Option2` are both specified in *appsettings.json*, but the value of `Option1` is overridden by the configured delegate in the highlighted code above.

When more than one configuration service is enabled, the last configuration source specified "wins". With the code above, the `HomeController.Index` method returns `option1 = value1_from_action, option2 = 2`.

When you bind options to configuration, each property in your options type is bound to a configuration key of the form `property[:sub-property:]`. For example, the `MyOptions.Option1` property is bound to the key `Option1`, which is read from the `option1` property in *appsettings.json*. A sub-property sample is shown later in this article.

In the following code, a third `IConfigureOptions<TOptions>` service is added to the service container. It binds `MySubOptions` to the section `subsection` of the *appsettings.json* file:

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds services required for using options.
    services.AddOptions();

    // Configure with Microsoft.Extensions.Options.ConfigurationExtensions
    // Binding the whole configuration should be rare, subsections are more typical.
    services.Configure<MyOptions>(Configuration);

    // Configure MyOptions using code.
    services.Configure<MyOptions>(myOptions =>
    {
        myOptions.Option1 = "value1_from_action";
    });

    // Configure using a sub-section of the appsettings.json file.
    services.Configure<MySubOptions>(Configuration.GetSection("subsection"));

    // Add framework services.
    services.AddMvc();
}
```

Note: This extension method requires the `Microsoft.Extensions.Options.ConfigurationExtensions` NuGet package.

Using the following *appsettings.json* file:

```
{
  "option1": "value1_from_json",
  "option2": -1,

  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  }
}
```

The `MySubOptions` class:

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }
    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

With the following `Controller`:

```
public class HomeController : Controller
{
    private readonly MySubOptions _subOptions;

    public HomeController(IOptions<MySubOptions> subOptionsAccessor)
    {
        _subOptions = subOptionsAccessor.Value;
    }

    public IActionResult Index()
    {
        var subOption1 = _subOptions.SubOption1;
        var subOption2 = _subOptions.SubOption2;
        return Content($"subOption1 = {subOption1}, subOption2 = {subOption2}");
    }
}
```

`subOption1 = subvalue1_from_json, subOption2 = 200` is returned.

IOptionsSnapshot

Requires ASP.NET Core 1.1 or higher.

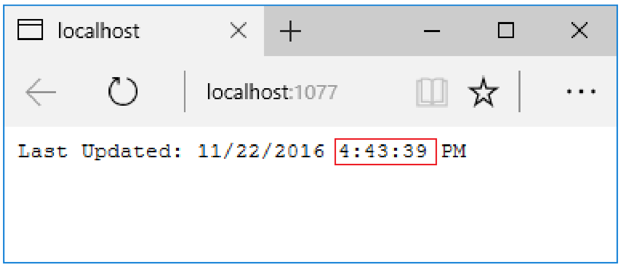
`IOptionsSnapshot` supports reloading configuration data when the configuration file has changed. It also has minimal overhead if you don't care about changes. Using `IOptionsSnapshot` with `reloadOnChange: true`, the options are bound to `IConfiguration` and reloaded when changed.

The following sample demonstrates how a new `IOptionsSnapshot` is created after *config.json* changes. Requests to server will return the same time when *config.json* has **not** changed. The first request after *config.json* changes will show a new time.

```
public class TimeOptions
{
    // Records the time when the options are created.
    public DateTime CreationTime { get; set; } = DateTime.Now;

    // Bound to config. Changes to the value of "Message"
```

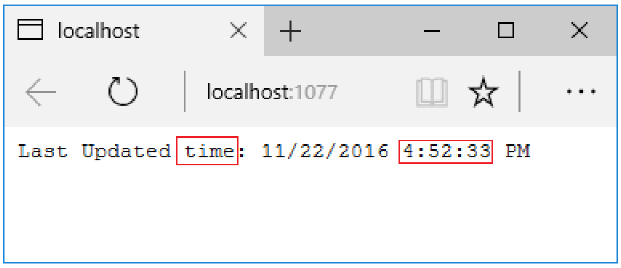
```
    // Bound to config. Changes to the value of "Message"
    // in config.json will be reflected in this property.
    public string Message { get; set; }
}

public class Controller
{
    public readonly TimeOptions _options;

    public Controller(IOptionsSnapshot<TimeOptions> options)
    {
        _options = options.Value;
    }

    public Task DisplayTimeAsync(HttpContext context)
    {
        return context.Response.WriteAsync(_options.Message + _options.CreationTime);
    }
}

public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            // reloadOnChange: true is required for config changes to be detected.
            .AddJsonFile("config.json", optional: false, reloadOnChange: true)
            .AddEnvironmentVariables();
        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; set; }

    public void Configure(IApplicationBuilder app)
    {
        // Simple mockup of a simple per request controller that writes
        // the creation time and message of TimeOptions.
        app.Run(DisplayTimeAsync);
    }

    public void ConfigureServices(IServiceCollection services)
    {
        // Simple mockup of a simple per request controller.
        services.AddScoped<Controller>();

        // Binds config.json to the options and setups the change tracking.
        services.Configure<TimeOptions>(Configuration.GetSection("Time"));
    }

    public Task DisplayTimeAsync(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        return context.RequestServices.GetRequiredService<Controller>().DisplayTimeAsync(context);
    }

    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseIISIntegration()
            .UseStartup<Startup>()
            .Build();
        host.Run();
    }
}
```

The following image shows the server output:



Refreshing the browser doesn't change the message value or time displayed (when *config.json* has not changed).

Change and save the *config.json* and then refresh the browser:



In-memory provider and binding to a POCO class

The following sample shows how to use the in-memory provider and bind to a class:

```
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;

// Add NuGet  <package id="Microsoft.Extensions.Configuration.Binder"
public class Program
{
    public static IConfigurationRoot 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();
        Configuration.GetSection("App:MainWindow").Bind(window);
        Console.WriteLine($"Left {window.Left}");
    }
}
```

Configuration values are returned as strings, but binding enables the construction of objects. Binding allows you to retrieve POCO objects or even entire object graphs. The following sample shows how to bind to `MyWindow` and use the options pattern with a ASP.NET Core MVC app:

```
public class MyWindow
{
    public int Height { get; set; }
    public int Width { get; set; }
    public int Top { get; set; }
    public int Left { get; set; }
}
```

```
{
  "AppConfiguration": {
    "MainWindow": {
      "Height": "400",
      "Width": "600",
      "Top": "5",
      "Left": "11"
    }
  }
}
```

Bind the custom class in `ConfigureServices` in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyWindow>(
        Configuration.GetSection("AppConfiguration:MainWindow"));
    services.AddMvc();
}
```

Display the settings from the `HomeController`:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Options;

public class HomeController : Controller
{
    private readonly IOptions<MyWindow> _optionsAccessor;

    public HomeController(IOptions<MyWindow> optionsAccessor)
    {
        _optionsAccessor = optionsAccessor;
    }
    public IActionResult Index()
    {
        var height = _optionsAccessor.Value.Height;
        var width = _optionsAccessor.Value.Width;
        var left = _optionsAccessor.Value.Left;
        var top = _optionsAccessor.Value.Top;

        return Content($"height = {height}, width = {width}, "
                    + $"Left = {left}, Top = {top}");
    }
}
```

GetValue

The following sample demonstrates the `GetValue<T>` extension method:

```
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;

// Add NuGet  <package id="Microsoft.Extensions.Configuration.Binder"
public class Program
{
    public static IConfigurationRoot 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}");
    }
}
```

The ConfigurationBinder's `GetValue<T>` method allows you to specify a default value (80 in the sample). `GetValue<T>` is for simple scenarios and does not bind to entire sections. `GetValue<T>` gets scalar values from `GetSection(key).Value` converted to a specific type.

Binding to an object graph

You can recursively bind to each object in a class. Consider the following `AppOptions` class:

```
public class AppOptions
{
    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; }
}
```

The following sample binds to the `AppOptions` class:

```
using Microsoft.Extensions.Configuration;
using System;
using System.IO;

// Add these NuGet packages:
// "Microsoft.Extensions.Configuration.Binder"
// "Microsoft.Extensions.Configuration.FileExtensions"
// "Microsoft.Extensions.Configuration.Json":
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 AppOptions();
        config.GetSection("App").Bind(appConfig);

        Console.WriteLine($"Height {appConfig.Window.Height}");
    }
}
```

ASP.NET Core 1.1 and higher can use `Get<T>`, which works with entire sections. `Get<T>` can be more convienent than using `Bind`. The following code shows how to use `Get<T>` with the sample above:

```
var appConfig = config.GetSection("App").Get<AppOptions>();
```

Using the following *appsettings.json* file:

```json
{
  "App": {
    "Profile": {
      "Machine": "Rick"
    },
    "Connection": {
      "Value": "connectionstring"
    },
    "Window": {
      "Height": "11",
      "Width": "11"
    }
  }
}
```

The program displays `Height 11`.

The following code can be used to unit test the configuration:

```csharp
[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 options = new AppOptions();
    config.GetSection("App").Bind(options);

    Assert.Equal("Rick", options.Profile.Machine);
    Assert.Equal(11, options.Window.Height);
    Assert.Equal(11, options.Window.Width);
    Assert.Equal("connectionstring", options.Connection.Value);
}
```

Simple sample of Entity Framework custom provider

In this section, we'll create a simple configuration provider that reads name-value pairs from a database using EF.

Define a `ConfigurationValue` entity for storing configuration values in the database:

```csharp
public class ConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```

Add a `ConfigurationContext` to store and access the configured values:

```
public class ConfigurationContext : DbContext
{
    public ConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<ConfigurationValue> Values { get; set; }
}
```

Create an class that implements IConfigurationSource:

```
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);
        }
    }
}
```

Create the custom configuration provider by inheriting from ConfigurationProvider. The configuration provider initializes the database when it's empty:

```
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;
        }
    }
}
```

The highlighted values from the database ("value_from_ef_1" and "value_from_ef_2") are displayed when the sample is run.

You can add an `EFConfigSource` extension method for adding the configuration source:

```
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));
        }
    }
}
```

The following code shows how to use the custom `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();
        builder.SetBasePath(Directory.GetCurrentDirectory());
        builder.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"]);
    }
}
```

Note the sample adds the custom `EFConfigProvider` after the JSON provider, so any settings from the database will override settings from the *appsettings.json* file.

Using the following *appsettings.json* file:

```
{
  "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"
}
```

The following is displayed:

```
key1=value_from_ef_1
key2=value_from_ef_2
key3=value_from_json_3
```

CommandLine configuration provider

The following sample enables the CommandLine configuration provider last:

```
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;
using System.Linq;

// Add NuGet  <package id="Microsoft.Extensions.Configuration.Binder"
// Add NuGet  <package id="Microsoft.Extensions.Configuration.CommandLine"
public class Program
{
    public static IConfigurationRoot 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", "Rick"},
                {"App:MainWindow:Left", "11"}
            };

        var builder = new ConfigurationBuilder();
        builder.AddInMemoryCollection(dict)
              .AddCommandLine(args, GetSwitchMappings(dict));
        Configuration = builder.Build();
        Console.WriteLine($"Hello {Configuration["Profile:MachineName"]}");

        // Set the default value to 80
        var left = Configuration.GetValue<int>("App:MainWindow:Left", 80);
        Console.WriteLine($"Left {left}");
    }
}
```

Use the following to pass in configuration settings:

```
dotnet run /Profile:MachineName=Bob /App:MainWindow:Left=1234
```

Which displays:

```
Hello Bob
Left 1234
```

The `GetSwitchMappings` method allows you to use `-` rather than `/` and it strips the leading subkey prefixes. For example:

```
dotnet run -MachineName=Bob -Left=7734
```

Displays:

```
Hello Bob
Left 7734
```

Command-line arguments must include a value (it can be null). For example:

```
dotnet run /Profile:MachineName=
```

Is OK, but

```
dotnet run /Profile:MachineName
```

results in an exception. An exception will be thrown if you specify a command-line switch prefix of - or -- for which there's no corresponding switch mapping.

The *web.config* file

web.config is required when you host the app in IIS or IIS-Express. It turns on the AspNetCoreModule in IIS to launch your app. It may also be used to configure other IIS settings and modules. If you are using Visual Studio and delete *web.config*, Visual Studio will create a new one.

Additional notes

Dependency Injection (DI) is not setup until after `ConfigureServices` is invoked and the configuration system is not DI aware.

`IConfiguration` has two specializations:

`IConfigurationRoot` Used for the root node. Can trigger a reload.
`IConfigurationSection` Represents a section of configuration values. The `GetSection` and `GetChildren` methods return an `IConfigurationSection`.

Additional Resources

[Working with Multiple Environments]
[Safe storage of app secrets during development]
[Dependency Injection]
[Azure Key Vault configuration provider]

Introduction to Logging in ASP.NET Core

By Steve Smith and Tom Dykstra

ASP.NET Core supports a logging API that works with a variety of logging providers. Built-in providers let you send logs to one or more destinations, and you can plug in a third-party logging framework. This article shows how to use the built-in logging API and providers in your code.

View or download sample code

How to add providers

A logging provider takes some action on logged data, such as display it on the console or store it in Azure blob storage. To use a provider, install its NuGet package and call the provider's extension method on an instance of `ILoggerFactory`, as shown in the following example.

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole()
        .AddDebug();
```

ASP.NET Core dependency injection (DI) provides the `ILoggerFactory` instance. The `AddConsole` and `AddDebug` extension methods are defined in the Microsoft.Extensions.Logging.Console and Microsoft.Extensions.Logging.Debug packages. Each extension method calls the `ILoggerFactory.AddProvider` method, passing in an instance of the provider.

Note

The sample application for this article adds logging providers in the `Configure` method of the `Startup` class. If you want to get log output from code that executes earlier, add logging providers in the `Startup` class constructor instead.

You'll find information about each built-in logging provider and links to third-party logging providers later in the article.

How to create logs

To create logs, get an `ILogger` object from DI and store it in a field, then call logging methods on that logger object.

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

This example requests `ILogger<TodoController>` from DI to specify the `TodoController` class as the *category* of logs that are created with the logger. Categories are explained later in this article.

Sample logging output

With the sample code shown above, you'll see logs in the console when you run from the command line, and in the Debug window when you run in Visual Studio in Debug mode.

Here's an example of what you see in the console if you run the sample application from the command line and go to URL `http://localhost:5000/api/todo/0`:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/invalidid
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (invalidid)
- ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) 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 243.2636ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 628.9188ms 404
```

Here's an example of what you see in the Debug window if you run the sample application from Visual Studio in debug mode and go to URL `http://localhost:55070/api/todo/0`:

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:55070/api/todo/invalidid
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (invalidid) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item invalidid
TodoApi.Controllers.TodoController:Warning: GetById(invalidid) 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 12.5003ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 19.0913ms 404
```

From these examples you can see that ASP.NET Core itself and your application code are using the same logging API and the same logging providers.

The remainder of this article explains some details and options for logging.

NuGet packages

The `ILogger` and `ILoggerFactory` interfaces are in Microsoft.Extensions.Logging.Abstractions, and default implementations for them are in Microsoft.Extensions.Logging.

Log category

A *category* is specified with each log that you create. The category may be any string, but a convention is to use the fully qualified name of the class from which the logs are written. For example: "TodoApi.Controllers.TodoController".

You specify the category when you create a logger object or request one from DI, and the category is automatically included with every log written by that logger. You can specify the category explicitly or you can use an extension method that derives the category from the type. To specify the category explicitly, call `CreateLogger` on an *ILoggerFactory* instance, as shown below.

```
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");
    }
```

Most of the time it will be easier to use `ILogger<T>`, as in the following example.

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

This is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

Each time you write a log, you specify its LogLevel. The log level indicates the degree of severity or importance. For example, you might write an `Information` log when a method ends normally, a `Warning` log when a method returns a 404 return code, and an `Error` log when you catch an unexpected exception.

In the following code example, the names of the methods specify the log level, the first parameter is the Log event ID, and the remaining parameters construct a log message:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

Log methods that include the level in the method name are extension methods for ILogger that the *Microsoft.Extensions.Logging* package provides. Behind the scenes these methods call a `Log` method that takes a `LogLevel` parameter. You can call the `Log` method directly rather than one of these extension methods, but the syntax is relatively complicated. For more information, see the ILogger interface and the logger extensions source code.

ASP.NET Core defines the following log levels, ordered here from least to highest severity.

Trace = 0

For information that is valuable only to a developer debugging an issue. These messages may contain sensitive application data and so should not be enabled in a production environment. *Disabled by default.* Example: `Credentials: {"User":"someuser", "Password":"P@ssword"}`

Debug = 1

For information that has short-term usefulness during development and debugging. Example: `Entering method Configure with flag set to true.`

Information = 2

For tracking the general flow of the application. These logs typically have some long-term value. Example: `Request received for path /api/todo`

Warning = 3

For abnormal or unexpected events in the application flow. These may include errors or other conditions that do not cause the application to stop, but which may need to be investigated. Handled exceptions are a common place to use the `Warning` log level. Example: `FileNotFoundException for file quotes.txt.`

Error = 4

For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an application-wide failure. Example log message: `Cannot insert record due to duplicate key violation.`

Critical = 5

For failures that require immediate attention. Examples: data loss scenarios, out of disk space.

You can use the log level to control how much log output is written to a particular storage medium or display window. For example, in production you might want all logs of `Information` level and higher to go to a high-volume data store, and all logs of `Warning` level and higher to go to a high-value data store. During development you might normally direct only logs of `Warning` or higher severity to the console, but add `Debug` level when you need to investigate a problem. The Log filtering section later in this article explains how to control which log levels a provider handles.

The ASP.NET Core framework writes `Debug` logs for framework events. Here's an example of what you see from the console provider if you run the sample application with the minimum log level set to `Debug` and go to URL `http://localhost:5000/api/todo/0`:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
dbug: Microsoft.AspNetCore.StaticFiles.StaticFileMiddleware[4]
      The request path /api/todo/0 does not match a supported file type
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 '6cada879-f7a8-4152-b244-
7b41831791cc' 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 '529c0e82-aea6-466c-bbe2-
e77ded858853' 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 HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 198.8402ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 550.6837ms 404
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HKV8M5ARIH5P" completed keep alive response.
```

Log event ID

Each time you write a log, you can specify an *event ID*. The sample app does this by using a locally-defined `LoggingEvents` class:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GENERATE_ITEMS = 1000;
    public const int LIST_ITEMS = 1001;
    public const int GET_ITEM = 1002;
    public const int INSERT_ITEM = 1003;
    public const int UPDATE_ITEM = 1004;
    public const int DELETE_ITEM = 1005;

    public const int GET_ITEM_NOTFOUND = 4000;
    public const int UPDATE_ITEM_NOTFOUND = 4001;
}
```

An event ID is an integer value that you can use to associate a set of logged events with one another. For instance, a log for adding an item to a shopping cart could be event ID 1000 and a log for completing a purchase could be event ID 1001.

In logging output, the event ID may be stored in a field or included in the text message, depending on the provider. The Debug provider doesn't show event IDs, but the console provider shows them in brackets after the category:

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

Log message format string

Each time you write a log, you provide a text message. The message string can contain named placeholders into which argument values are placed, as in the following example:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {ID}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({ID}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

The order of placeholders, not their names, determines which parameters are used for them. For example, if you have the following code:

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

The resulting log message would look like this:

```
Parameter values: parm1, parm2
```

The logging framework does message formatting in this way to make it possible for logging providers to implement semantic logging, also known as structured logging. Because the arguments themselves are passed to the logging system, not just the formatted message string, logging providers can store the parameter values as fields in addition to the message string. For example, if you are directing your log output to Azure Table Storage, and your logger method call looks like this:

```
_logger.LogInformation("Getting item {ID} at {RequestTime}", id, DateTime.Now);
```

Each Azure Table entity could have `ID` and `RequestTime` properties, which would simplify queries on log data. You could find all logs within a particular `RequestTime` range, without having to parse the time out of the text message.

Logging exceptions

The logger methods have overloads that let you pass in an exception, as in the following example:

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, ex, "GetById({ID}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

Different providers handle the exception information in different ways. Here's an example of Debug provider output from the code shown above.

```
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
```

Log filtering

Some logging providers let you specify when logs should be written to a storage medium or ignored based on log level and category.

The `AddConsole` and `AddDebug` extension methods provide overloads that let you pass in filtering criteria. The following sample code causes the console provider to ignore logs below `Warning` level, while the Debug provider ignores logs that the framework creates.

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(LogLevel.Warning)
        .AddDebug((category, logLevel) => (category.Contains("TodoApi") && logLevel >= LogLevel.Trace));
```

The `AddEventLog` method has an overload that takes an `EventLogSettings` instance, which may contain a filtering function in its `Filter` property. The TraceSource provider does not provide any of those overloads, since its logging level and other parameters are based on the `SourceSwitch` and `TraceListener` it uses.

You can set filtering rules for all providers that are registered with an `ILoggerFactory` instance by using the `WithFilter` extension method. The example below limits framework logs (category begins with "Microsoft" or "System") to warnings while letting the app log at debug level.

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .WithFilter(new FilterLoggerSettings
        {
            { "Microsoft", LogLevel.Warning },
            { "System", LogLevel.Warning },
            { "ToDoApi", LogLevel.Debug }
        })
        .AddConsole()
        .AddDebug();
```

If you want to use filtering to prevent all logs from being written for a particular category, you can specify `LogLevel.None` as the minimum log level for that category. The integer value of `LogLevel.None` is 6, which is higher than `LogLevel.Critical` (5).

The `WithFilter` extension method is provided by the Microsoft.Extensions.Logging.Filter NuGet package. The method returns a new `ILoggerFactory` instance that will filter the log messages passed to all logger providers registered with it. It does not affect any other `ILoggerFactory` instances, including the original `ILoggerFactory` instance.

Log scopes

You can group a set of logical operations within a *scope* in order to attach the same data to each log that is created as part of that set. For example, you might want every log created as part of processing a transaction to include the transaction ID.

A scope is an `IDisposable` type that is returned by the `ILogger.BeginScope<TState>` method and lasts until it is disposed. You use a scope by wrapping your logger calls in a `using` block, as shown here:

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GET_ITEM, "Getting item {ID}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GET_ITEM_NOTFOUND, "GetById({ID}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

The following code enables scopes for the console provider:

```
public void Configure(IApplicationBuilder app,
    IHostingEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory
        .AddConsole(includeScopes: true)
        .AddDebug();
```

Each log message includes the scoped information:

```
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
```

Built-in logging providers

ASP.NET Core ships the following providers:

Console
Debug
EventSource
EventLog
TraceSource
Azure App Service

The console provider

The Microsoft.Extensions.Logging.Console provider package sends log output to the console.

```
loggerFactory.AddConsole()
```

AddConsole overloads let you pass in an a minimum log level, a filter function, and a boolean that indicates whether scopes are supported. Another option is to pass in an `IConfiguration` object, which can specify scopes support and logging levels.

If you are considering the console provider for use in production, be aware that it has a significant impact on performance.

When you create a new project in Visual Studio, the `AddConsole` method looks like this:

```
loggerFactory.AddConsole(Configuration.GetSection("Logging"));
```

This code refers to the `Logging` section of the *appSettings.json* file:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

The settings shown limit framework logs to warnings while allowing the app to log at debug level, as explained in the Log filtering section. For more information, see Configuration.

The Debug provider

The Microsoft.Extensions.Logging.Debug provider package writes log output by using the System.Diagnostics.Debug class.

```
loggerFactory.AddDebug()
```

AddDebug overloads let you pass in a minimum log level or a filter function.

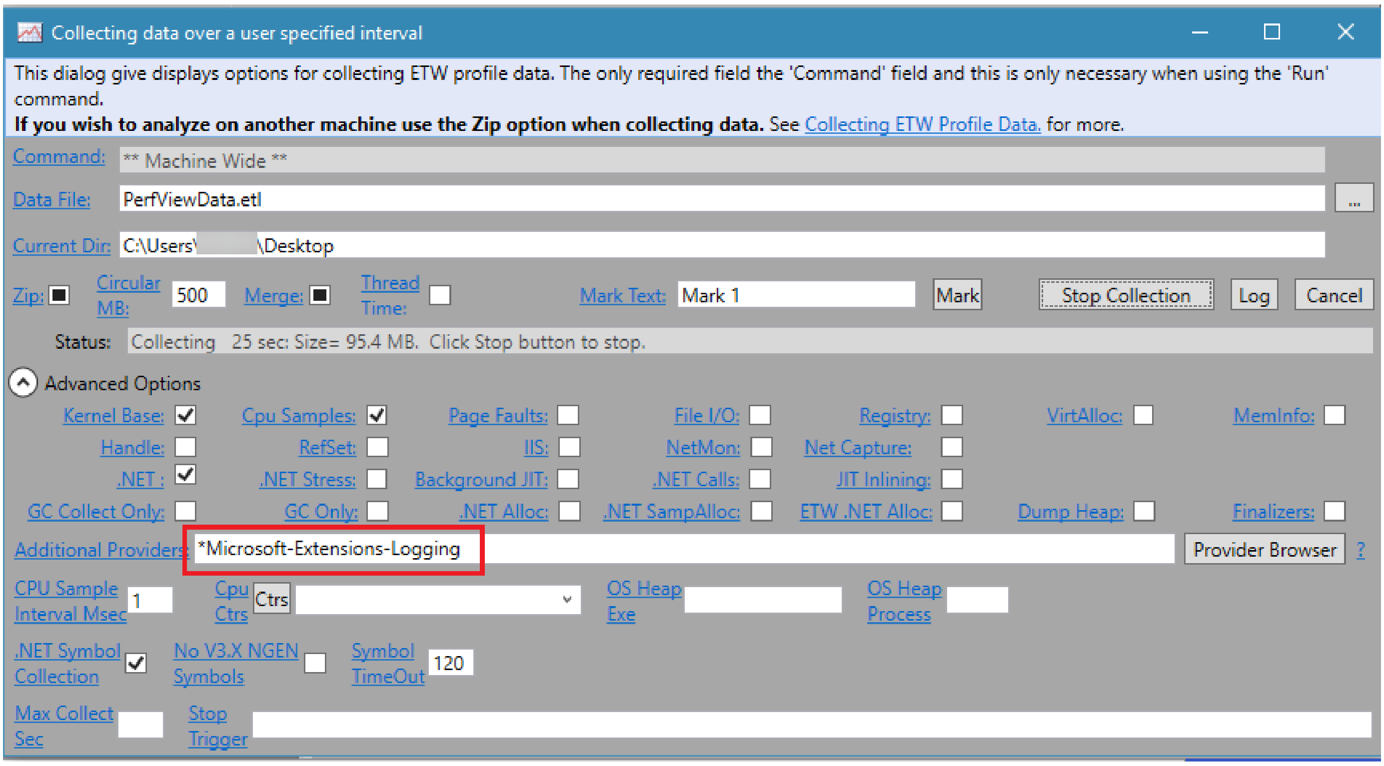
The EventSource provider

For apps that target ASP.NET Core 1.1.0 or higher, the Microsoft.Extensions.Logging.EventSource provider package can implement event tracing. On Windows, it uses ETW. The provider is cross-platform, but there are no event collection and display tools yet for Linux or macOS.

```
loggerFactory.AddEventSourceLogger()
```

The best way to collect and view logs is to use the PerfView utility. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. (Don't miss the asterisk at the start of the string.)



Capturing events on Nano Server requires some additional setup:

Connect PowerShell remoting to the Nano Server:

```
Enter-PSSession [name]
```

Create an ETW session:

```
New-EtwTraceSession -Name "MyAppTrace" -LocalFilePath C:\trace.etl
```

Add ETW providers for CLR, ASP.NET Core, and others as needed. The ASP.NET Core provider GUID is `3ac73b97-af73-50e9-0822-5da4367920d0`.

```
Add-EtwTraceProvider -Guid "{e13c0d23-ccbc-4e12-931b-d9cc2eee27e4}" -SessionName MyAppTrace
Add-EtwTraceProvider -Guid "{3ac73b97-af73-50e9-0822-5da4367920d0}" -SessionName MyAppTrace
```

Run the site and do whatever actions you want tracing information for.

Stop the tracing session when you're finished:

```
Stop-EtwTraceSession -Name "MyAppTrace"
```

The resulting *C:\trace.etl* file can be analyzed with PerfView as on other editions of Windows.

The Windows EventLog provider

The Microsoft.Extensions.Logging.EventLog provider package sends log output to the Windows Event Log.

```
loggerFactory.AddEventLog()
```

AddEventLog overloads let you pass in `EventLogSettings` or a minimum log level.

The TraceSource provider

The Microsoft.Extensions.Logging.TraceSource provider package uses the System.Diagnostics.TraceSource libraries and providers.

```
loggerFactory.AddTraceSource(sourceSwitchName);
```

AddTraceSource overloads let you pass in a source switch and a trace listener.

To use this provider, an application has to run on the .NET Framework (rather than .NET Core). The provider lets you route messages to a variety of listeners, such as the TextWriterTraceListener used in the sample application.

The following example configures a `TraceSource` provider that logs `Warning` and higher messages to the console window.

```
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));
```

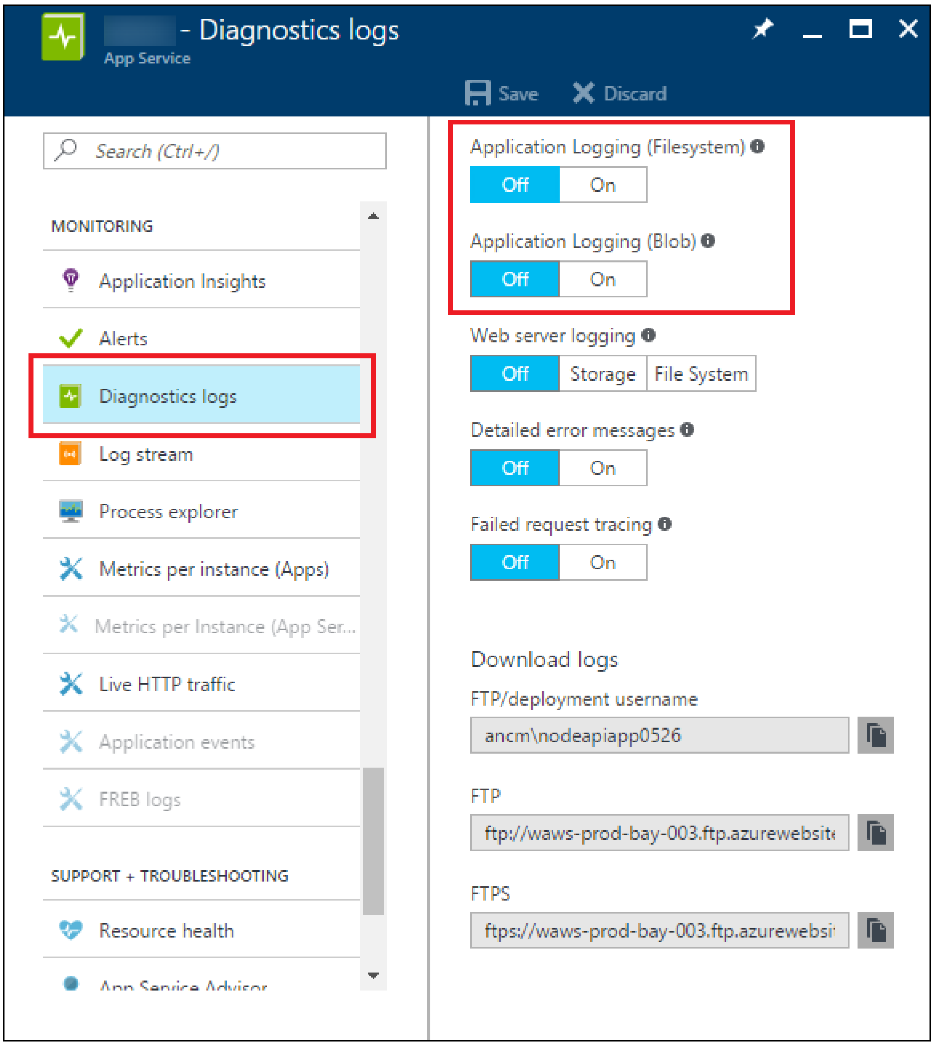
The Azure App Service provider

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account. The provider is available only for apps that target ASP.NET Core 1.1.0 or higher.

```
loggerFactory.AddAzureWebAppDiagnostics();
```

An `AddAzureWebAppDiagnostics` overload lets you pass in AzureAppServicesDiagnosticsSettings, with which you can override default settings such as the logging output template, blob name, and file size limit. (*Output template* is a message format string that is applied to all logs, on top of the one that you provide when you call an `ILogger` method.)

When you deploy to an App Service app, your application honors the settings in the Diagnostic Logs section of the **App Service** blade of the Azure portal. When you change those settings, the changes take effect immediately without requiring that you restart the app or redeploy code to it.



The default location for log files is in the *D:\home\LogFiles\Application* folder, and the default file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB and the default maximum number of files retained is 2. The default blob name is *{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt*. For more information about default behavior, see AzureAppServicesDiagnosticsSettings.

The provider only works when your project runs in the Azure environment. It has no effect when you run locally -- it does not write to local files or local development storage for blobs.

Note

If you don't need to change the provider default settings, there's an alternative way to set up App Service logging in your application. Install Microsoft.AspNetCore.AzureAppServicesIntegration (which includes the logging package as a dependency), and call its extension method on `WebHostBuilder` in your `Main` method.

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseAzureAppServices()
    .UseStartup<Startup>()
    .Build();
```

Behind the scenes, `UseAzureAppServices` calls `UseIISIntegration` and the logging provider extension method `AddAzureWebAppDiagnostics`.

Third-party logging providers

Here are some third-party logging frameworks that work with ASP.NET Core:

elmah.io - provider for the Elmah.Io service

Loggr - provider for the Loggr service

NLog - provider for the NLog library

Serilog - provider for the Serilog library

Some third-party frameworks can do semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers: add a NuGet package to your project and call an extension method on `ILoggerFactory`. For more information, see each framework's documentation.

You can create your own custom providers as well, to support other logging frameworks or your own logging requirements.

File Providers in ASP.NET Core

By Steve Smith

ASP.NET Core abstracts file system access through the use of File Providers.

View or download sample code

File Provider abstractions

File Providers are an abstraction over file systems. The main interface is `IFileProvider`. `IFileProvider` exposes methods to get file information (`IFileInfo`), directory information (`IDirectoryContents`), and to set up change notifications (using an `IChangeToken`).

`IFileInfo` provides methods and properties about individual files or directories. It has two boolean properties, `Exists` and `IsDirectory`, as well as properties describing the file's `Name`, `Length` (in bytes), and `LastModified` date. You can read from the file using its `CreateReadStream` method.

File Provider implementations

Three implementations of `IFileProvider` are available: Physical, Embedded, and Composite. The physical provider is used to access the actual system's files. The embedded provider is used to access files embedded in assemblies. The composite provider is used to provide combined access to files and directories from one or more other providers.

PhysicalFileProvider

The `PhysicalFileProvider` provides access to the physical file system. It wraps the `System.IO.File` type (for the physical provider), scoping all paths to a directory and its children. This scoping limits access to a certain directory and its children, preventing access to the file system outside of this boundary. When instantiating this provider, you must provide it with a directory path, which serves as the base path for all requests made to this provider (and which restricts access outside of this path). In an ASP.NET Core app, you can instantiate a `PhysicalFileProvider` provider directly, or you can request an `IFileProvider` in a Controller or service's constructor through dependency injection. The latter approach will typically yield a more flexible and testable solution.

The sample below shows how to create a `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
```

You can iterate through its directory contents or get a specific file's information by providing a subpath.

To request a provider from a controller, specify it in the controller's constructor and assign it to a local field. Use the local instance from your action methods:

```
public class HomeController : Controller
{
    private readonly IFileProvider _fileProvider;

    public HomeController(IFileProvider fileProvider)
    {
        _fileProvider = fileProvider;
    }

    public IActionResult Index()
    {
        var contents = _fileProvider.GetDirectoryContents("");
        return View(contents);
    }
```

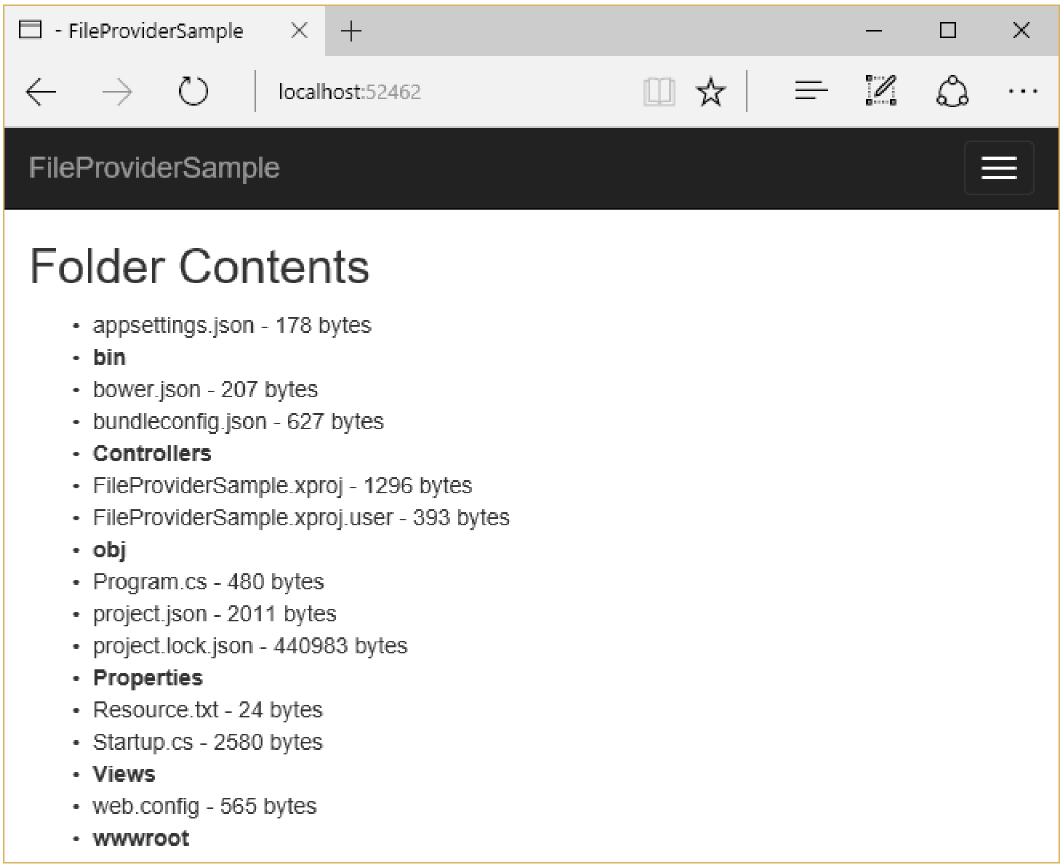
Then, create the provider in the app's `Startup` class:

```
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);
        }
```

In the *Index.cshtml* view, iterate through the `IDirectoryContents` provided:

```
@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>
```

The result:



EmbeddedFileProvider

The `EmbeddedFileProvider` is used to access files embedded in assemblies. In .NET Core, you embed files in an assembly with the `<EmbeddedResource>` element in the *.csproj* file:

```
<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>
```

You can use globbing patterns when specifying files to embed in the assembly. These patterns can be used to match one or more files.

Note

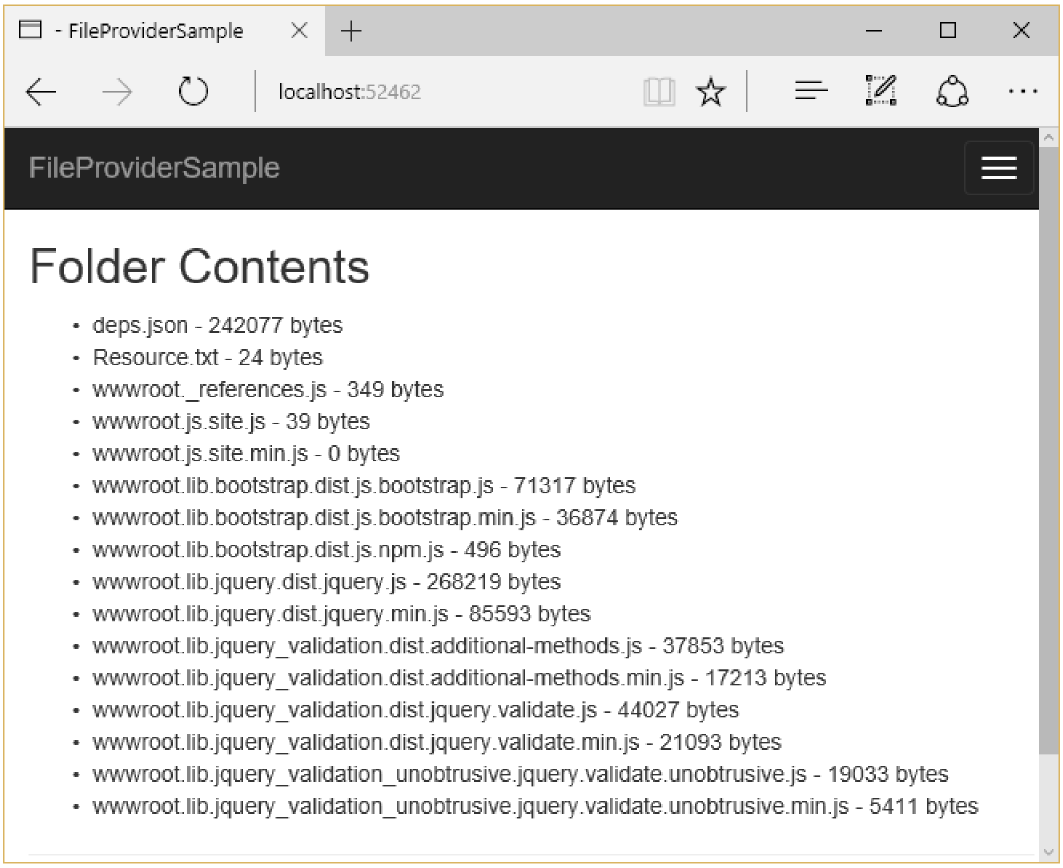
It's unlikely you would ever want to actually embed every .js file in your project in its assembly; the above sample is for demo purposes only.

When creating an `EmbeddedFileProvider`, pass the assembly it will read to its constructor.

```
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
```

The snippet above demonstrates how to create an `EmbeddedFileProvider` with access to the currently executing assembly.

Updating the sample app to use an `EmbeddedFileProvider` results in the following output:



Note

Embedded resources do not expose directories. Rather, the path to the resource (via its namespace) is embedded in its filename using `.` separators.

Tip

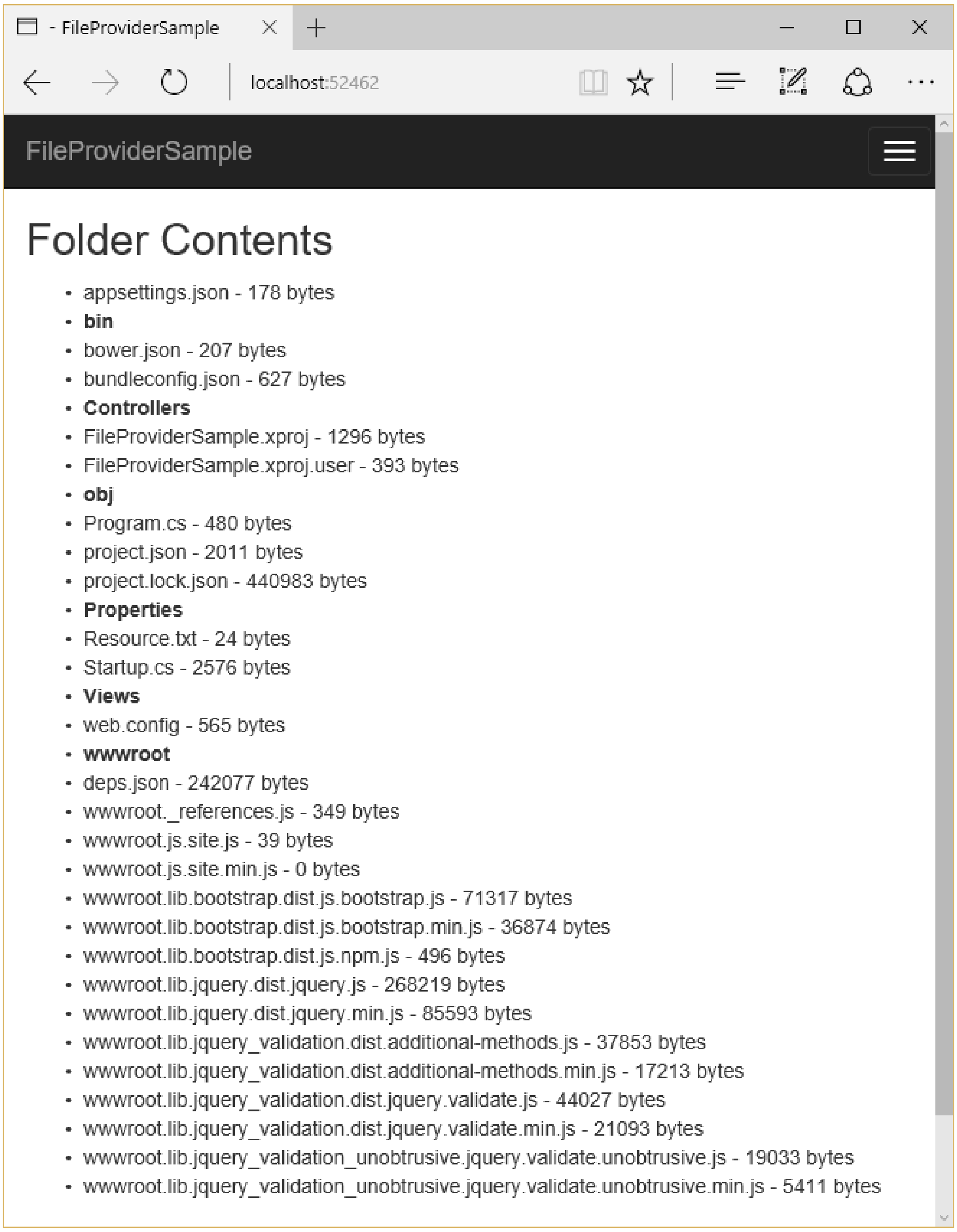
The `EmbeddedFileProvider` constructor accepts an optional `baseNamespace` parameter. Specifying this will scope calls to `GetDirectoryContents` to those resources under the provided namespace.

CompositeFileProvider

The `CompositeFileProvider` combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, you pass one or more `IFileProvider` instances to its constructor:

```
var physicalProvider = _hostingEnvironment.ContentRootFileProvider;
var embeddedProvider = new EmbeddedFileProvider(Assembly.GetEntryAssembly());
var compositeProvider = new CompositeFileProvider(physicalProvider, embeddedProvider);
```

Updating the sample app to use a `CompositeFileProvider` that includes both the physical and embedded providers configured previously, results in the following output:



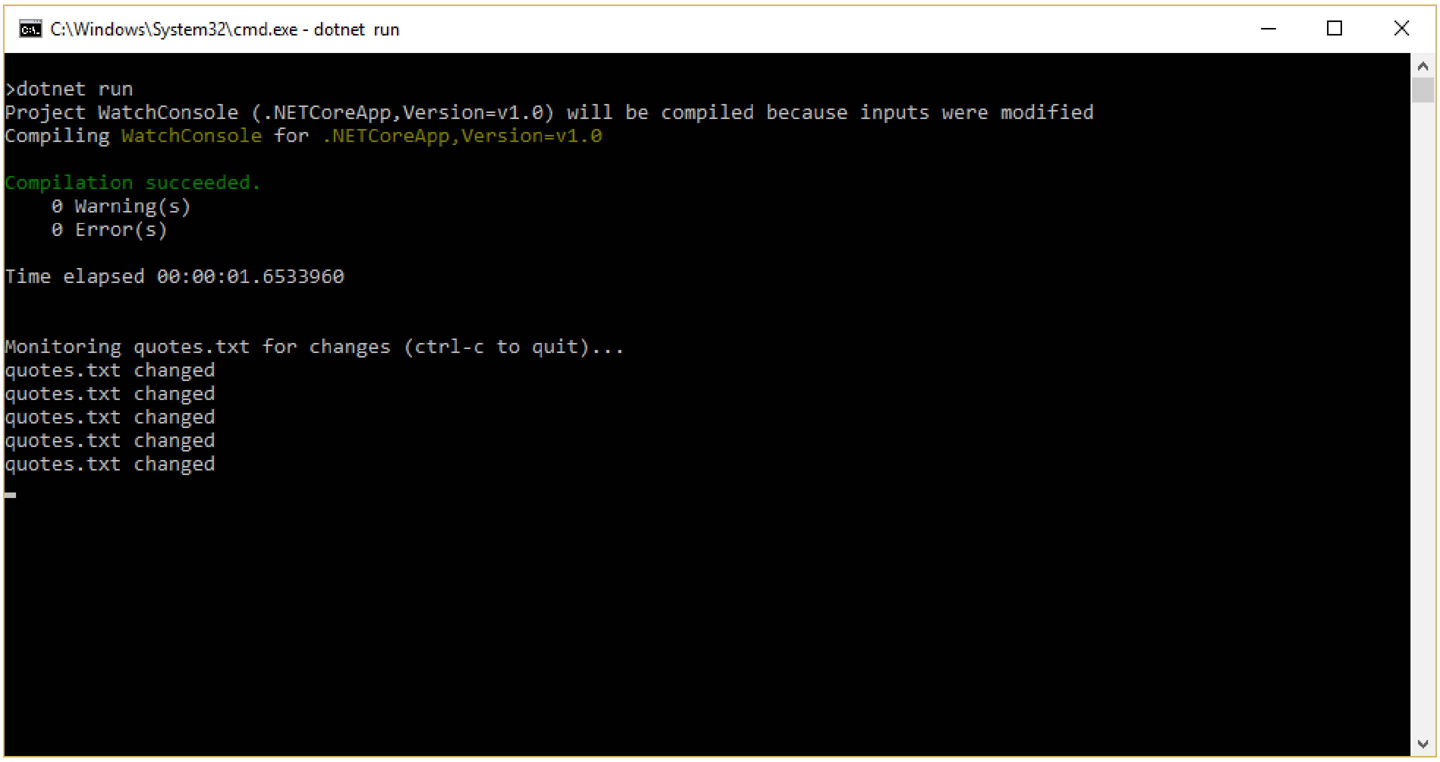
Watching for changes

The `IFileProvider` `Watch` method provides a way to watch one or more files or directories for changes. This method accepts a path string, which can use globbing patterns to specify multiple files, and returns an `IChangeToken`. This token exposes a `HasChanged` property that can be inspected, and a `RegisterChangeCallback` method that is called when changes are detected to the specified path string. Note that each change token only calls its associated callback in response to a single change. To enable constant monitoring, you can use a `TaskCompletionSource` as shown below, or re-create `IChangeToken` instances in response to changes.

In this article's sample, a console application is configured to display a message whenever a text file is modified:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.Extensions.FileProviders;
using Microsoft.Extensions.Primitives;

namespace WatchConsole
{
    public class Program
    {
        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");
        }
    }
}
```

The result, after saving the file several times:



Note

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLINGFILEWATCHER` environment variable to `1` or `true` to poll the file system for changes every 4 seconds.

Globbing patterns

File system paths use wildcard patterns called *globbing patterns*. These simple patterns can be used to specify groups of files. The two wildcard characters are `*` and `**`.

`*`

Matches anything at the current folder level, or any filename, or any file extension. Matches are terminated by `/` and `.` characters in the file path.

`**`

Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

Globbing pattern examples

`directory/file.txt`

Matches a specific file in a specific directory.

`directory/*.txt`

Matches all files with `.txt` extension in a specific directory.

`directory/*/bower.json`

Matches all `bower.json` files in directories exactly one level below the `directory` directory.

`directory/**/*.txt`

Matches all files with `.txt` extension found anywhere under the `directory` directory.

File Provider usage in ASP.NET Core

Several parts of ASP.NET Core utilize file providers. `IHostingEnvironment` exposes the app's content root and web root as `IFileProvider` types. The static files middleware uses file providers to locate static files. Razor makes heavy use of `IFileProvider` in locating views. Dotnet's publish functionality uses file providers and globbing patterns to specify which files should be published.

Recommendations for use in apps

If your ASP.NET Core app requires file system access, you can request an instance of `IFileProvider` through dependency injection, and then use its methods to perform the access, as shown in this sample. This allows you to configure the provider once, when the app starts up, and reduces the number of implementation types your app instantiates.

Introduction to Dependency Injection in ASP.NET Core

By Steve Smith and Scott Addie

ASP.NET Core is designed from the ground up to support and leverage dependency injection. ASP.NET Core applications can leverage built-in framework services by having them injected into methods in the Startup class, and application services can be configured for injection as well. The default services container provided by ASP.NET Core provides a minimal feature set and is not intended to replace other containers.

View or download sample code

What is Dependency Injection?

Dependency injection (DI) is a technique for achieving loose coupling between objects and their collaborators, or dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs in order to perform its actions are provided to the class in some fashion. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. This approach is known as "constructor injection".

When classes are designed with DI in mind, they are more loosely coupled because they do not have direct, hard-coded dependencies on their collaborators. This follows the Dependency Inversion Principle, which states that "*high level modules should not depend on low level modules; both should depend on abstractions.*" Instead of referencing specific implementations, classes request abstractions (typically `interfaces`) which are provided to them when the class is constructed. Extracting dependencies into interfaces and providing implementations of these interfaces as parameters is also an example of the Strategy design pattern.

When a system is designed to use DI, with many classes requesting their dependencies via their constructor (or properties), it's helpful to have a class dedicated to creating these classes with their associated dependencies. These classes are referred to as *containers*, or more specifically, Inversion of Control (IoC) containers or Dependency Injection (DI) containers. A container is essentially a factory that is responsible for providing instances of types that are requested from it. If a given type has declared that it has dependencies, and the container has been configured to provide the dependency types, it will create the dependencies as part of creating the requested instance. In this way, complex dependency graphs can be provided to classes without the need for any hard-coded object construction. In addition to creating objects with their dependencies, containers typically manage object lifetimes within the application.

ASP.NET Core includes a simple built-in container (represented by the `IServiceProvider` interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET's container refers to the types it manages as *services*. Throughout the rest of this article, *services* will refer to types that are managed by ASP.NET Core's IoC container. You configure the built-in container's services in the `ConfigureServices` method in your application's `Startup` class.

▯ Note

Martin Fowler has written an extensive article on Inversion of Control Containers and the Dependency Injection Pattern. Microsoft Patterns and Practices also has a great description of Dependency Injection.

▯ Note

This article covers Dependency Injection as it applies to all ASP.NET applications. Dependency Injection within MVC controllers is covered in Dependency Injection and Controllers.

Constructor Injection Behavior

Constructor injection requires that the constructor in question be *public*. Otherwise, your app will throw an `InvalidOperationException`:

> A suitable constructor for type 'YourType' could not be located. Ensure the type is concrete and services are registered for all parameters of a public constructor.

Constructor injection requires that only one applicable constructor exist. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection. If more than one exists, your app will throw an `InvalidOperationException`:

> Multiple constructors accepting all given argument types have been found in type 'YourType'. There should only be one applicable constructor.

Constructors can accept arguments that are not provided by dependency injection, but these must support default values. For example:

```
// 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;
}
```

Using Framework-Provided Services

The `ConfigureServices` method in the `Startup` class is responsible for defining the services the application will use, including platform features like Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` has the following services defined (depending on how the host was configured):

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.IHostingEnvironment	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.Logging.ILogger<T>	Singleton
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
Microsoft.Extensions.Options.IOptions<T>	Singleton
System.Diagnostics.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<T>	Transient

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IApplicationLifetime	Singleton

Below is an example of how to add additional services to the container using a number of extension methods like `AddDbContext`, `AddIdentity`, and `AddMvc`.

```
// 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>();
}
```

The features and middleware provided by ASP.NET, such as MVC, follow a convention of using a single Add*ServiceName* extension method to register all of the services required by that feature.

湴 Tip

You can request certain framework-provided services within `Startup` methods through their parameter lists - see Application Startup for more details.

Registering Your Own Services

You can register your own application services as follows. The first generic type represents the type (typically an interface) that will be requested from the container. The second generic type represents the concrete type that will be instantiated by the container and used to fulfill such requests.

```
services.AddTransient<IEmailSender, AuthMessageSender>();
services.AddTransient<ISmsSender, AuthMessageSender>();
```

▯ Note

Each `services.Add<ServiceName>` extension method adds (and potentially configures) services. For example, `services.AddMvc()` adds the services MVC requires. It's recommended that you follow this convention, placing extension methods in the `Microsoft.Extensions.DependencyInjection` namespace, to encapsulate groups of service registrations.

The `AddTransient` method is used to map abstract types to concrete services that are instantiated separately for every object that requires it. This is known as the service's *lifetime*, and additional lifetime options are described below. It is important to choose an appropriate lifetime for each of the services you register. Should a new instance of the service be provided to each class that requests it? Should one instance be used throughout a given web request? Or should a single instance be used for the lifetime of the application?

In the sample for this article, there is a simple controller that displays character names, called `CharactersController`. Its `Index` method displays the current list of characters that have been stored in the application, and initializes the collection with a handful of characters if none exist. Note that although this application uses Entity Framework Core and the `ApplicationDbContext` class for its persistence, none of that is apparent in the controller. Instead, the specific data access mechanism has been abstracted behind an interface, `ICharacterRepository`, which follows the repository pattern. An instance of `ICharacterRepository` is requested via the constructor and assigned to a private field, which is then used to access characters as necessary.

```
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"));
        }
    }
}
```

The `ICharacterRepository` defines the two methods the controller needs to work with `Character` instances.

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

namespace DependencyInjectionSample.Interfaces
{
    public interface ICharacterRepository
    {
        IEnumerable<Character> ListAll();
        void Add(Character character);
    }
}
```

This interface is in turn implemented by a concrete type, `CharacterRepository`, that is used at runtime.

Note

The way DI is used with the `CharacterRepository` class is a general model you can follow for all of your application services, not just in "repositories" or data access classes.

```
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();
        }
    }
}
```

Note that `CharacterRepository` requests an `ApplicationDbContext` in its constructor. It is not unusual for dependency injection to be used in a chained fashion like this, with each requested dependency in turn requesting its own dependencies. The container is responsible for resolving all of the dependencies in the graph and returning the fully resolved service.

Note

Creating the requested object, and all of the objects it requires, and all of the objects those require, is sometimes referred to as an *object graph*. Likewise, the collective set of dependencies that must be resolved is typically referred to as a *dependency tree* or *dependency graph*.

In this case, both `ICharacterRepository` and in turn `ApplicationDbContext` must be registered with the services container in `ConfigureServices` in `Startup`. `ApplicationDbContext` is configured with the call to the extension method `AddDbContext<T>`. The following code shows the registration of the `CharacterRepository` type.

```
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>();
}
```

Entity Framework contexts should be added to the services container using the `Scoped` lifetime. This is taken care of automatically if you use the helper methods as shown above. Repositories that will make use of Entity Framework should use the same lifetime.

▯ Warning

The main danger to be wary of is resolving a `Scoped` service from a singleton. It's likely in such a case that the service will have incorrect state when processing subsequent requests.

Services that have dependencies should register them in the container. If a service's constructor requires a primitive, such as a `string`, this can be injected by using the options pattern and configuration.

Service Lifetimes and Registration Options

ASP.NET services can be configured with the following lifetimes:

Transient

Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

Scoped

Scoped lifetime services are created once per request.

Singleton

Singleton lifetime services are created the first time they are requested (or when `ConfigureServices` is run if you specify an instance there) and then every subsequent request will use the same instance. If your application requires singleton behavior, allowing the services container to manage the service's lifetime is recommended instead of implementing the singleton design pattern and managing your object's lifetime in the class yourself.

Services can be registered with the container in several ways. We have already seen how to register a service implementation with a given type by specifying the concrete type to use. In addition, a factory can be specified, which will then be used to create the instance on demand. The third approach is to directly specify the instance of the type to use, in which case the container will never attempt to create an instance (nor will it dispose of the instance).

To demonstrate the difference between these lifetime and registration options, consider a simple interface that represents one or more tasks as an *operation* with a unique identifier, `OperationId`. Depending on how we configure the lifetime for this service, the container will provide either the same or different instances of the service to the requesting class. To make it clear which lifetime is being requested, we will create one type per lifetime option:

```
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
    {
    }
}
```

We implement these interfaces using a single class, `Operation`, that accepts a `Guid` in its constructor, or uses a new `Guid` if none is provided.

Next, in `ConfigureServices`, each type is added to the container according to its named lifetime:

```
    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>();
}
```

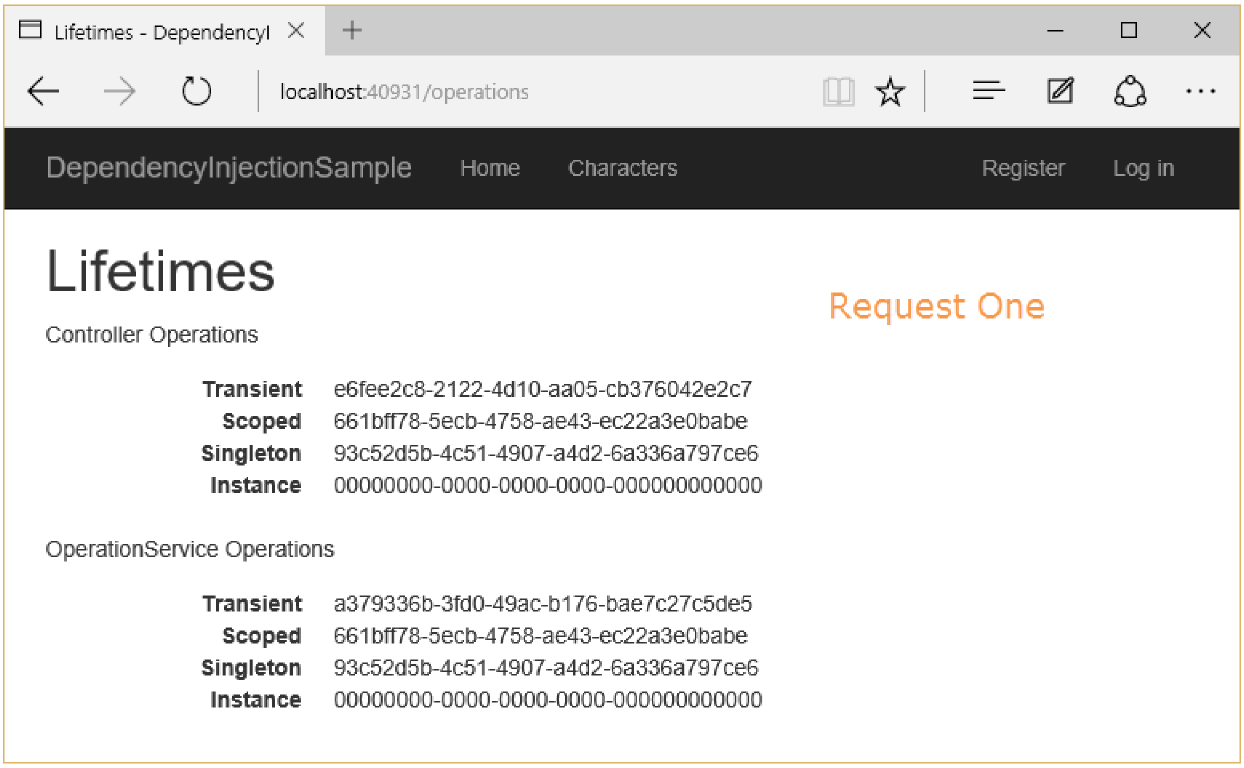
Note that the `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty` so it will be clear when this type is in use (its Guid will be all zeroes). We have also registered an `OperationService` that depends on each of the other `Operation` types, so that it will be clear within a request whether this service is getting the same instance as the controller, or a new one, for each operation type. All this service does is expose its dependencies as properties, so they can be displayed in the view.

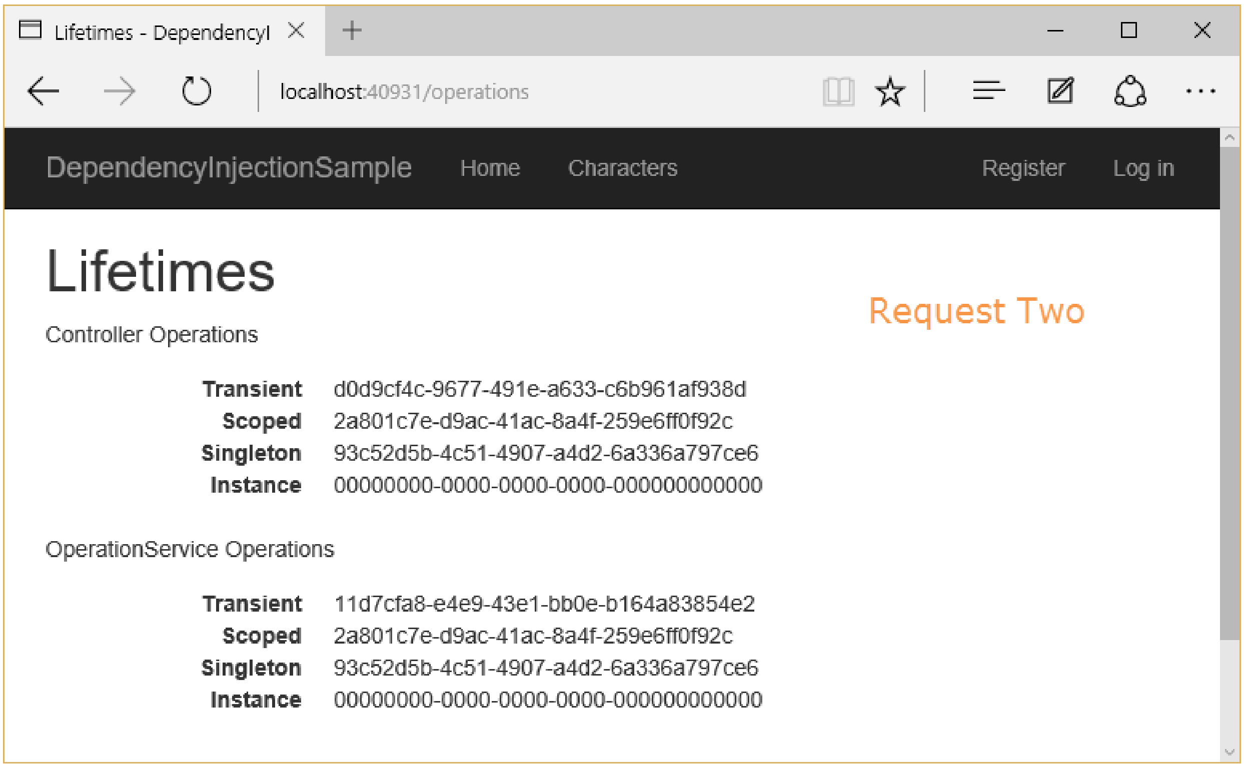
```
using DependencyInjectionSample.Interfaces;

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;
        }
    }
}
```

To demonstrate the object lifetimes within and between separate individual requests to the application, the sample includes an `OperationsController` that requests each kind of `IOperation` type as well as an `OperationService`. The `Index` action then displays all of the controller's and service's `OperationId` values.

```
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();
        }
    }
}
```

Now two separate requests are made to this controller action:





Observe which of the `OperationId` values vary within a request, and between requests.

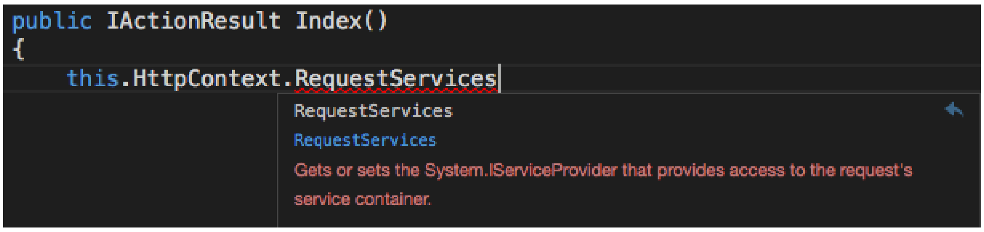
Transient objects are always different; a new instance is provided to every controller and every service.

Scoped objects are the same within a request, but different across different requests

Singleton objects are the same for every object and every request (regardless of whether an instance is provided in `ConfigureServices`)

Request Services

The services available within an ASP.NET request from `HttpContext` are exposed through the `RequestServices` collection.



Request Services represent the services you configure and request as part of your application. When your objects specify dependencies, these are satisfied by the types found in `RequestServices`, not `ApplicationServices`.

Generally, you shouldn't use these properties directly, preferring instead to request the types your classes you require via your class's constructor, and letting the framework inject these dependencies. This yields classes that are easier to test (see Testing) and are more loosely coupled.

Note

Prefer requesting dependencies as constructor parameters to accessing the `RequestServices` collection.

Designing Your Services For Dependency Injection

You should design your services to use dependency injection to get their collaborators. This means avoiding the use of stateful static method calls (which result in a code smell known as static cling) and the direct instantiation of dependent classes within your services. It may help to remember the phrase, New is Glue, when choosing whether to instantiate a type or to request it via dependency injection. By following the SOLID Principles of Object Oriented Design, your classes will naturally tend to be small, well-factored, and easily tested.

What if you find that your classes tend to have way too many dependencies being injected? This is generally a sign that your class is trying to do too much, and is probably violating SRP - the Single Responsibility Principle. See if you can refactor the class by moving some of its responsibilities into a new class. Keep in mind that your `Controller` classes should be focused on UI concerns, so business rules and data access implementation details should be kept in classes appropriate to these separate concerns.

With regards to data access specifically, you can inject the `DbContext` into your controllers (assuming you've added EF to the services container in `ConfigureServices`). Some developers prefer to use a repository interface to the database rather than injecting the `DbContext` directly. Using an interface to encapsulate the data access logic in one place can minimize how many places you will have to change when your database changes.

Disposing of services

The container will call `Dispose` for `IDisposable` types it creates. However, if you add an instance to the container yourself, it will not be disposed.

Example:

```
// Services implement IDisposable:
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}
public class Service3 : 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>();

    // container did not create instance so it will NOT dispose it
    services.AddSingleton<Service3>(new Service3());
    services.AddSingleton(new Service3());
}
```

 Note

In version 1.0, the container called dispose on *all* `IDisposable` objects, including those it did not create.

Replacing the default services container

The built-in services container is meant to serve the basic needs of the framework and most consumer applications built on it. However, developers can replace the built-in container with their preferred container. The `ConfigureServices` method typically returns `void`, but if its signature is changed to return `IServiceProvider`, a different container can be configured and returned. There are many IOC containers available for .NET. In this example, the Autofac package is used.

First, install the appropriate container package(s):

`Autofac`
`Autofac.Extensions.DependencyInjection`

Next, configure the container in `ConfigureServices` and return an `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);
}
```

 Note

When using a third-party DI container, you must change `ConfigureServices` so that it returns `IServiceProvider` instead of `void`.

Finally, configure Autofac as normal in `DefaultModule`:

```
public class DefaultModule : Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterType<CharacterRepository>().As<ICharacterRepository>();
    }
}
```

At runtime, Autofac will be used to resolve types and inject dependencies. Learn more about using Autofac and ASP.NET Core.

Thread safety

Singleton services need to be thread safe. If a singleton service has a dependency on a transient service, the transient service may also need to be thread safe depending how it’s used by the singleton.

Recommendations

When working with dependency injection, keep the following recommendations in mind:

DI is for objects that have complex dependencies. Controllers, services, adapters, and repositories are all examples of objects that might be added to DI.

Avoid storing data and configuration directly in DI. For example, a user's shopping cart shouldn't typically be added to the services container. Configuration should use the Options Model. Similarly, avoid "data holder" objects that only exist to allow access to some other object. It's better to request the actual item needed via DI, if possible.

Avoid static access to services.

Avoid service location in your application code.

Avoid static access to `HttpContext`.

Note

Like all sets of recommendations, you may encounter situations where ignoring one is required. We have found exceptions to be rare -- mostly very special cases within the framework itself.

Remember, dependency injection is an *alternative* to static/global object access patterns. You will not be able to realize the benefits of DI if you mix it with static object access.

Additional Resources

Application Startup

Testing

Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)

Container-Managed Application Design, Prelude: Where does the Container Belong?

Explicit Dependencies Principle

Inversion of Control Containers and the Dependency Injection Pattern (Fowler)

Working with multiple environments

By Steve Smith

ASP.NET Core introduces improved support for controlling application behavior across multiple environments, such as development, staging, and production. Environment variables are used to indicate which environment the application is running in, allowing the app to be configured appropriately.

View or download sample code

Development, Staging, Production

ASP.NET Core references a particular environment variable, `ASPNETCORE_ENVIRONMENT` to describe the environment the application is currently running in. This variable can be set to any value you like, but three values are used by convention: `Development`, `Staging`, and `Production`. You will find these values used in the samples and templates provided with ASP.NET Core.

The current environment setting can be detected programmatically from within your application. In addition, you can use the Environment tag helper to include certain sections in your view based on the current application environment.

Note

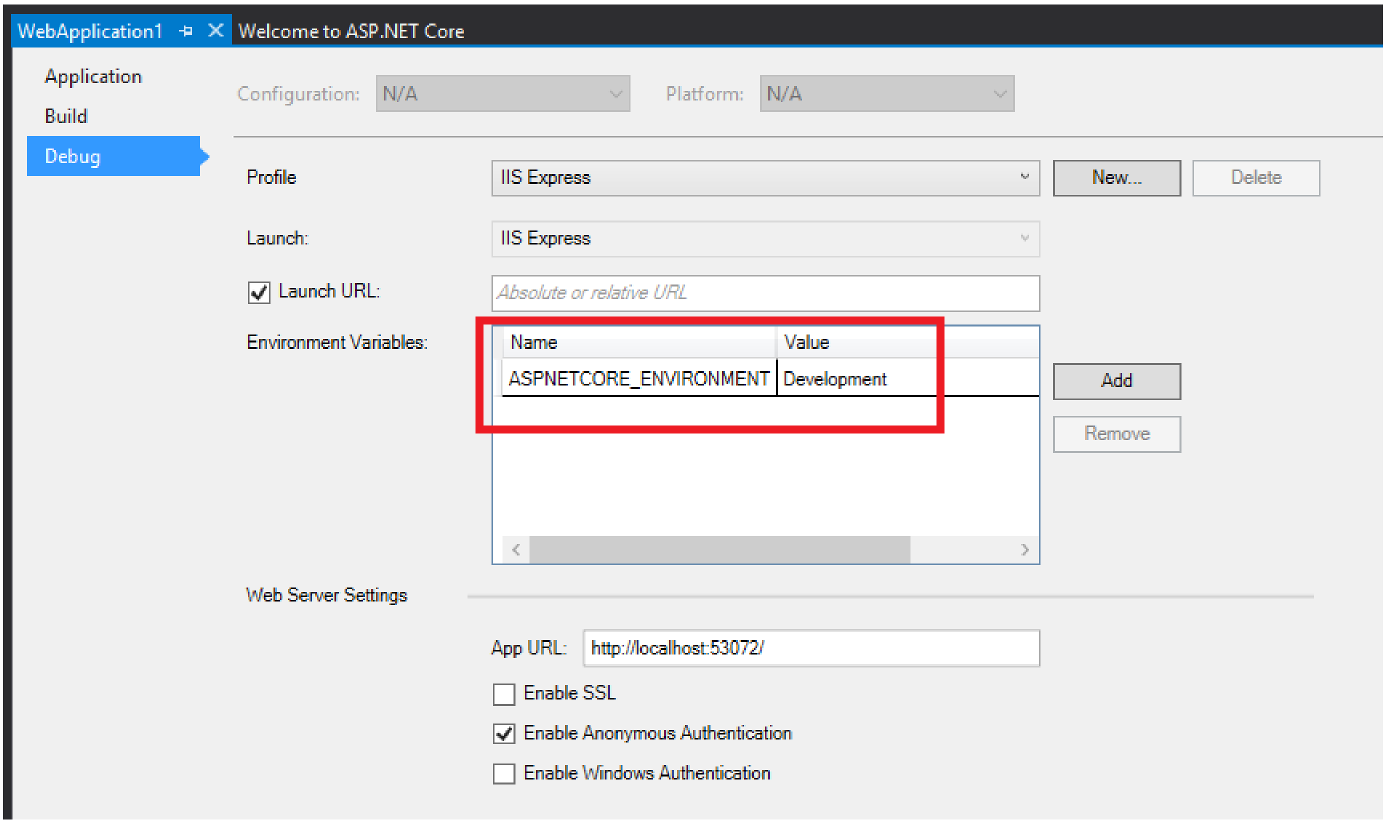
On Windows and macOS, the specified environment name is case insensitive. Whether you set the variable to `Development` or `development` or `DEVELOPMENT` the results will be the same. However, Linux is a **case sensitive** OS by default. Environment variables, file names and settings should assume case sensitivity for best practice.

Development

This should be the environment used when developing an application. It is typically used to enable features that you wouldn't want to be available when the app runs in production, such as the developer exception page.

If you're using Visual Studio, the environment can be configured in your project's debug profiles. Debug profiles specify the server to use when launching the application and any environment variables to be set. Your project can have multiple debug profiles that set environment variables differently. You manage these profiles by using the **Debug** tab of your web application project's **Properties** menu. The values you set in project properties are persisted in the *launchSettings.json* file, and you can also configure profiles by editing that file directly.

The profile for IIS Express is shown here:



Here is a `launchSettings.json` file that includes profiles for `Development` and `Staging`:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:40088/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express (Staging)": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    }
  }
}
```

Changes made to project profiles may not take effect until the web server used is restarted (in particular, Kestrel must be restarted before it will detect changes made to its environment).

 Warning

Environment variables stored in *launchSettings.json* are not secured in any way and will be part of the source code repository for your project, if you use one. **Never store credentials or other secret data in this file.** If you need a place to store such data, use the *Secret Manager* tool described in Safe storage of app secrets during development.

Staging

By convention, a `Staging` environment is a pre-production environment used for final testing before deployment to production. Ideally, its physical characteristics should mirror that of production, so that any issues that may arise in production occur first in the staging environment, where they can be addressed without impact to users.

Production

The `Production` environment is the environment in which the application runs when it is live and being used by end users. This environment should be configured to maximize security, performance, and application robustness. Some common settings that a production environment might have that would differ from development include:

Turn on caching

Ensure all client-side resources are bundled, minified, and potentially served from a CDN

Turn off diagnostic ErrorPages

Turn on friendly error pages

Enable production logging and monitoring (for example, Application Insights)

This is by no means meant to be a complete list. It's best to avoid scattering environment checks in many parts of your application. Instead, the recommended approach is to perform such checks within the application's `Startup` class(es) wherever possible

Setting the environment

The method for setting the environment depends on the operating system.

Windows

To set the `ASPNETCORE_ENVIRONMENT` for the current session, if the app is started using `dotnet run`, the following commands are used

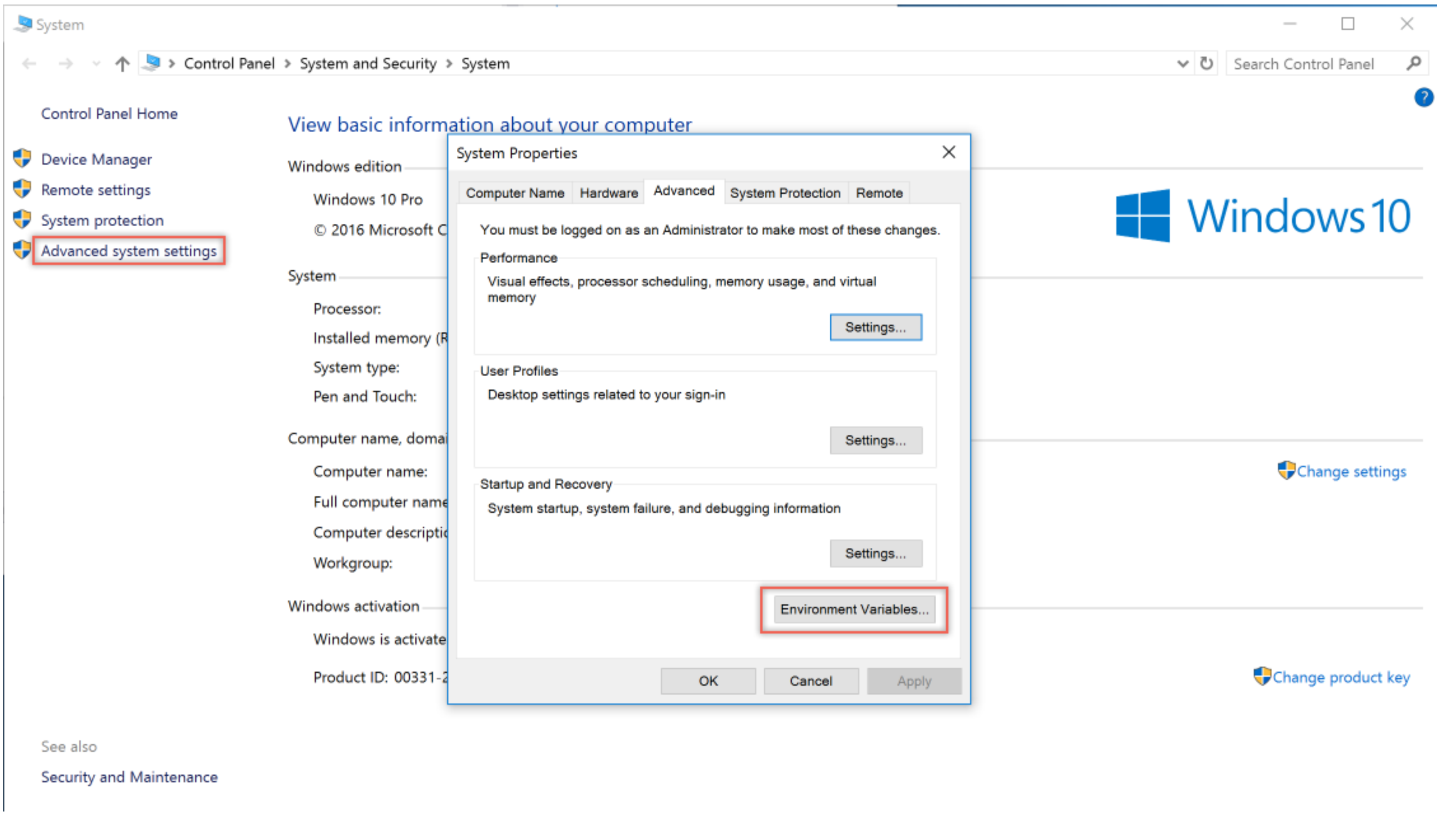
Command line

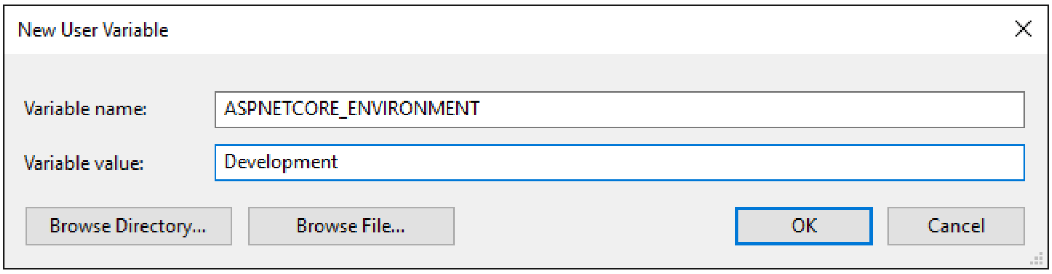
```
set ASPNETCORE_ENVIRONMENT=Development
```

PowerShell

```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

These commands take effect only for the current window. When the window is closed, the ASPNETCORE_ENVIRONMENT setting reverts to the default setting or machine value. In order to set the value globally on Windows open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value.





macOS

Setting the current environment for macOS can be done in-line when running the application;

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

or using `export` to set it prior to running the app.

```
export ASPNETCORE_ENVIRONMENT=Development
```

Machine level environment variables are set in the *.bashrc* or *.bash_profile* file. Edit the file using any text editor and add the following statment.

```
export ASPNETCORE_ENVIRONMENT=Development
```

Linux

For Linux distros, use the `export` command at the command line for session based variable settings and *bash_profile* file for machine level environment settings.

Determining the environment at runtime

The `IHostingEnvironment` service provides the core abstraction for working with environments. This service is provided by the ASP.NET hosting layer, and can be injected into your startup logic via Dependency Injection. The ASP.NET Core web site template in Visual Studio uses this approach to load environment-specific configuration files (if present) and to customize the app's error handling settings. In both cases, this behavior is achieved by referring to the currently specified environment by calling

`EnvironmentName` or `IsEnvironment` on the instance of `IHostingEnvironment` passed into the appropriate method.

Note

If you need to check whether the application is running in a particular environment, use `env.IsEnvironment("environmentname")` since it will correctly ignore case (instead of checking if `env.EnvironmentName == "Development"` for example).

For example, you can use the following code in your Configure method to setup environment specific error handling:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
```

If the app is running in a `Development` environment, then it enables the runtime support necessary to use the "BrowserLink" feature in Visual Studio, development-specific error pages (which typically should not be run in production) and special database error pages (which provide a way to apply migrations and should therefore only be used in development). Otherwise, if the app is not running in a development environment, a standard error handling page is configured to be displayed in response to any unhandled exceptions.

You may need to determine which content to send to the client at runtime, depending on the current environment. For example, in a development environment you generally serve non-minimized scripts and style sheets, which makes debugging easier. Production and test environments should serve the minified versions and generally from a CDN. You can do this using the Environment tag helper. The Environment tag helper will only render its contents if the current environment matches one of the environments specified using the `names` attribute.

```
<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>
```

To get started with using tag helpers in your application see Introduction to Tag Helpers.

Startup conventions

ASP.NET Core supports a convention-based approach to configuring an application's startup based on the current environment. You can also programmatically control how your application behaves according to which environment it is in, allowing you to create and manage your own conventions.

When an ASP.NET Core application starts, the `Startup` class is used to bootstrap the application, load its configuration settings, etc. (learn more about ASP.NET startup). However, if a class exists named `Startup{EnvironmentName}` (for example `StartupDevelopment`), and the `ASPNETCORE_ENVIRONMENT` environment variable matches that name, then that `Startup` class is used instead. Thus, you could configure `Startup` for development, but have a separate `StartupProduction` that would be used when the app is run in production. Or vice versa.

Note

Calling `WebHostBuilder.UseStartup<TStartup>()` overrides configuration sections.

In addition to using an entirely separate `Startup` class based on the current environment, you can also make adjustments to how the application is configured within a `Startup` class. The `Configure()` and `ConfigureServices()` methods support environment-specific versions similar to the `Startup` class itself, of the form `Configure{EnvironmentName}()` and `Configure{EnvironmentName}Services()`. If you define a method `ConfigureDevelopment()` it will be called instead of `Configure()` when the environment is set to development. Likewise, `ConfigureDevelopmentServices()` would be called instead of `ConfigureServices()` in the same environment.

Summary

ASP.NET Core provides a number of features and conventions that allow developers to easily control how their applications behave in different environments. When publishing an application from development to staging to production, environment variables set appropriately for the environment allow for optimization of the application for debugging, testing, or production use, as appropriate.

Additional Resources

Configuration

Introduction to Tag Helpers

Introduction to hosting in ASP.NET Core

By Steve Smith

To run an ASP.NET Core app, you need to configure and launch a host using `WebHostBuilder`.

What is a Host?

ASP.NET Core apps require a *host* in which to execute. A host must implement the `IWebHost` interface, which exposes collections of features and services, and a `Start` method. The host is typically created using an instance of a `WebHostBuilder`, which builds and returns a `WebHost` instance. The `WebHost` references the server that will handle requests. Learn more about servers.

What is the difference between a host and a server?

The host is responsible for application startup and lifetime management. The server is responsible for accepting HTTP requests. Part of the host's responsibility includes ensuring the application's services and the server are available and properly configured. You can think of the host as being a wrapper around the server. The host is configured to use a particular server; the server is unaware of its host.

Setting up a Host

You create a host using an instance of `WebHostBuilder`. This is typically done in your app's entry point: `public static void Main`, (which in the project templates is located in a *Program.cs* file). A typical *Program.cs*, shown below, demonstrates how to use a `WebHostBuilder` to build a host.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();

            host.Run();
        }
    }
}
```

The `WebHostBuilder` is responsible for creating the host that will bootstrap the server for the app. `WebHostBuilder` requires you provide a server that implements `IServer` (`UseKestrel` in the code above). `UseKestrel` specifies the Kestrel server will be used by the app.

The server's *content root* determines where it searches for content files, like MVC View files. The default content root is the folder from which the application is run.

 Note

Specifying `Directory.GetCurrentDirectory` as the content root will use the web project's root folder as the app's content root when the app is started from this folder (for example, calling `dotnet run` from the web project folder). This is the default used in Visual Studio and `dotnet new` templates.

If the app should work with IIS, the `UseIISIntegration` method should be called as part of building the host. Note that this does not configure a *server*, like `UseKestrel` does. To use IIS with ASP.NET Core, you must specify both `UseKestrel` and `UseIISIntegration`. Kestrel is designed to be run behind a proxy and should not be deployed directly facing the Internet. `UseIISIntegration` specifies IIS as the reverse proxy server.

Note

`UseKestrel` and `UseIISIntegration` are very different actions. IIS is only used as a reverse proxy. `UseKestrel` creates the web server and hosts the code. `UseIISIntegration` specifies IIS as the reverse proxy server. It also examines environment variables used by IIS/IISExpress and makes decisions like which dynamic port use, which headers to set, etc. However, it doesn't deal with or create an `IServer`.

A minimal implementation of configuring a host (and an ASP.NET Core app) would include just a server and configuration of the app's request pipeline:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .Configure(app =>
    {
        app.Run(async (context) => await context.Response.WriteAsync("Hi!"));
    })
    .Build();

host.Run();
```

Note

When setting up a host, you can provide `Configure` and `ConfigureServices` methods, instead of or in addition to specifying a `Startup` class (which must also define these methods - see Application Startup). Multiple calls to `ConfigureServices` will append to one another; calls to `Configure` or `UseStartup` will replace previous settings.

Configuring a Host

The `WebHostBuilder` provides methods for setting most of the available configuration values for the host, which can also be set directly using `UseSetting` and associated key. For example, to specify the application name:

```
new WebHostBuilder()
    .UseSetting("applicationName", "MyApp")
```

Host Configuration Values

Application Name `string`

Key: `applicationName`. This configuration setting specifies the value that will be returned from `IHostingEnvironment.ApplicationName`.

Capture Startup Errors `bool`

Key: `captureStartupErrors`. Defaults to `false`. When `false`, errors during startup result in the host exiting. When `true`, the host will capture any exceptions from the `Startup` class and attempt to start the server. It will display an error page (generic, or detailed, based on the Detailed Errors setting, below) for every request. Set using the `CaptureStartupErrors` method.

```
new WebHostBuilder()
    .CaptureStartupErrors(true)
```

Content Root `string`

Key: `contentRoot`. Defaults to the folder where the application assembly resides (for Kestrel; IIS will use the web project root by default). This setting determines where ASP.NET Core will begin searching for content files, such as MVC Views. Also used as the base path for the Web Root Setting. Set using the `UseContentRoot` method. Path must exist, or host will fail to start.

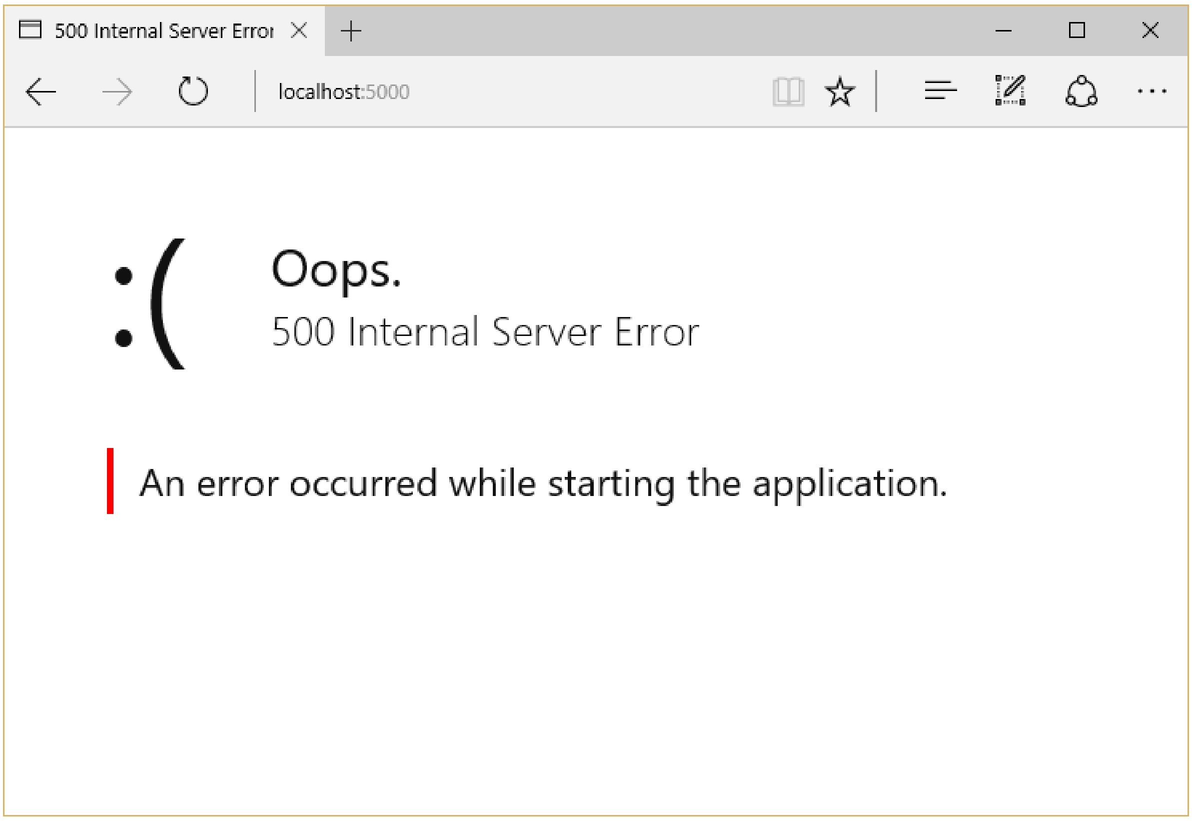
```
new WebHostBuilder()
    .UseContentRoot("c:\\mywebsite")
```

Detailed Errors `bool`

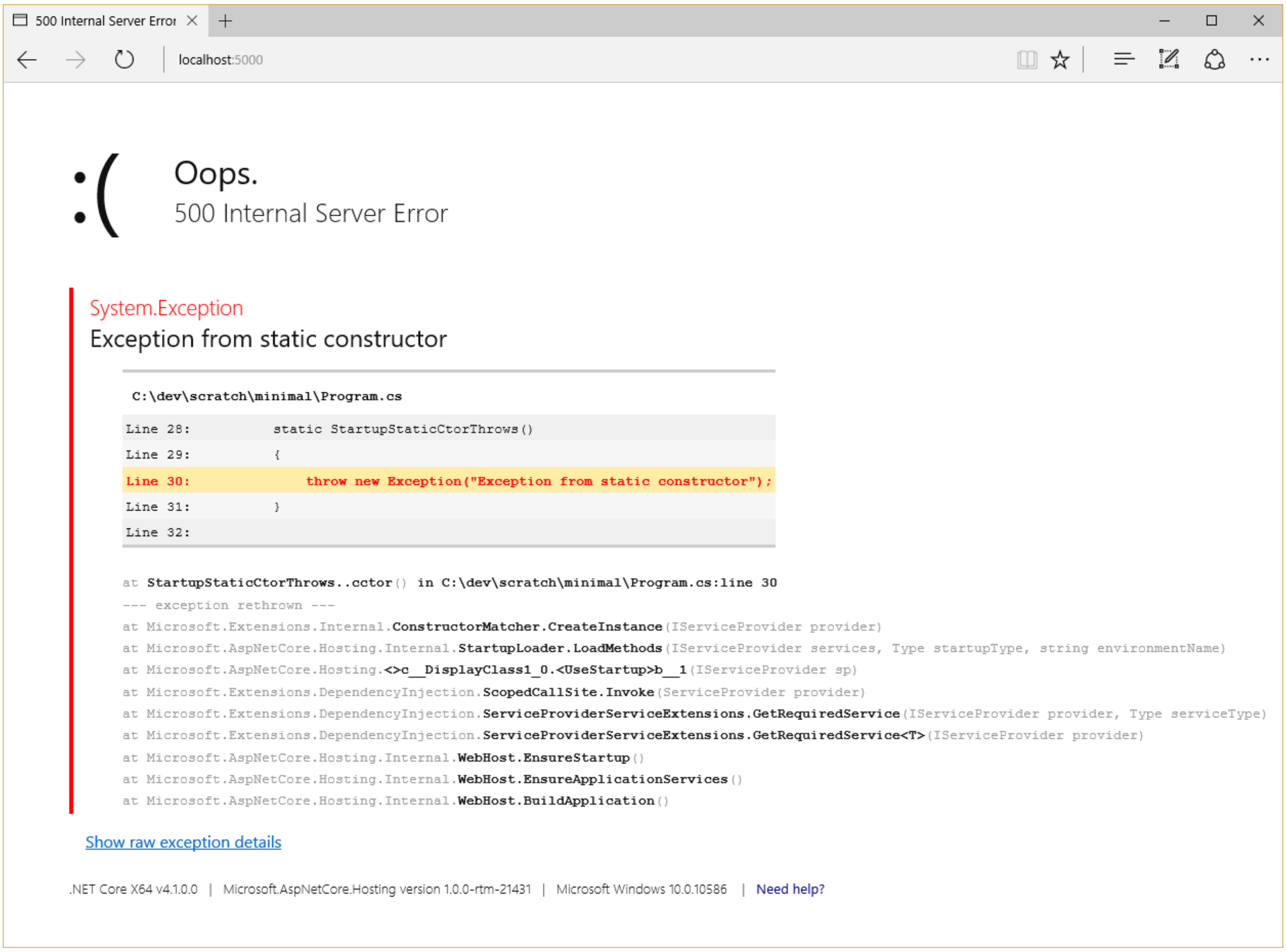
Key: `detailedErrors`. Defaults to `false`. When `true` (or when Environment is set to "Development"), the app will display details of startup exceptions, instead of just a generic error page. Set using `UseSetting`.

```
new WebHostBuilder()
    .UseSetting("detailedErrors", "true")
```

When Detailed Errors is set to `false` and Capture Startup Errors is `true`, a generic error page is displayed in response to every request to the server.



When Detailed Errors is set to `true` and Capture Startup Errors is `true`, a detailed error page is displayed in response to every request to the server.



Environment `string`

Key: `environment`. Defaults to "Production". May be set to any value. Framework-defined values include "Development", "Staging", and "Production". Values are not case sensitive. See Working with Multiple Environments. Set using the `UseEnvironment` method.

```
new WebHostBuilder()
    .UseEnvironment("Development")
```

Note

By default, the environment is read from the `ASPNETCORE_ENVIRONMENT` environment variable. When using Visual Studio, environment variables may be set in the *launchSettings.json* file.

Server URLs `string`

Key: `urls`. Set to a semicolon (;) separated list of URL prefixes to which the server should respond. For example, `http://localhost:123`. The domain/host name can be replaced with "*" to indicate the server should listen to requests on any IP address or host using the specified port and protocol (for example, `http://*:5000` or `https://*:5001`). The protocol (`http://` or `https://`) must be included with each URL. The prefixes are interpreted by the configured server; supported formats will vary between servers.

```
new WebHostBuilder()
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

Startup Assembly `string`

Key: `startupAssembly`. Determines the assembly to search for the `Startup` class. Set using the `UseStartup` method. May instead reference specific type using `WebHostBuilder.UseStartup<StartupType>`. If multiple `UseStartup` methods are called, the last one takes precedence.

```
new WebHostBuilder()
    .UseStartup("StartupAssemblyName")
```

Web Root `string`

Key: `webroot`. If not specified the default is `(Content Root Path)\wwwroot`, if it exists. If this path doesn't exist, then a no-op file provider is used. Set using `UseWebRoot`.

```
new WebHostBuilder()
    .UseWebRoot("public")
```

Overriding Configuration

Use Configuration to set configuration values to be used by the host. These values may be subsequently overridden. This is specified using `UseConfiguration`.

```
public static void Main(string[] args)
{
    var config = new ConfigurationBuilder()
        .AddJsonFile("hosting.json", optional: true)
        .AddCommandLine(args)
        .Build();

    var host = new WebHostBuilder()
        .UseConfiguration(config)
        .UseKestrel()
        .Configure(app =>
        {
            app.Run(async (context) => await context.Response.WriteAsync("Hi!"));
        })
        .Build();

    host.Run();
}
```

In the example above, command-line arguments may be passed in to configure the host, or configuration settings may optionally be specified in a *hosting.json* file. To specify the host run on a particular URL, you could pass in the desired value from a command prompt:

```
dotnet run --urls "http://*:5000"
```

The `Run` method starts the web app and blocks the calling thread until the host is shutdown.

```
host.Run();
```

You can run the host in a non-blocking manner by calling its `Start` method:

```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

Pass a list of URLs to the `Start` method and it will listen on the URLs specified:

```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

Ordering Importance

`WebHostBuilder` settings are first read from certain environment variables, if set. These environment variables must use the format `ASPNETCORE_{configurationKey}`, so for example to set the URLs the server will listen on by default, you would set `ASPNETCORE_URLS`.

You can override any of these environment variable values by specifying configuration (using `UseConfiguration`) or by setting the value explicitly (using `UseUrls` for instance). The host will use whichever option sets the value last. For this reason, `UseIISIntegration` must appear after `UseUrls`, because it replaces the URL with one dynamically provided by IIS. If you want to programmatically set the default URL to one value, but allow it to be overridden with configuration, you could configure the host as follows:

```
var config = new ConfigurationBuilder()
    .AddCommandLine(args)
    .Build();

var host = new WebHostBuilder()
    .UseUrls("http://*:1000") // default URL
    .UseConfiguration(config) // override from command line
    .UseKestrel()
    .Build();
```

Additional resources

Publishing to IIS

Publish to a Linux Production Environment

Hosting ASP.NET Core as a Windows Service

Hosting ASP.NET Core Embedded in Another Application

Using Apache Web Server as a reverse-proxy

Introduction to session and application state in ASP.NET Core

By Rick Anderson, Steve Smith, and Diana LaRose

HTTP is a stateless protocol. A web server treats each HTTP request as an independent request and does not retain user values from previous requests. This article discusses different ways to preserve application and session state between requests.

Session state

Session state is a feature in ASP.NET Core that you can use to save and store user data while the user browses your web app. Consisting of a dictionary or hash table on the server, session state persists data across requests from a browser. The session data is backed by a cache.

ASP.NET Core maintains session state by giving the client a cookie that contains the session ID, which is sent to the server with each request. The server uses the session ID to fetch the session data. Because the session cookie is specific to the browser, you cannot share sessions across browsers. Session cookies are deleted only when the browser session ends. If a cookie is received for an expired session, a new session that uses the same session cookie is created.

The server retains a session for a limited time after the last request. You can either set the session timeout or use the default value of 20 minutes. Session state is ideal for storing user data that is specific to a particular session but doesn't need to be persisted permanently. Data is deleted from the backing store either when you call `Session.Clear` or when the session expires in the data store. The server does not know when the browser is closed or when the session cookie is deleted.

 Warning

Do not store sensitive data in session. The client might not close the browser and clear the session cookie (and some browsers keep session cookies alive across windows). Also, a session might not be restricted to a single user; the next user might continue with the same session.

The in-memory session provider stores session data on the local server. If you plan to run your web app on a server farm, you must use sticky sessions to tie each session to a specific server. The Windows Azure Web Sites platform defaults to sticky sessions (Application Request Routing or ARR). However, sticky sessions can affect scalability and complicate web app updates. A better option is to use the Redis or SQL Server distributed caches, which don't require sticky sessions. For more information, see Working with a Distributed Cache. For details on setting up service providers, see Configuring Session later in this article.

The remainder of this section describes the options for storing user data.

TempData

ASP.NET Core MVC exposes the TempData property on a controller. This property stores data for only a single request after the current one. `TempData` is particularly useful for redirection, when data is needed for more than a single request. `TempData` is built on top of session state.

Cookie-based TempData provider

In ASP.NET Core 1.1 and higher, you can use the cookie-based TempData provider to store a user's TempData in a cookie. To enable the cookie-based TempData provider, register the `CookieTempDataProvider` service in `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    // Add CookieTempDataProvider after AddMvc and include ViewFeatures.
    // using Microsoft.AspNetCore.Mvc.ViewFeatures;
    services.AddSingleton<ITempDataProvider, CookieTempDataProvider>();
}
```

The cookie data is encoded with the Base64UrlTextEncoder. Because the cookie is encrypted and chunked, the single cookie size limit does not apply. The cookie data is not compressed, because compressing encryped data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Query strings

You can pass a limited amount of data from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. However, for this reason, you should never use query strings for sensitive data. In addition to being easily shared, including data in query strings can create opportunities for Cross-Site Request Forgery (CSRF) attacks, which can trick users into visiting malicious sites while authenticated. Attackers can then steal user data from your app or take malicious actions on behalf of the user. Any preserved application or session state must protect against CSRF attacks. For more information on CSRF attacks, see Preventing Cross-Site Request Forgery (XSRF/CSRF) Attacks in ASP.NET Core.

Post data and hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multipage forms. However, because the client can potentially tamper with the data, the server must always revalidate it.

Cookies

Cookies provide a way to store user-specific data in web applications. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the actual data stored on the server. Most browsers restrict cookies to 4096 bytes. In addition, only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated on the server. Although the durability of the cookie on a client is subject to user intervention and expiration, they are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. Because the user is only identified and not authenticated in most cases, you can typically secure a cookie by storing the user name, account name, or a unique user ID (such as a GUID) in the cookie. You can then use the cookie to access the user personalization infrastructure of a site.

HttpContext.Items

The `Items` collection is a good location to store data that is needed only while processing one particular request. The collection's contents are discarded after each request. The `Items` collection is best used as a way for components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters. For more information, see Working with HttpContext.Items, later in this article.

Cache

Caching is an efficient way to store and retrieve data. You can control the lifetime of cached items based on time and other considerations. Learn more about Caching.

Configuring Session

The `Microsoft.AspNetCore.Session` package provides middleware for managing session state. To enable the session middleware, `Startup` must contain:

Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session.
AddSession call, which requires NuGet package "Microsoft.AspNetCore.Session".
UseSession call.

The following code shows how to set up the in-memory session provider.

```
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.CookieHttpOnly = true;
        });
    }

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

You can reference Session from `HttpContext` once it is installed and configured.

If you try to access `Session` before `UseSession` has been called, the exception `InvalidOperationException: Session has not been configured for this application or request` is thrown.

If you try to create a new `Session` (that is, no session cookie has been created) after you have already begun writing to the `Response` stream, the exception `InvalidOperationException: The session cannot be established after the response has started` is thrown. The exception can be found in the web server log; it will not be displayed in the browser.

Loading Session asynchronously

The default session provider in ASP.NET Core loads the session record from the underlying IDistributedCache store asynchronously only if the ISession.LoadAsync method is explicitly called before the `TryGetValue`, `Set`, or `Remove` methods. If `LoadAsync` is not called first, the underlying session record is loaded synchronously, which could potentially impact the ability of the app to scale.

To have applications enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method is not called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Implementation Details

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named ".AspNet.Session", and it uses a path of "/". Because the cookie default does not specify a domain, it is not made available to the client-side script on the page (because `CookieHttpOnly` defaults to `true`).

To override session defaults, use `SessionOptions`:

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

    // Adds a default in-memory implementation of IDistributedCache.
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.CookieName = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
    });
}
```

The server uses the `IdleTimeout` property to determine how long a session can be idle before its contents are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session middleware (read from or written to) resets the timeout.

Because `Session` is *non-locking*, if two requests both attempt to modify the contents of session, the last one overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. Two requests that are modifying different parts of the session (different keys) might still impact each other.

Setting and getting Session values

Session is accessed through the `Session` property on `HttpContext`. This property is an ISession implementation.

The following example shows setting and getting an int and a string:

```
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}\"");
    }
```

If you add the following extension methods, you can set and get serializable objects to Session:

```
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);
    }
}
```

The following sample shows how to set and get a serializable object:

```
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.TimeOfDay.ToString();
    var currentTime = DateTime.Now.TimeOfDay.ToString();

    return Content($"Current time: {currentTime} - "
                 + $"session time: {sessionTime}");
}
```

Working with HttpContext.Items

The `HttpContext` abstraction provides support for a dictionary collection of type `IDictionary<object, object>`, called `Items`. This collection is available from the start of an *HttpRequest* and is discarded at the end of each request. You can access it by assigning a value to a keyed entry, or by requesting the value for a particular key.

In the sample below, Middleware adds `isVerified` to the `Items` collection.

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

Later in the pipeline, another middleware could access it:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Verified request? " +
        context.Items["isVerified"]);
});
```

For middleware that will only be used by a single app, `string` keys are acceptable. However, middleware that will be shared between applications should use unique object keys to avoid any chance of key collisions. If you are developing middleware that must work across multiple applications, use a unique object key defined in your middleware class as shown below:

```
public class SampleMiddleware
{
    public static readonly object SampleKey = new Object();

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[SampleKey] = "some value";
        // additional code omitted
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        string value = HttpContext.Items[SampleMiddleware.SampleKey];
    }
}
```

This approach also has the advantage of eliminating repetition of "magic strings" in multiple places in the code.

Application state data

Use Dependency Injection to make data available to all users:

Define a service containing the data (for example, a class named `MyAppData`).

```
public class MyAppData
{
    // Declare properties/methods/etc.
}
```

Add the service class to `ConfigureServices` (for example `services.AddSingleton<MyAppData>();`). Consume the data service class in each controller:

```
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.)
    }
}
```

Common errors when working with session

"Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

This is usually caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Working with a Distributed Cache and In memory caching.

Additional Resources

Sample code used in this document

Web server implementations in ASP.NET Core

By Tom Dykstra, Steve Smith, Stephen Halter, and Chris Ross

An ASP.NET Core application runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the application as sets of request features composed into an `HttpContext`.

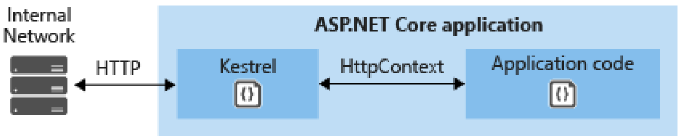
ASP.NET Core ships two server implementations:

Kestrel is a cross-platform HTTP server based on libuv, a cross-platform asynchronous I/O library.

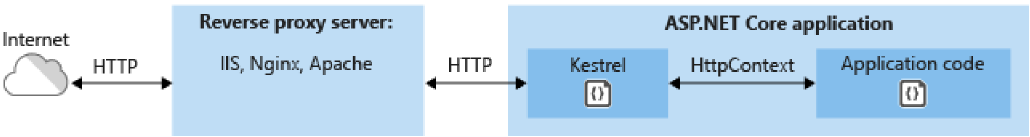
WebListener is a Windows-only HTTP server based on the Http.Sys kernel driver.

Kestrel

Kestrel is the web server that is included by default in ASP.NET Core new-project templates. If your application accepts requests only from an internal network, you can use Kestrel by itself.



If you expose your application to the Internet, you must use IIS, Nginx, or Apache as a *reverse proxy server*. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling, as shown in the following diagram.



The most important reason for using a reverse proxy for edge deployments (exposed to traffic from the Internet) is security. Kestrel is relatively new and does not yet have a full complement of defenses against attacks. This includes but isn't limited to appropriate timeouts, size limits, and concurrent connection limits. For more information about when to use Kestrel with a reverse proxy, see Kestrel.

You can't use IIS, Nginx, or Apache without Kestrel or a custom server implementation. ASP.NET Core was designed to run in its own process so that it can behave consistently across platforms. IIS, Nginx, and Apache dictate their own startup process and environment; to use them directly, ASP.NET Core would have to adapt to the needs of each one. Using a web server implementation such as Kestrel gives ASP.NET Core control over the startup process and environment. So rather than trying to adapt ASP.NET Core to IIS, Nginx, or Apache, you just set up those web servers to proxy requests to Kestrel. This arrangement allows your `Program.Main` and `Startup` classes to be essentially the same no matter where you deploy.

IIS with Kestrel

When you use IIS or IIS Express as a reverse proxy for ASP.NET Core, the ASP.NET Core application runs in a process separate from the IIS worker process. In the IIS process, a special IIS module runs to coordinate the reverse proxy relationship. This is the *ASP.NET Core Module*. The primary functions of the ASP.NET Core Module are to start the ASP.NET Core application, restart it when it crashes, and forward HTTP traffic to it. For more information, see ASP.NET Core Module.

Nginx with Kestrel

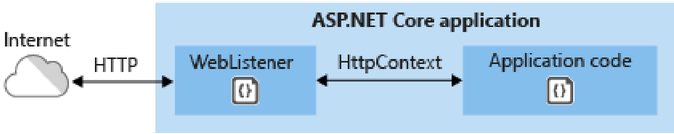
For information about how to use Nginx on Linux as a reverse proxy server for Kestrel, see Publish to a Linux Production Environment.

Apache with Kestrel

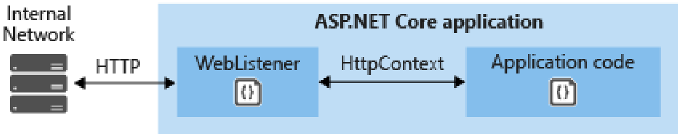
For information about how to use Apache on Linux as a reverse proxy server for Kestrel, see Using Apache Web Server as a reverse proxy.

WebListener

If you run your ASP.NET Core app on Windows, WebListener is an alternative that you can use for scenarios where you want to expose your app to the Internet but you can't use IIS.



WebListener can also be used in place of Kestrel for applications that are exposed only to an internal network, if you need one of its features that Kestrel doesn't support.



For internal network scenarios, Kestrel is generally recommended for best performance, but in some scenarios you might want to use a feature that only WebListener offers. For information about WebListener features, see WebListener.

Notes about ASP.NET Core server infrastructure

The `IApplicationBuilder` available in the `Startup` class `Configure` method exposes the `ServerFeatures` property of type `IFeatureCollection`. Kestrel and WebListener both expose only a single feature, `IServerAddressesFeature`, but different server implementations may expose additional functionality.

`IServerAddressesFeature` can be used to find out which port the server implementation has bound to at runtime.

Custom servers

You can create custom server implementations to use in place of Kestrel or WebListener. The Open Web Interface for .NET (OWIN) guide demonstrates how to write a Nowin-based `IServer` implementation. You're free to implement just the feature interfaces your application needs, though at a minimum you must support `IHttpRequestFeature` and `IHttpResponseFeature`.

Next steps

For more information, see the following resources:

Kestrel
Kestrel with IIS
Kestrel with Nginx
Kestrel with Apache
WebListener

Introduction to Kestrel web server implementation in ASP.NET Core

By Tom Dykstra, Chris Ross, and Stephen Halter

Kestrel is a cross-platform web server for ASP.NET Core based on libuv, a cross-platform asynchronous I/O library. Kestrel is the web server that is included by default in ASP.NET Core new project templates.

Kestrel supports the following features:

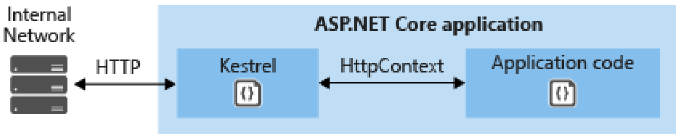
HTTPS
Opaque upgrade used to enable WebSockets
Unix sockets for high performance behind Nginx

Kestrel is supported on all platforms and versions that .NET Core supports.

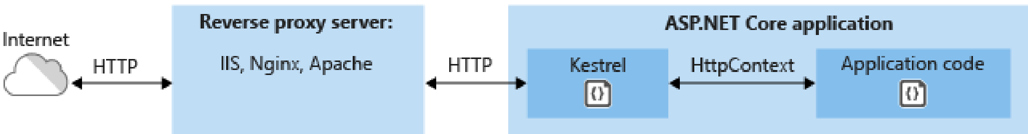
View or download sample code

When to use Kestrel with a reverse proxy

If your application accepts requests only from an internal network, you can use Kestrel by itself.



If you expose your application to the Internet, you must use IIS, Nginx, or Apache as a *reverse proxy server*. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling.



A reverse proxy is required for edge deployments (exposed to traffic from the Internet) for security reasons. Kestrel is relatively new and does not yet have a full complement of defenses against attacks. This includes but isn't limited to appropriate timeouts, size limits, and concurrent connection limits.

Another scenario that requires a reverse proxy is when you have multiple applications that share the same port running on a single server. That doesn't work with Kestrel directly because Kestrel doesn't support sharing a port between multiple processes. When you configure Kestrel to listen on a port, it handles all traffic for that port regardless of host header. A reverse proxy that can share ports must then forward to Kestrel on a unique port.

Even if a reverse proxy server isn't required, using one can simplify load balancing and SSL set-up -- only your reverse proxy server requires an SSL certificate, and that server can communicate with your application servers on the internal network using plain HTTP.

How to use Kestrel in ASP.NET Core apps

Install the Microsoft.AspNetCore.Server.Kestrel NuGet package.

Call the `UseKestrel` extension method on `WebHostBuilder` in your `Main` method, specifying any Kestrel options that you need, as shown in the following example:

```
public static int Main(string[] args)
{
    Console.WriteLine("Running demo with Kestrel.");

    var config = new ConfigurationBuilder()
        .AddCommandLine(args)
        .Build();

    var builder = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseConfiguration(config)
        .UseStartup<Startup>()
        .UseKestrel(options =>
        {
            if (config["threadCount"] != null)
            {
                options.ThreadCount = int.Parse(config["threadCount"]);
            }
        })
        .UseUrls("http://localhost:5000");

    var host = builder.Build();
    host.Run();

    return 0;
}
```

URL prefixes

By default ASP.NET Core binds to `http://localhost:5000`. You can configure URL prefixes and ports for Kestrel to listen on by using the `UseUrls` extension method, the `urls` command-line argument, or the ASP.NET Core configuration system. For more information about these methods, see Hosting. For information about how URL binding works when you use IIS as a reverse proxy, see ASP.NET Core Module.

URL prefixes for Kestrel can be in any of the following formats.

IPv4 address with port number

```
http://65.55.39.10:80/
https://65.55.39.10:443/
```

0.0.0.0 is a special case that binds to all IPv4 addresses.

IPv6 address with port number

```
http://[0:0:0:0:0:ffff:4137:270a]:80/
https://[0:0:0:0:0:ffff:4137:270a]:443/
```

[::] is the IPv6 equivalent of IPv4 0.0.0.0.

Host name with port number

```
http://contoso.com:80/
http://*:80/
https://contoso.com:443/
https://*:443/
```

Host names, *, and +, are not special. Anything that is not a recognized IP address or "localhost" will bind to all IPv4 and IPv6 IPs. If you need to bind different host names to different ASP.NET Core applications on the same port, use WebListener or a reverse proxy server such as IIS, Nginx, or Apache.

"Localhost" name with port number or loopback IP with port number

```
http://localhost:5000/
http://127.0.0.1:5000/
http://[::1]:5000/
```

When `localhost` is specified, Kestrel tries to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 is not supported), Kestrel logs a warning.

Unix socket

```
http://unix:/run/dan-live.sock
```

If you specify port number 0, Kestrel dynamically binds to an available port. Binding to port 0 is allowed for any host name or IP except for `localhost` name.

When you specify port 0, you can use `IServerAddressesFeature` to determine which port Kestrel actually bound to at runtime. The following example gets the bound port and displays it on the console.

```
public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();

    var serverAddressesFeature = app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync($"<p>Request URL: {context.Request.GetDisplayUrl()}<p>");
    });
}
```

URL prefixes for SSL

Be sure to include URL prefixes with `https:` if you call the `UseHttps` extension method, as shown below.

```
var host = new WebHostBuilder()
    .UseKestrel(options =>
    {
        options.UseHttps("testCert.pfx", "testPassword");
    })
   .UseUrls("http://localhost:5000", "https://localhost:5001")
   .UseContentRoot(Directory.GetCurrentDirectory())
   .UseStartup<Startup>()
   .Build();
```

Note

HTTPS and HTTP cannot be hosted on the same port.

Next steps

For more information, see the following resources:

Sample app for this article
Kestrel source code

Your First ASP.NET Core Application on a Mac Using Visual Studio Code.

The tutorial uses Kestrel by itself locally, then deploys the app to Azure where it runs under Windows using IIS as a reverse proxy server.

Introduction to ASP.NET Core Module

By Tom Dykstra, Rick Strahl, and Chris Ross

ASP.NET Core Module (ANCM) lets you run ASP.NET Core applications behind IIS, using IIS for what it's good at (security, manageability, and lots more) and using Kestrel for what it's good at (being really fast), and getting the benefits from both technologies at once. **ANCM works only with Kestrel; it isn't compatible with WebListener.**

Supported Windows versions:

Windows 7 and Windows Server 2008 R2 and later

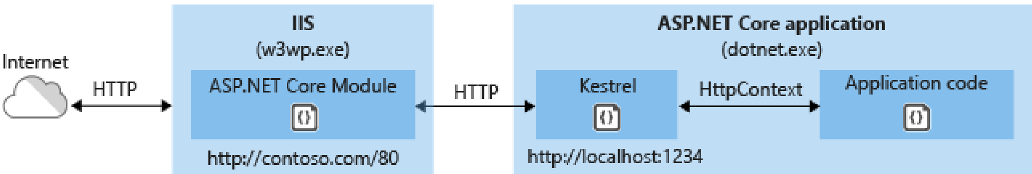
View or download sample code

What ASP.NET Core Module does

ANCM is a native IIS module that hooks into the IIS pipeline and redirects traffic to the backend ASP.NET Core application. Most other modules, such as windows authentication, still get a chance to run. ANCM only takes control when a handler is selected for the request, and handler mapping is defined in the application *web.config* file.

Because ASP.NET Core applications run in a process separate from the IIS worker process, ANCM also does process management. ANCM starts the process for the ASP.NET Core application when the first request comes in and restarts it when it crashes. This is essentially the same behavior as classic ASP.NET applications that run in-process in IIS and are managed by WAS (Windows Activation Service).

Here's a diagram that illustrates the relationship between IIS, ANCM, and ASP.NET Core applications.



Requests come in from the Web and hit the kernel mode Http.Sys driver which routes them into IIS on the primary port (80) or SSL port (443). ANCM forwards the requests to the ASP.NET Core application on the HTTP port configured for the application, which is not port 80/443.

Kestrel listens for traffic coming from ANCM. ANCM specifies the port via environment variable at startup, and the UseIISIntegration method configures the server to listen on `http://localhost:{port}`. There are additional checks to reject requests not from ANCM. (ANCM does not support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.)

Kestrel picks up requests from ANCM and pushes them into the ASP.NET Core middleware pipeline, which then handles them and passes them on as `HttpContext` instances to application logic. The application's responses are then passed back to IIS, which pushes them back out to the HTTP client that initiated the requests.

ANCM has a few other functions as well:

Sets environment variables.
Logs `stdout` output to file storage.
Forwards Windows authentication tokens.

How to use ANCM in ASP.NET Core apps

This section provides an overview of the process for setting up an IIS server and ASP.NET Core application. For detailed instructions, see Publishing to IIS.

Install ANCM

The ASP.NET Core Module has to be installed in IIS on your servers and in IIS Express on your development machines. For servers, ANCM is included in the ASP.NET Core Server Hosting Bundle. For development machines, Visual Studio automatically installs ANCM in IIS Express, and in IIS if it is already installed on the machine.

Install the IISIntegration NuGet package

In your application, install Microsoft.AspNetCore.Server.IISIntegration. This is an interoperability pack that reads environment variables broadcast by ANCM to set up your app. The environment variables provide configuration information such as the port to listen on.

Call UseIISIntegration

In your application's `Main` method, call the `UseIISIntegration` extension method on `WebHostBuilder`.

```
public static int Main(string[] args)
{
    var config = new ConfigurationBuilder()
        .AddCommandLine(args)
        .Build();

    var builder = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseConfiguration(config)
        .UseStartup<Startup>()
        .UseUrls("http://localhost:5001")
        .UseIISIntegration()
        .UseKestrel(options =>
        {
            if (config["threadCount"] != null)
            {
                options.ThreadCount = int.Parse(config["threadCount"]);
            }
        });

    var host = builder.Build();
    host.Run();

    return 0;
}
```

The `UseIISIntegration` method looks for environment variables that ANCM sets, and it does nothing if they aren't found. This behavior facilitates scenarios like developing and testing on MacOS and deploying to a server that runs IIS. While running on the Mac, Kestrel acts as the web server, but when the app is deployed to the IIS environment, it automatically hooks up to ANCM and IIS.

Don't call UseUrls

ANCM generates a dynamic port to assign to the back-end process. `IWebHostBuilder.UseIISIntegration` picks up this dynamic port and configures Kestrel to listen on `http://locahost:{dynamicPort}/`. This overwrites other URL configurations like calls to `IWebHostBuilder.UseUrls`. Therefore, you don't need to call `UseUrls` when you use ANCM. When you run the app without IIS, it listens on the default port number at `http://localhost:5000`.

If you need to set the port number for when you run the app without IIS, you can call `UseURLs`. When you run without IIS, the port number that you provide to `UseUrls` will take effect because `IISIntegration` will do nothing. But when you run with IIS, the port number specified by ANCM will override whatever you passed to `UseUrls`.

In ASP.NET Core 1.0, if you call `UseUrls`, do it **before** you call `IISIntegration` so that the ANCM-configured port doesn't get overwritten. This calling order is not required in ASP.NET Core 1.1, because the ANCM setting will always override `UseUrls`.

Configure ANCM options in Web.config

Configuration for the ASP.NET Core Module is stored in the *Web.config* file that is located in the application's root folder. Settings in this file point to the startup command and arguments that start your ASP.NET Core app. For sample Web.config code and guidance on configuration options, see ASP.NET Core Module Configuration Reference.

Run with IIS Express in development

IIS Express can be launched by Visual Studio using the default profile defined by the ASP.NET Core templates.

Next steps

For more information, see the following resources:

Sample app for this article
ASP.NET Core Module source code
ASP.NET Core Module Configuration Reference
Publishing to IIS

WebListener web server implementation in ASP.NET Core

By Tom Dykstra and Chris Ross

WebListener is a web server for ASP.NET Core that runs only on Windows. It's built on the Http.Sys kernel mode driver. WebListener is an alternative to Kestrel that can be used for direct connection to the Internet without relying on IIS as a reverse proxy server. In fact, **WebListener can't be used with IIS or IIS Express, as it isn't compatible with the ASP.NET Core Module.**

Although WebListener was developed for ASP.NET Core, it can be used directly in any .NET Core or .NET Framework application via the Microsoft.Net.Http.Server NuGet package.

WebListener supports the following features:

Windows Authentication
Port sharing
HTTPS with SNI
HTTP/2 over TLS (Windows 10)
Direct file transmission
Response caching
WebSockets (Windows 8)

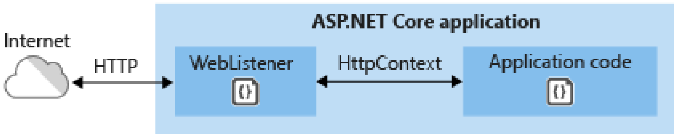
Supported Windows versions:

Windows 7 and Windows Server 2008 R2 and later

View or download sample code

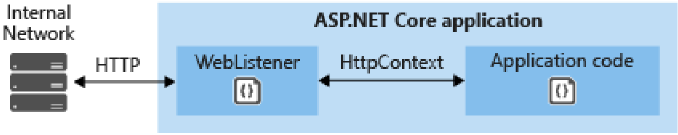
When to use WebListener

WebListener is useful for deployments where you need to expose the server directly to the Internet without using IIS.



Because it's built on Http.Sys, WebListener doesn't require a reverse proxy server for protection against attacks. Http.Sys is mature technology that protects against many kinds of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of Http.Sys.

WebListener is also a good choice for internal deployments when you need one of the features it offers that you can't get by using Kestrel.



How to use WebListener

Here's an overview of setup tasks for the host OS and your ASP.NET Core application.

Configure Windows Server

Install the version of .NET that your application requires, such as .NET Core or .NET Framework 4.5.1.

Preregister URL prefixes to bind to WebListener, and set up SSL certificates

If you don't preregister URL prefixes in Windows, you have to run your application with administrator privileges. The only exception is if you bind to localhost using HTTP (not HTTPS) with a port number greater than 1024; in that case administrator privileges aren't required.

For details, see How to preregister prefixes and configure SSL later in this article.

Open firewall ports to allow traffic to reach WebListener.

You can use netsh.exe or PowerShell cmdlets.

There are also Http.Sys registry settings.

Configure your ASP.NET Core application

Install the NuGet package Microsoft.AspNetCore.Server.WebListener. This also installs Microsoft.Net.Http.Server as a dependency.

Call the `UseWebListener` extension method on WebHostBuilder in your `Main` method, specifying any WebListener options and settings that you need, as shown in the following example:

```
public static int Main(string[] args)
{
    Console.WriteLine("Running demo with WebListener.");

    var config = new ConfigurationBuilder()
        .AddCommandLine(args)
        .Build();

    var builder = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseConfiguration(config)
        .UseStartup<Startup>()
        .UseWebListener(options =>
        {
            options.ListenerSettings.Authentication.Schemes = AuthenticationSchemes.None;
            options.ListenerSettings.Authentication.AllowAnonymous = true;
        });

    var host = builder.Build();
    host.Run();

    return 0;
}
```

Configure URLs and ports to listen on

By default ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, you can use the `UseURLs` extension method, the `urls` command-line argument or the ASP.NET Core configuration system. For more information, see Hosting.

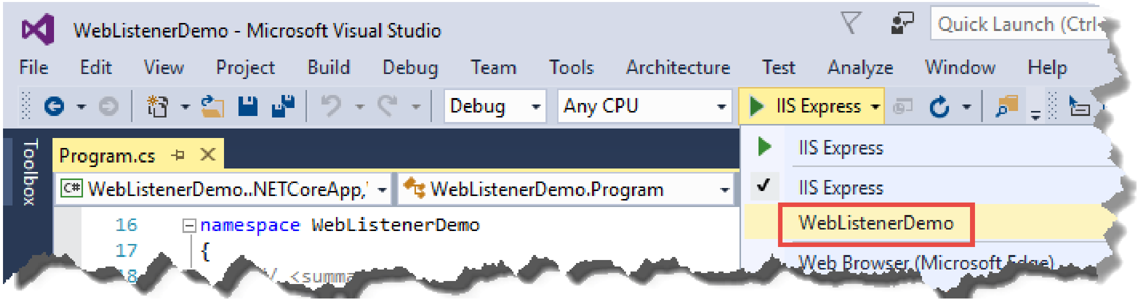
Web Listener uses the Http.Sys prefix string formats. There are no prefix string format requirements that are specific to WebListener.

Note

Make sure that you specify the same prefix strings in `UseUrls` that you preregister on the server.

Make sure your application is not configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console application you have to manually change the selected profile, as shown in the following screen shot.



How to use WebListener outside of ASP.NET Core

Install the Microsoft.Net.Http.Server NuGet package.

Preregister URL prefixes to bind to WebListener, and set up SSL certificates as you would for use in ASP.NET Core.

There are also Http.Sys registry settings.

Here's a code sample that demonstrates WebListener use outside of ASP.NET Core:

```
var settings = new WebListenerSettings();
settings.UrlPrefixes.Add("http://localhost:8080");

using (WebListener listener = new WebListener(settings))
{
    listener.Start();

    while (true)
    {
        var context = await listener.AcceptAsync();
        byte[] bytes = Encoding.ASCII.GetBytes("Hello World: " + DateTime.Now);
        context.Response.ContentLength = bytes.Length;
        context.Response.ContentType = "text/plain";

        await context.Response.Body.WriteAsync(bytes, 0, bytes.Length);
        context.Dispose();
    }
}
```

Preregister URL prefixes and configure SSL

Both IIS and WebListener rely on the underlying Http.Sys kernel mode driver to listen for requests and do initial processing. In IIS, the management UI gives you a relatively easy way to configure everything. However, if you're using WebListener you need to configure Http.Sys yourself. The built-in tool for doing that is netsh.exe.

The most common tasks you need to use netsh.exe for are reserving URL prefixes and assigning SSL certificates.

NetSh.exe is not an easy tool to use for beginners. The following example shows the bare minimum needed to reserve URL prefixes for ports 80 and 443:

```
netsh http add urlacl url=http://+:80/ user=Users
netsh http add urlacl url=https://+:443/ user=Users
```

The following example shows how to assign an SSL certificate:

```
netsh http add sslcert ipport=0.0.0.0:443 certhash=MyCertHash_Here appid={00000000-0000-0000-0000-
000000000000}".
```

Here is the official reference documentation:

Netsh Commands for Hypertext Transfer Protocol (HTTP)
UrlPrefix Strings

The following resources provide detailed instructions for several scenarios. Articles that refer to `HttpListener` apply equally to `WebListener`, as both are based on Http.Sys.

How to: Configure a Port with an SSL Certificate
HTTPS Communication - HttpListener based Hosting and Client Certification This is a third-party blog and is fairly old but still has useful information.
How To: Walkthrough Using HttpListener or Http Server unmanaged code (C++) as an SSL Simple Server This too is an older blog with useful information.
How Do I Set Up A .NET Core WebListener With SSL?

Here are some third-party tools that can be easier to use than the netsh.exe command line. These are not provided by or endorsed by Microsoft. The tools run as administrator by default, since netsh.exe itself requires administrator privileges.

HttpSysManager provides UI for listing and configuring SSL certificates and options, prefix reservations, and certificate trust lists.
HttpConfig lets you list or configure SSL certificates and URL prefixes. The UI is more refined than HttpSysManager and exposes a few more configuration options, but otherwise it provides similar functionality. It cannot create a new certificate trust list (CTL), but can assign existing ones.

For generating self-signed SSL certificates, Microsoft provides command-line tools: MakeCert.exe and the PowerShell cmdlet New-SelfSignedCertificate. There are also third-party UI tools that make it easier for you to generate self-signed SSL certificates:

SelfCert
Makecert UI

Next steps

For more information, see the following resources:

Sample app for this article
WebListener source code
Hosting

Request Features in ASP.NET Core

By Steve Smith

Web server features related to how HTTP requests and responses are handled have been factored into interfaces. These interfaces are used by server implementations and middleware to create and modify the application's hosting pipeline.

Feature interfaces

ASP.NET Core defines a number of HTTP feature interfaces in `Microsoft.AspNetCore.Http.Features` which are used by servers to identify the features they support. The following feature interfaces handle requests and return responses:

`IHttpRequestFeature` Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body.

`IHttpResponseFeature` Defines the structure of an HTTP response, including the status code, headers, and body of the response.

`IHttpAuthenticationFeature` Defines support for identifying users based on a `ClaimsPrincipal` and specifying an authentication handler.

`IHttpUpgradeFeature` Defines support for HTTP Upgrades, which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

`IHttpBufferingFeature` Defines methods for disabling buffering of requests and/or responses.

`IHttpConnectionFeature` Defines properties for local and remote addresses and ports.

`IHttpRequestLifetimeFeature` Defines support for aborting connections, or detecting if a request has been terminated prematurely, such as by a client disconnect.

`IHttpSendFileFeature` Defines a method for sending files asynchronously.

`IHttpWebSocketFeature` Defines an API for supporting web sockets.

`IHttpRequestIdentifierFeature` Adds a property that can be implemented to uniquely identify requests.

`ISessionFeature` Defines `ISessionFactory` and `ISession` abstractions for supporting user sessions.

`ITlsConnectionFeature` Defines an API for retrieving client certificates.

`ITlsTokenBindingFeature` Defines methods for working with TLS token binding parameters.

Note

`ISessionFeature` is not a server feature, but is implemented by the `SessionMiddleware` (see Managing Application State).

Feature collections

The `Features` property of `HttpContext` provides an interface for getting and setting the available HTTP features for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features.

Middleware and request features

While servers are responsible for creating the feature collection, middleware can both add to this collection and consume features from the collection. For example, the `StaticFileMiddleware` accesses the `IHttpSendFileFeature` feature. If the feature exists, it is used to send the requested static file from its physical path. Otherwise, a slower alternative method is used to send the file. When available, the `IHttpSendFileFeature` allows the operating system to open the file and perform a direct kernel mode copy to the network card.

Additionally, middleware can add to the feature collection established by the server. Existing features can even be replaced by middleware, allowing the middleware to augment the functionality of the server. Features added to the collection are available immediately to other middleware or the underlying application itself later in the request pipeline.

By combining custom server implementations and specific middleware enhancements, the precise set of features an application requires can be constructed. This allows missing features to be added without requiring a change in server, and ensures only the minimal amount of features are exposed, thus limiting attack surface area and improving performance.

Summary

Feature interfaces define specific HTTP features that a given request may support. Servers define collections of features, and the initial set of features supported by that server, but middleware can be used to enhance these features.

Additional Resources

Servers

Middleware

Open Web Interface for .NET (OWIN)

Introduction to Open Web Interface for .NET (OWIN)

By Steve Smith and Rick Anderson

ASP.NET Core supports the Open Web Interface for .NET (OWIN). OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

OWIN provides a decoupling layer that allows two frameworks with disparate object models to be used together. The `Microsoft.AspNetCore.Owin` package provides two adapter implementations:

ASP.NET Core to OWIN
OWIN to ASP.NET Core

This allows ASP.NET Core to be hosted on top of an OWIN compatible server/host, or for other OWIN compatible components to be run on top of ASP.NET Core.

Note: Using these adapters comes with a performance cost. Applications using only ASP.NET Core components should not use the Owin package or adapters.

View or download sample code

Running OWIN middleware in the ASP.NET pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by installing this package.

OWIN middleware conforms to the OWIN specification, which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "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);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET pipeline with the `UseOwin` extension method.

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

▯ Note

Response headers should only be modified prior to the first write to the response stream.

▯ Note

Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        // do something before
        return OwinHello;
        // do something after
    });
});
```

Using ASP.NET Hosting on an OWIN-based server

OWIN-based servers can host ASP.NET applications. One such server is Nowin, a .NET OWIN web server. In the sample for this article, I've included a project that references Nowin and uses it to create an `IServer` capable of self-hosting ASP.NET Core.

```
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();
        }
    }
}
```

`IServer` is an interface that requires an `Features` property and a `Start` method.

`Start` is responsible for configuring and starting the server, which in this case is done through a series of fluent API calls that set addresses parsed from the IServerAddressesFeature. Note that the fluent configuration of the `_builder` variable specifies that requests will be handled by the `appFunc` defined earlier in the method. This `Func` is called on each request to process incoming requests.

We'll also add an `IWebHostBuilder` extension to make it easy to add and configure the Nowin server.

```csharp
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();
        }
    }
}
```

With this in place, all that's required to run an ASP.NET application using this custom server to call the extension in *Program.cs*:

```csharp

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();
        }
    }
}
```

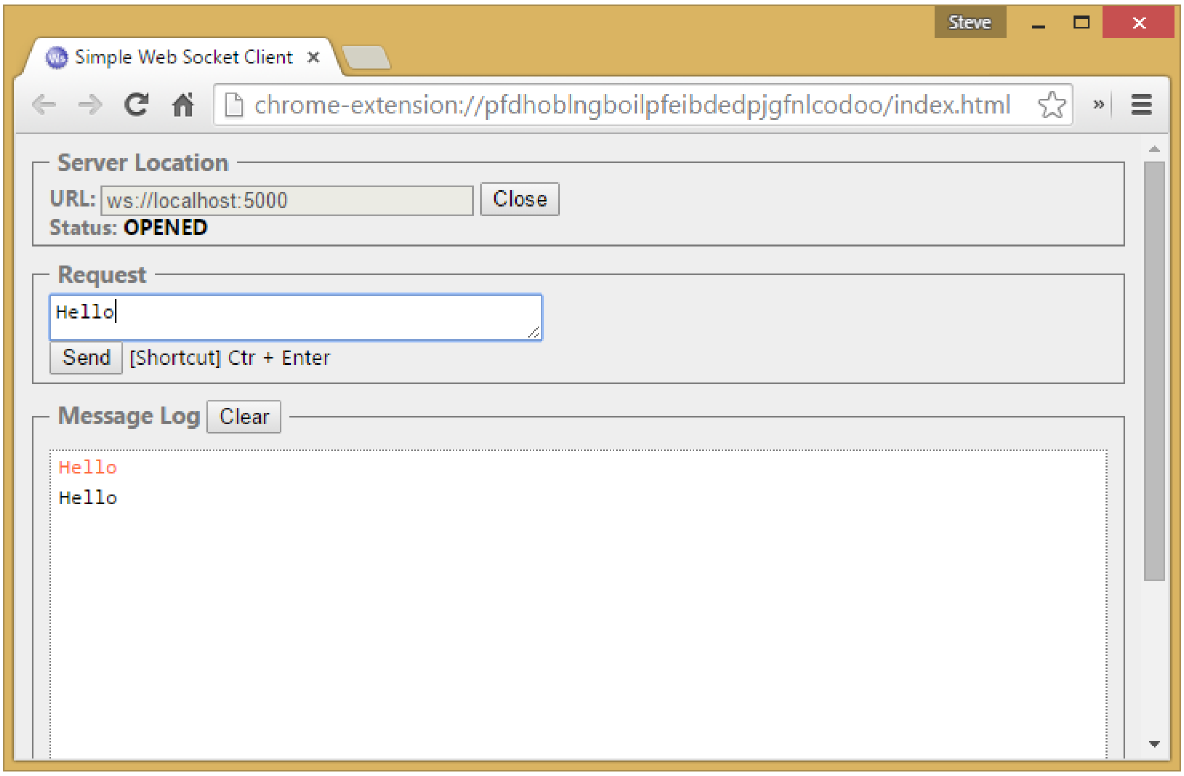
Learn more about ASP.NET Servers.

Run ASP.NET Core on an OWIN-based server and use its WebSockets support

Another example of how OWIN-based servers' features can be leveraged by ASP.NET Core is access to features like WebSockets. The .NET OWIN web server used in the previous example has support for Web Sockets built in, which can be leveraged by an ASP.NET Core application. The example below shows a simple web app that supports Web Sockets and echoes back everything sent to the server through WebSockets.

```
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);
    }
}
```

This sample is configured using the same `NowinServer` as the previous one - the only difference is in how the application is configured in its `Configure` method. A test using a simple websocket client demonstrates the application:



OWIN environment

You can construct a OWIN environment using the `HttpContext`.

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the primary specification, extensions, and OWIN Key Guidelines and Common Keys.

Request Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestScheme	`String`	
owin.RequestMethod	`String`	
owin.RequestPathBase	`String`	
owin.RequestPath	`String`	
owin.RequestQueryString	`String`	
owin.RequestProtocol	`String`	
owin.RequestHeaders	`IDictionary<string,string[]>`	

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestBody	`Stream`	

Request Data (OWIN v1.1.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestId	`String`	Optional

Response Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.ResponseStatusCode	`int`	Optional
owin.ResponseReasonPhrase	`String`	Optional
owin.ResponseHeaders	`IDictionary<string,string[]>`	
owin.ResponseBody	`Stream`	

Other Data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.CallCancelled	`CancellationToken`	
owin.Version	`String`	

Common Keys

KEY	VALUE (TYPE)	DESCRIPTION
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

KEY	VALUE (TYPE)	DESCRIPTION
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	See delegate signature
opaque.Stream	`Stream`	
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	See delegate signature
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	`String`	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	`WebSocketSendAsync`	See delegate signature
websocket.ReceiveAsync	`WebSocketReceiveAsync`	See delegate signature
websocket.CloseAsync	`WebSocketCloseAsync`	See delegate signature
websocket.CallCancelled	`CancellationToken`	
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

Additional Resources

Middleware

Servers

选择 .NET Core 还是 .NET Framework

No matter the web application you are creating, ASP.NET has a solution for you: from enterprise web applications targeting Windows Server, to small microservices targeting Linux containers, and everything in between.

ASP.NET Core

ASP.NET Core is a new open-source and cross-platform .NET framework for building modern cloud-based web applications on Windows, Mac, or Linux.

ASP.NET

ASP.NET is a mature web platform that provides all the services that you require to build enterprise-class server-based web applications using .NET on Windows.

哪一个适合我？

ASP.NET CORE	ASP.NET
Build for Windows, Mac, or Linux	Build for Windows
Use MVC, or Web API	Use Web Forms, SignalR, MVC, Web API, or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio or Visual Studio Code using C#	Develop with Visual Studio using C#, VB or F#
New platform	Mature platform
Ultra performance	High performance
Choose .NET Framework or .NET Core runtime	Use .NET Framework runtime

ASP.NET Core 系列

网站
APIs

ASP.NET 系列

网站
APIs
实时

资源

介绍 ASP.NET
介绍 ASP.NET Core

Overview of ASP.NET Core MVC

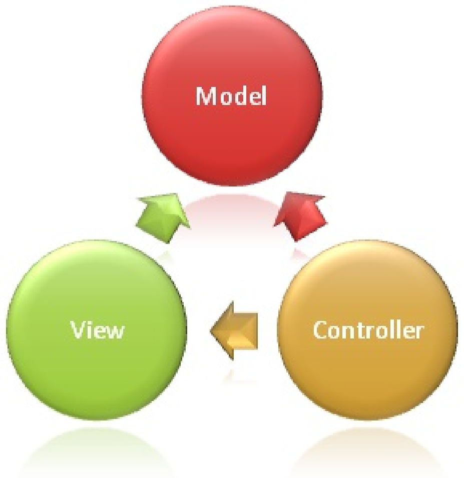
By Steve Smith

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

What is the MVC pattern?

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it’s easier to code, debug, and test something (model, view, or controller) that has a single job (and follows the Single Responsibility Principle). It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, you have to modify an object containing business logic every time you change the user interface. This is likely to introduce errors and require the retesting of all business logic after every minimal user interface change.

Note

Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views will typically use ViewModel types specifically designed to contain the data to display on that view; the controller will create and populate these ViewModel instances from the model.

Note

There are many ways to organize the model in an app that uses the MVC architectural pattern. Learn more about some different kinds of model types.

View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in

HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

 Note

Controllers should not be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, use the Single Responsibility Principle to push business logic out of the controller and into the domain model.

 Tip

If you find that your controller actions frequently perform the same kinds of actions, you can follow the Don't Repeat Yourself principle by moving these common actions into filters.

What is ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

Features

ASP.NET Core MVC includes the following features:

Routing

Model binding

Model validation

Dependency injection

Filters

Areas

Web APIs

Testability

Razor view engine

Strongly typed views

Tag Helpers

View Components

Routing

ASP.NET Core MVC is built on top of ASP.NET Core's routing, a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

Convention-based routing enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```

Attribute routing enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
  [HttpGet("{id}")]
  public IActionResult GetProduct(int id)
  {
    ...
  }
}
```

Model binding

ASP.NET Core MVC model binding converts client request data (form values, route data, query string parameters, HTTP headers) into objects that the controller can handle. As a result, your controller logic doesn't have to do the work of figuring out the incoming request data; it simply has the data as parameters to its action methods.

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null) { ... }
```

Model validation

ASP.NET Core MVC supports validation by decorating your model object with data annotation validation attributes. The validation attributes are checked on the client side before values are posted to the server, as well as on the server before the controller action is called.

```
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; }
}
```

A controller action:

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    if (ModelState.IsValid)
    {
      // work with the model
    }
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

The framework will handle validating request data both on the client and on the server. Validation logic specified on model types is added to the rendered views as unobtrusive annotations and is enforced in the browser with jQuery Validation.

Dependency injection

ASP.NET Core has built-in support for dependency injection (DI). In ASP.NET Core MVC, controllers can request needed services through their constructors, allowing them to follow the Explicit Dependencies Principle.

Your app can also use dependency injection in view files, using the `@inject` directive:

```
@inject SomeService ServiceName
<!DOCTYPE html>
<html>
<head>
  <title>@ServiceName.GetTitle</title>
</head>
<body>
  <h1>@ServiceName.GetTitle</h1>
</body>
</html>
```

Filters

Filters help developers encapsulate cross-cutting concerns, like exception handling or authorization. Filters enable running custom pre- and post-processing logic for action methods, and can be configured to run at certain points within the execution pipeline for a given request. Filters can be applied to controllers or actions as attributes (or can be run globally). Several filters (such as `Authorize`) are included in the framework.

```
[Authorize]
   public class AccountController : Controller
   {
```

Areas

Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is effectively an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models.

Web APIs

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that can reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support for formatting data as JSON or XML. Write custom formatters to add support for your own formats.

Use link generation to enable support for hypermedia. Easily enable support for cross-origin resource sharing (CORS) so that your Web APIs can be shared across multiple Web applications.

Testability

The framework's use of interfaces and dependency injection make it well-suited to unit testing, and the framework includes features (like a TestHost and InMemory provider for Entity Framework) that make integration testing quick and easy as well. Learn more about testing controller logic.

Razor view engine

ASP.NET Core MVC views use the Razor view engine to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code. Razor is used to dynamically generate web content on the server. You can cleanly mix server code with client side content and code.

```
<ul>
  @for (int i = 0; i < 5; i++) {
    <li>List item @i</li>
  }
</ul>
```

Using the Razor view engine you can define layouts, partial views and replaceable sections.

Strongly typed views

Razor views in MVC can be strongly typed based on your model. Controllers can pass a strongly typed model to views enabling your views to have type checking and IntelliSense support.

For example, the following view defines a model of type `IEnumerable<Product>`:

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
    }
</ul>
```

Tag Helpers

Tag Helpers enable server side code to participate in creating and rendering HTML elements in Razor files. You can use tag helpers to define custom tags (for example, `<environment>`) or to modify the behavior of existing tags (for example, `<label>`). Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience.

There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in LinkTagHelper can be used to create a link to the `Login` action of the `AccountsController`:

```
<p>
    Thank you for confirming your email.
    Please <a asp-controller="Account" asp-action="Login">Click here to Log in</a>.
</p>
```

The `EnvironmentTagHelper` can be used to include different scripts in your views (for example, raw or minified) based on the runtime environment, such as Development, Staging, or Production:

```
<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>
```

Tag Helpers provide an HTML-friendly development experience and a rich IntelliSense environment for creating HTML and Razor markup. Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element.

View Components

View Components allow you to package rendering logic and reuse it throughout the application. They're similar to partial views, but with associated logic.

使用 Visual Studio 创建 ASP.NET Core MVC 应用程序

本系列教程将教会你使用 Visual Studio 创建基本的 ASP.NET Core MVC 应用程序

ASP.NET Core MVC 和 Visual Studio 入门

作者 Rick Anderson

翻译： 娄宇(Lyrics)

校对： 刘怡(AlexLEWIS)、夏申斌、张硕(Apple)

这篇教程将告诉你如何使用 Visual Studio 2017 构建一个 ASP.NET Core MVC Web 应用程序的基础知识。

Note

参考 用 Visual Studio Code 在 mac 上创建首个 ASP.NET Core 应用程序 做为 Mac 系统开发教程。

如果需要 Visual Studio 2015 版本的教程，参考 VS 2015 版本ASP.NET Core 文档 PDF 格式.

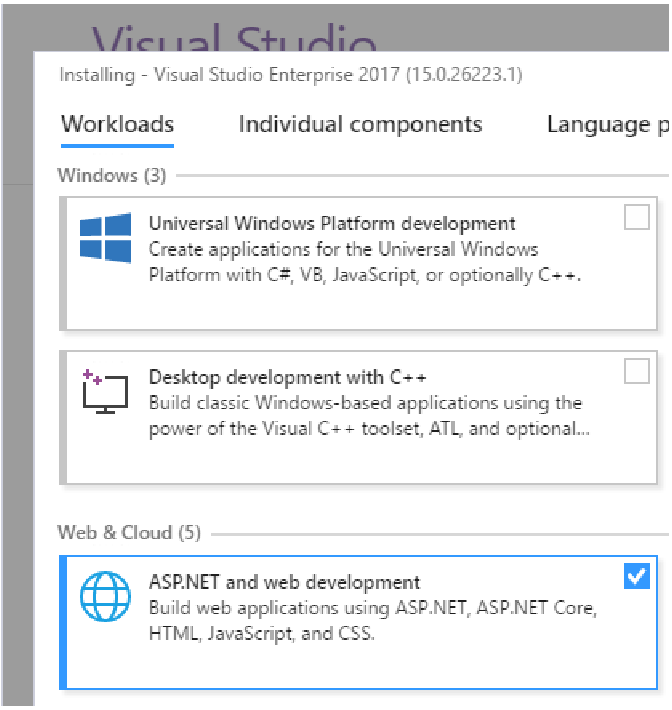
安装 Visual Studio 和 .NET Core

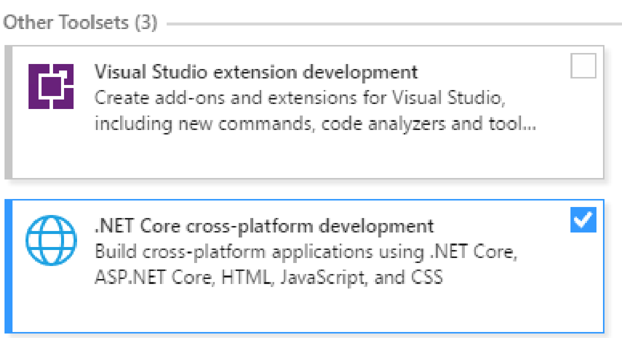
安装 Visual Studio Community 2017。选择社区版下载。如果你已经安装了 Visual Studio 2017 ，略过这一步。

Visual Studio 2017 主页安装程序

运行安装程序安装一下模块：

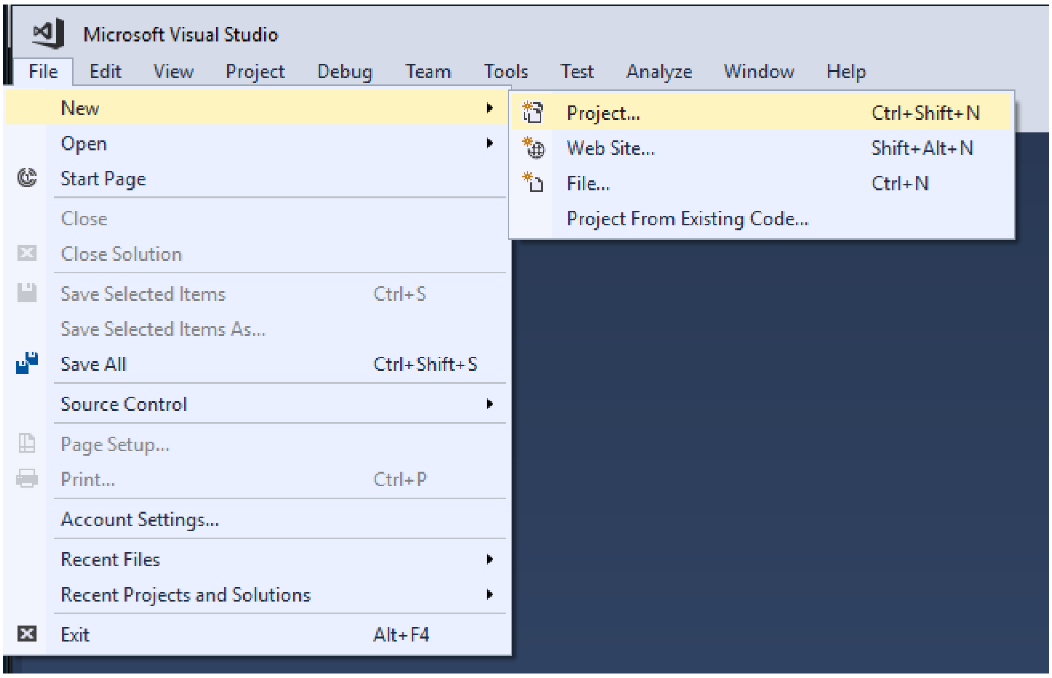
ASP.NET and web development (在 **Web & Cloud** 目录)
.NET Core cross-platform development (在 **Other Toolsets** 目录)





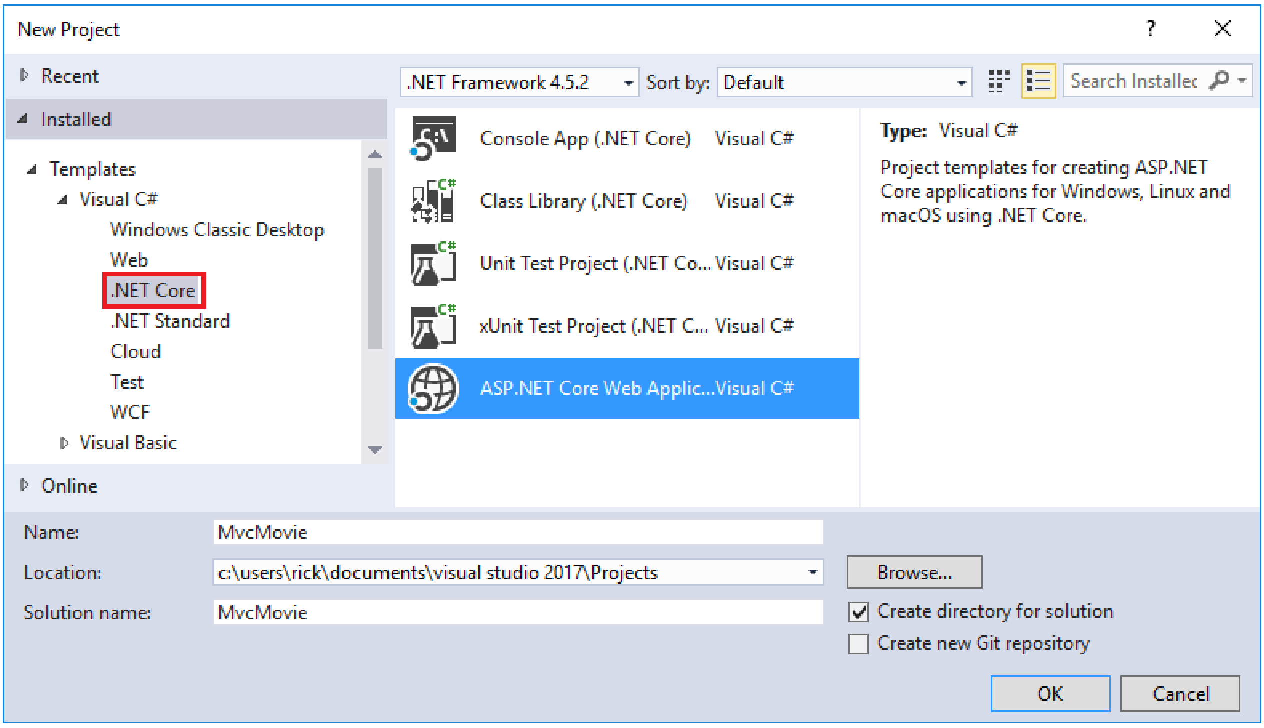
创建 Web 应用程序

在 Visual Studio 中, 选择 **File > New > Project**.



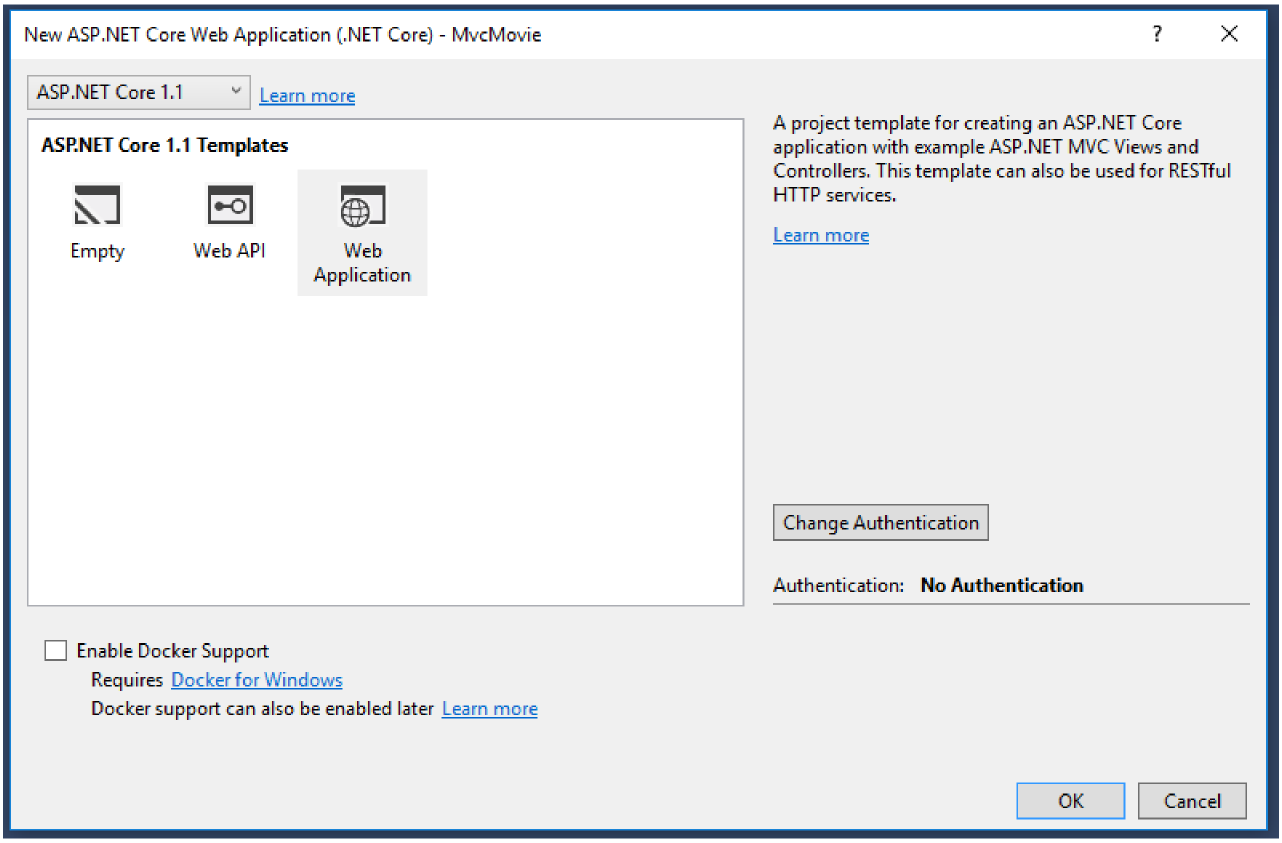
按以下步骤完成 **New Project** 对话框设置:

在右侧面板,点击 **.NET Core**
在中间面板, 点击 **ASP.NET Core Web Application (.NET Core)**
项目命名为 "MvcMovie" (请确保项目名必须是 "MvcMovie", 这样当你拷贝代码的时候, 名称空间可以保持一致。)
Tap **OK**



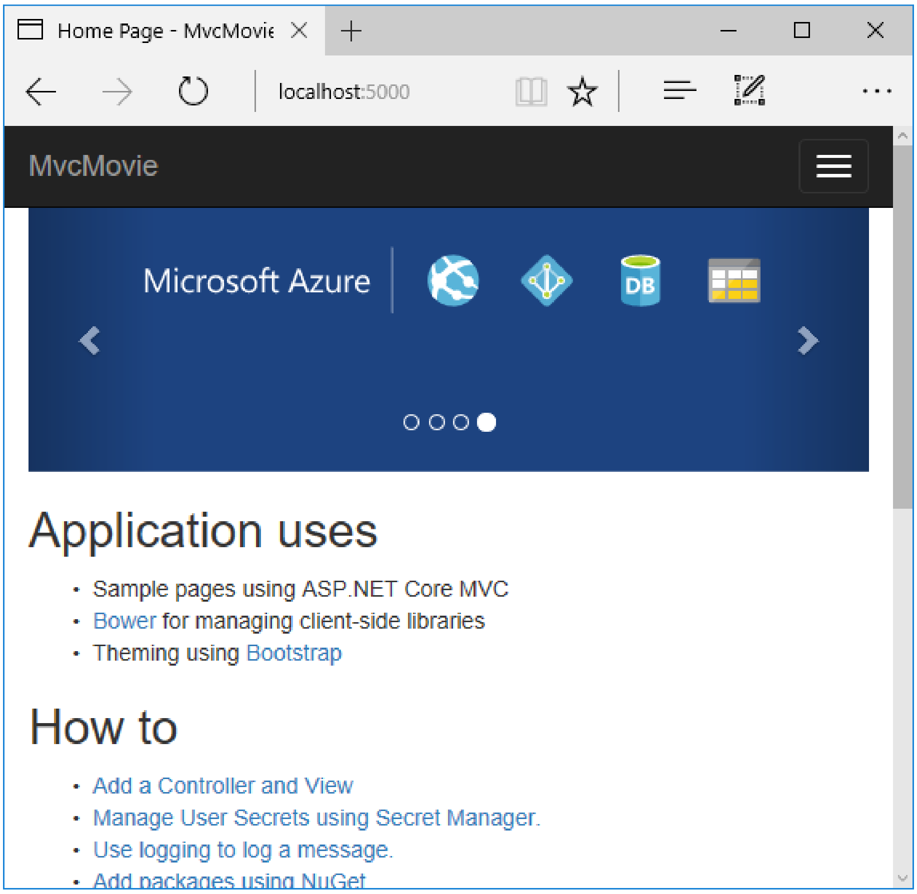
按以下步骤完成 **New ASP.NET Core Web Application (.NET Core) - MvcMovie** 对话框设置:

在版本选择下拉框中选择 **ASP.NET Core 1.1**
点击 **Web Application**
保持默认 **No Authentication** 选项
店家 **OK**.



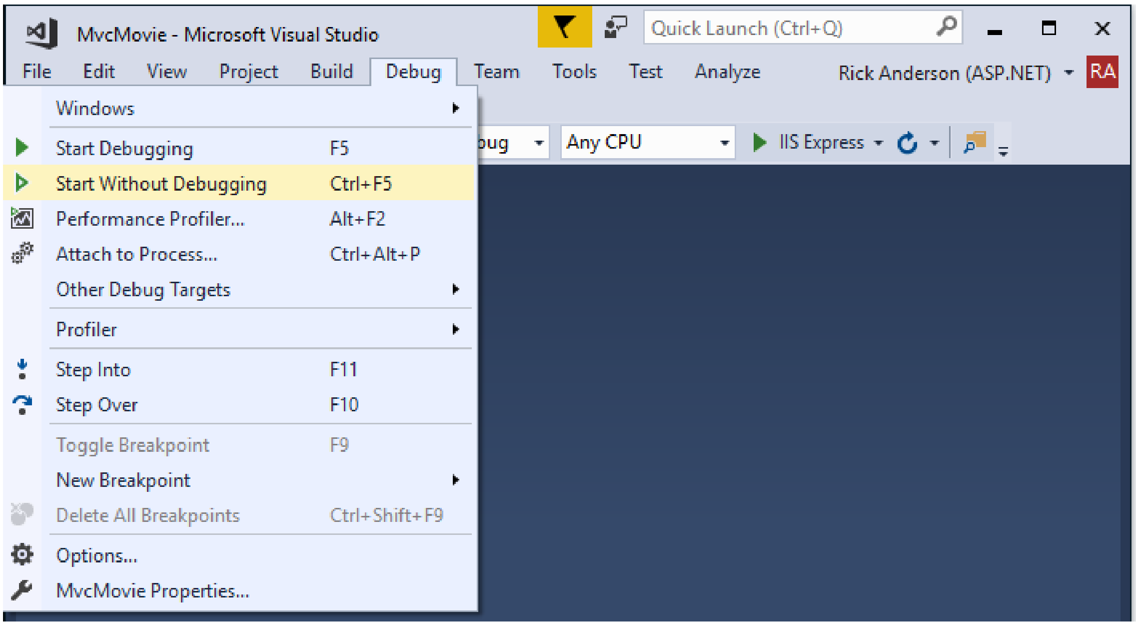
Visual Studio 给刚才创建的 MVC 项目提供了默认模板，输入项目名并选择一些选项后便可得到一个应用程序。这就是一个简单的起步项目，一个很好的开始。

按下 **F5** 以 Debug 模式运行这个应用程序，或者按下 **Ctrl+F5** 以非 Debug 模式运行。

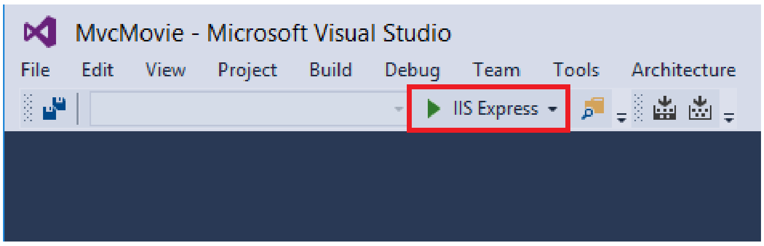


Visual Studio 启动 IIS Express 并且运行你的应用程序。注意地址栏显示的 `localhost:端口#` 而不是像 `example.com`。 那是因为 `localhost` 总是指向本地计算机，在本例中也就是运行你这个应用程序的计算机。当 Visual Studio 创建一个 Web 项目，Web 服务器使用随机的端口。如上图所示，端口号是 5000。当你运行这个应用程序，你可能会看到不同的端口号。

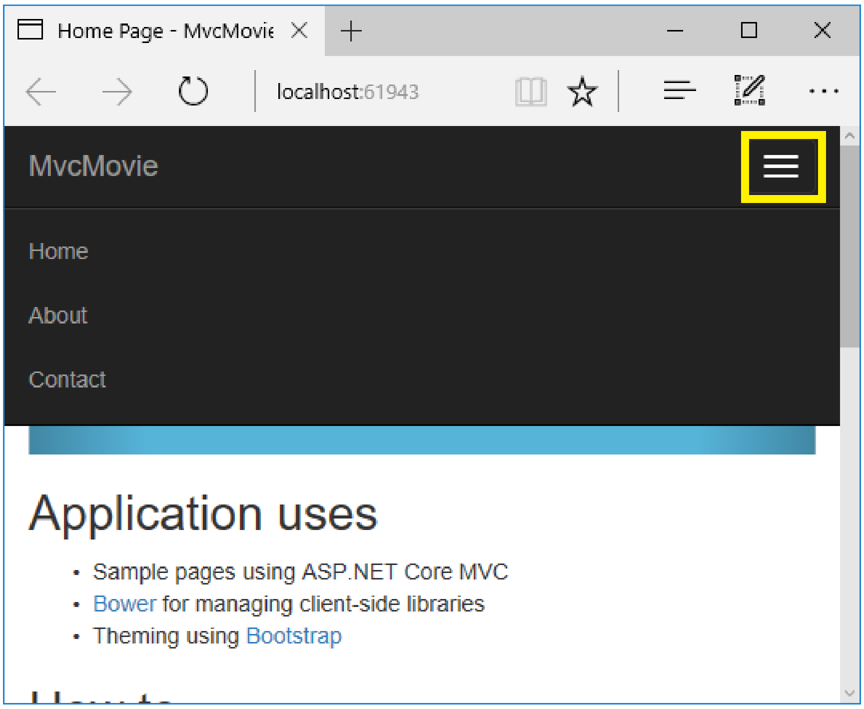
通过 **Ctrl+F5** (非调试模式)启动这个应用程序允许你进行代码更改，保存文件，刷新浏览器，之后查看代码改变。许多开发者更倾向于使用非调试模式来快速启动应用程序和查看变化。
你可以通过 **Debug** 菜单项选择以调试模式或者非调试模式启动应用程序：



你可以通过点击 **IIS Express** 按钮调试应用程序



默认的模板提供 **Home, About** 以及 **Contact** 链接。下面的浏览器图片没有显示这些链接。根据您的浏览器的尺寸，您可能需要点击导航图标来显示他们。



如果你当前在调试模式运行， 可以点击 **Shift-F5** 停止调试。

我们将在本教程下一节中学习 MVC 并尝试写些代码。

Adding a controller

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

Models: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a SQL Server database.

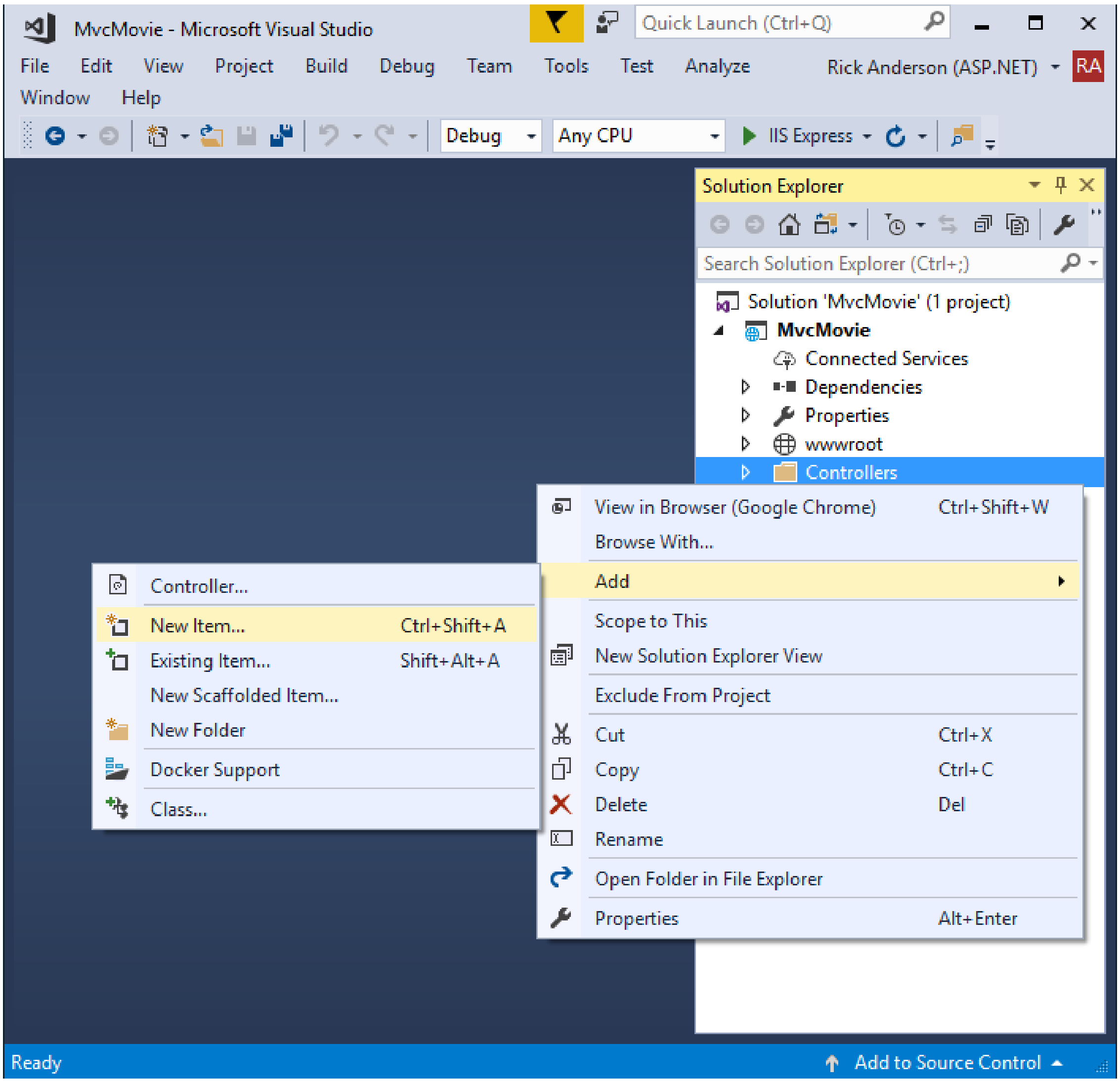
Views: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.

Controllers: Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `http://localhost:1234/Home/About` has route data of `Home` (the controller) and `About` (the action method to call on the home controller). `http://localhost:1234/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. We'll talk about route data later in the tutorial.

The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

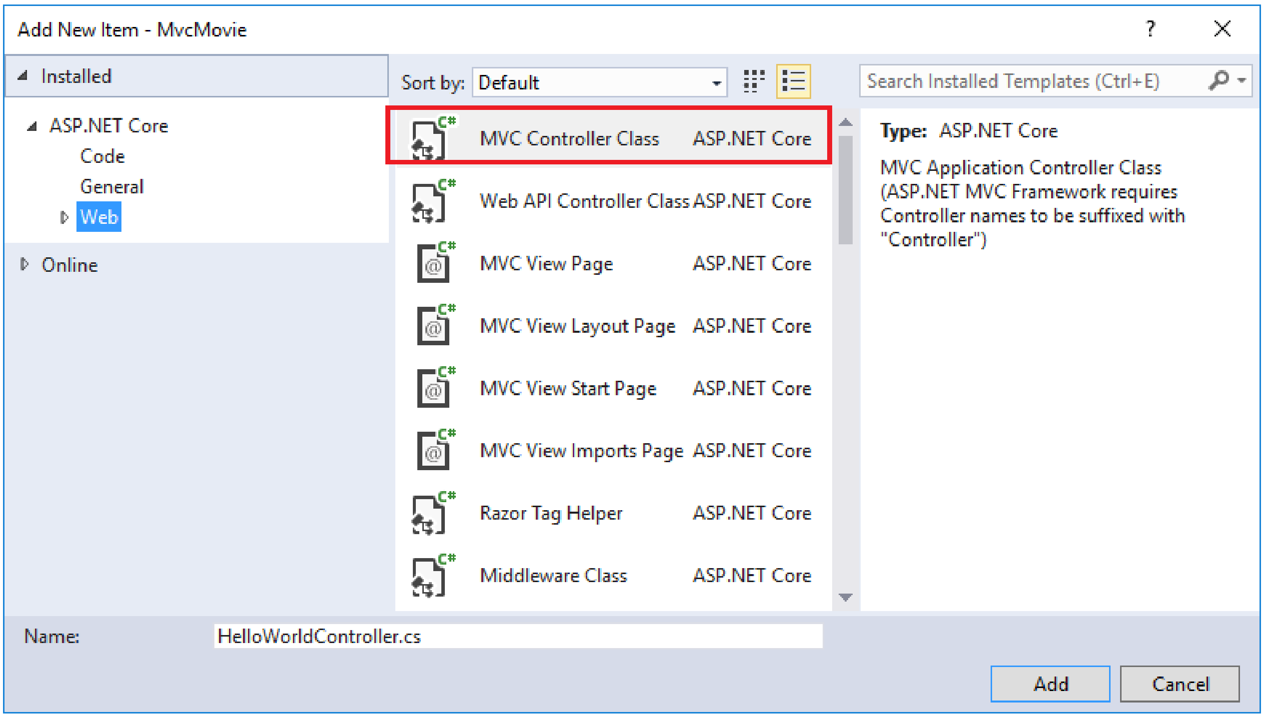
We'll be covering all these concepts in this tutorial series and show you how to use them to build a simple movie app. The MVC project currently contains folders for the *Controllers* and *Views*. A *Models* folder will be added in a later step.

In **Solution Explorer**, right-click **Controllers > Add > New Item**



Select **MVC Controller Class**

In the **Add New Item** dialog, enter **HelloWorldController**.



Replace the contents of *Controllers/HelloWorldController.cs* with the following:

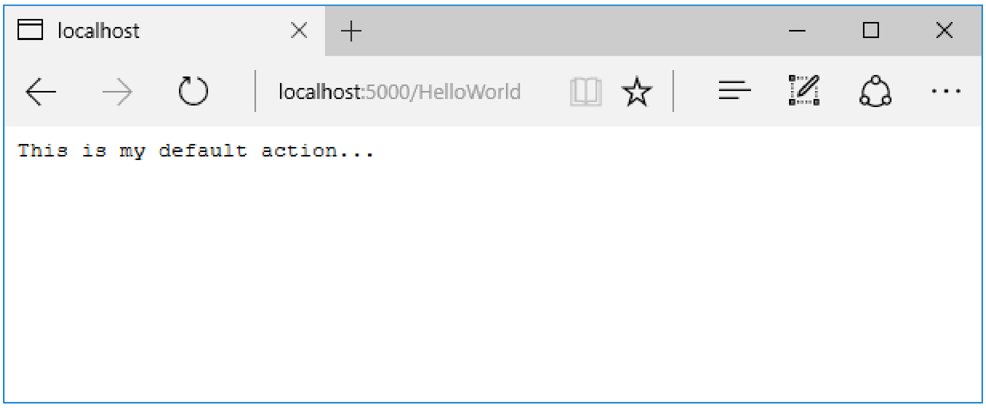
```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

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...";
        }
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `http://localhost:1234/HelloWorld`, and combines the protocol used: `HTTP`, the network location of the web server (including the TCP port): `localhost:1234` and the target URI `HelloWorld`.

The first comment states this is an HTTP GET method that is invoked by appending "/HelloWorld/" to the base URL. The second comment specifies an HTTP GET method that is invoked by appending "/HelloWorld/Welcome/" to the URL. Later on in the tutorial we'll use the scaffolding engine to generate `HTTP POST` methods.

Run the app in non-debug mode (press Ctrl+F5) and append "HelloWorld" to the path in the address bar. (In the image below, `http://localhost:5000/HelloWorld` is used, but you'll have to replace *5000* with the port number of your app.) The `Index` method returns a string. You told the system to return some HTML, and it did!



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

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

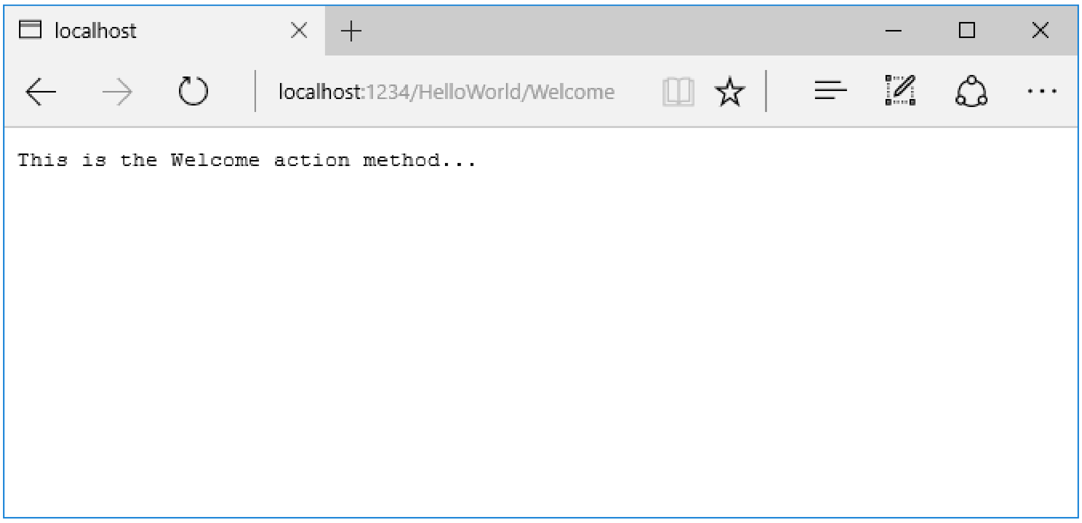
You set the format for routing in the *Startup.cs* file.

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

When you run the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:xxxx/HelloWorld` maps to the `HelloWorldController` class. The second part of the URL segment determines the action method on the class. So `localhost:xxxx/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that we only had to browse to `localhost:xxxx/HelloWorld` and the `Index` method was called by default. This is because `Index` is the default method that will be called on a controller if a method name is not explicitly specified. The third part of the URL segment (`id`) is for route data. We'll see route data later on in this tutorial.

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". For this URL, the controller is `HelloWorld` and `Welcome` is the action method. We haven't used the `[Parameters]` part of the URL yet.



Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, `/HelloWorld/Welcome?name=Scott&numtimes=4`). Change the `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.

```
// 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}");
}
```

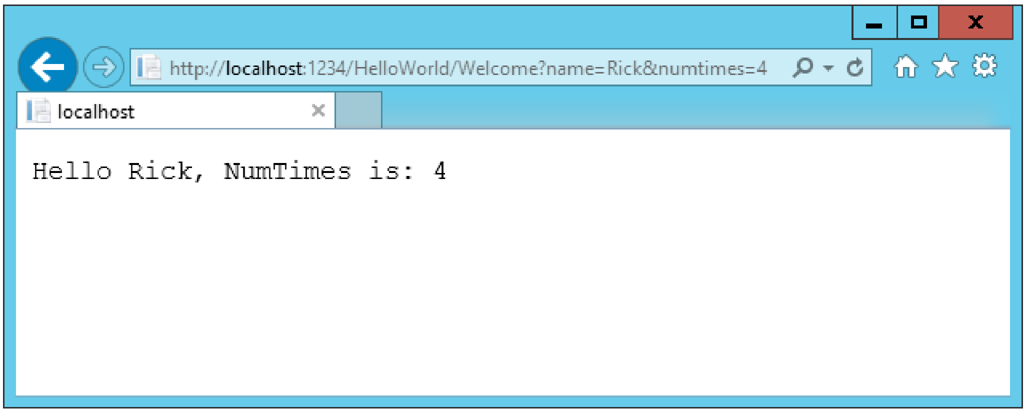
The code above uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript). It also uses Interpolated Strings.

In Visual Studio, in non-debug mode (Ctrl+F5), you don't need to build the app after changing code. Just save the file, refresh your browser and you can see the changes.

Run your app and browse to:

```
http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4
```

(Replace xxxx with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See Model Binding for more information.

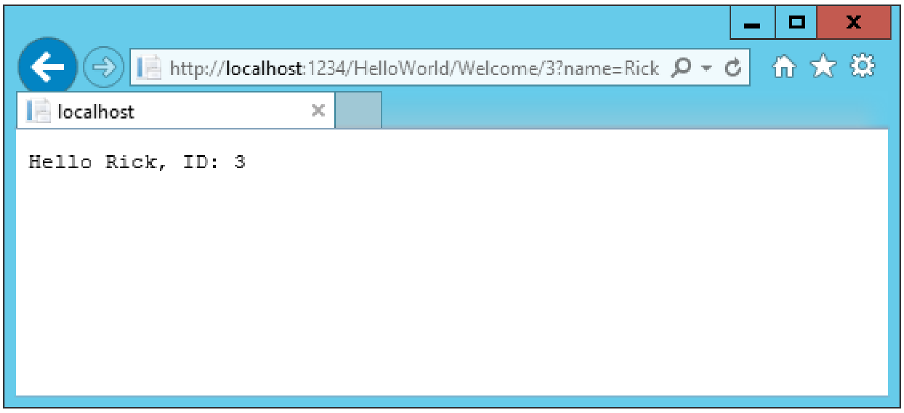


In the sample above, the URL segment (`Parameters`) is not used, the `name` and `numTimes` parameters are passed as query strings. The `?` (question mark) in the above URL is a separator, and the query strings follow. The `&` character separates query strings.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `http://localhost:xxx/HelloWorld/Welcome/3?name=Rick`



This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

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

In these examples the controller has been doing the "VC" portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead we'll typically use a separate Razor view template file to help generate the HTML response. We'll do that in the next tutorial.

上一节 下一节

Adding a view

By Rick Anderson

In this section you're going to modify the `HelloWorldController` class to use Razor view template files to cleanly encapsulate the process of generating HTML responses to a client.

You'll create a view template file using Razor. Razor-based view templates have a *.cshtml* file extension. They provide an elegant way to create HTML output using C#.

Currently the `Index` method returns a string with a message that is hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

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

The `Index` method above returns a View object. It uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods) such as the `Index` method above, generally return an `IActionResult` (or a class derived from `ActionResult`), not primitive types like string.

Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.

Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.

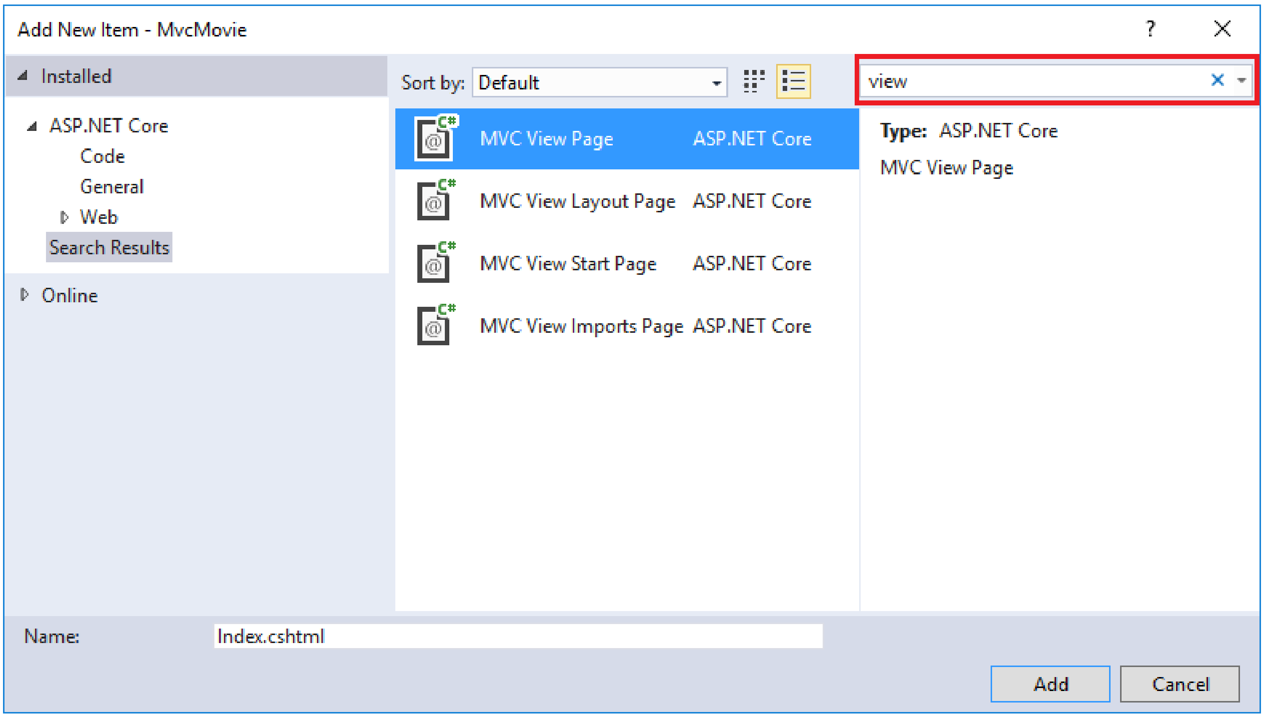
In the **Add New Item - MvcMovie** dialog

In the search box in the upper-right, enter *view*

Tap **MVC View Page**

In the **Name** box, change the name if necessary to *Index.cshtml*.

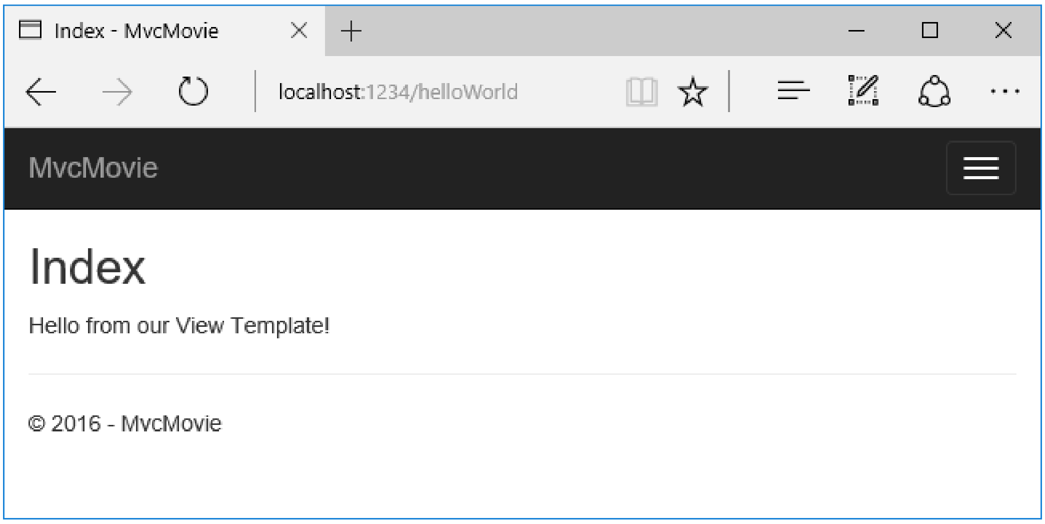
Tap **Add**



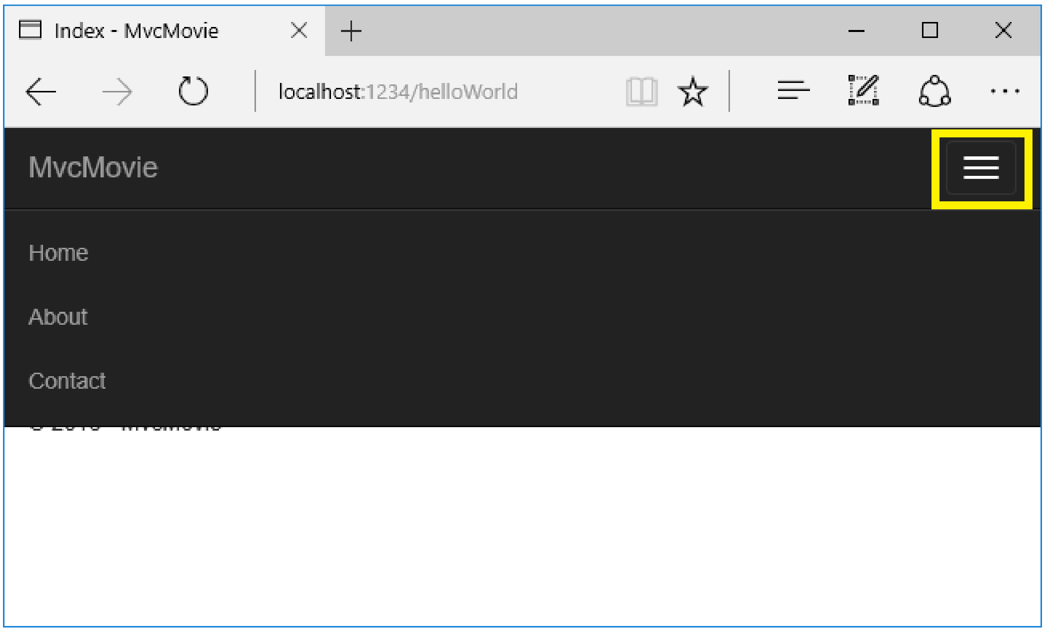
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

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

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `http://localhost:xxxx/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much work; it simply ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, MVC defaulted to using the *Index.cshtml* view file in the */Views/HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



If your browser window is small (for example on a mobile device), you might need to toggle (tap) the Bootstrap navigation button in the upper right to see the **Home**, **About**, and **Contact** links.



Changing views and layout pages

Tap the menu links (**MvcMovie**, **Home**, **About**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **About** link, the **Views/Home/About.cshtml** view is rendered inside the `RenderBody` method.

Change the title and menu link in the layout file

Change the contents of the title element. Change the anchor text in the layout template to "Movie App" and the controller from `Home` to `Movies` as highlighted below:

```
@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">MvcMovie</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-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

 Warning

We haven't implemented the `Movies` controller yet, so if you click on that link, you'll get a 404 (Not found) error.

Save your changes and tap the **About** link. Notice how the title on the browser tab now displays **About - Movie App** instead of **About - Mvc Movie**. Tap the **Contact** link and notice that it also displays **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

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

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. You can use the `Layout` property to set a different layout view, or set it to `null` so no layout file will be used.

Change the title of the `Index` view.

Open *Views/HelloWorld/Index.cshtml*. There are two places to make a change:

The text that appears in the title of the browser.
The secondary header (`<h2>` element).

You'll make them slightly different so you can see which bit of code changes which part of the app.

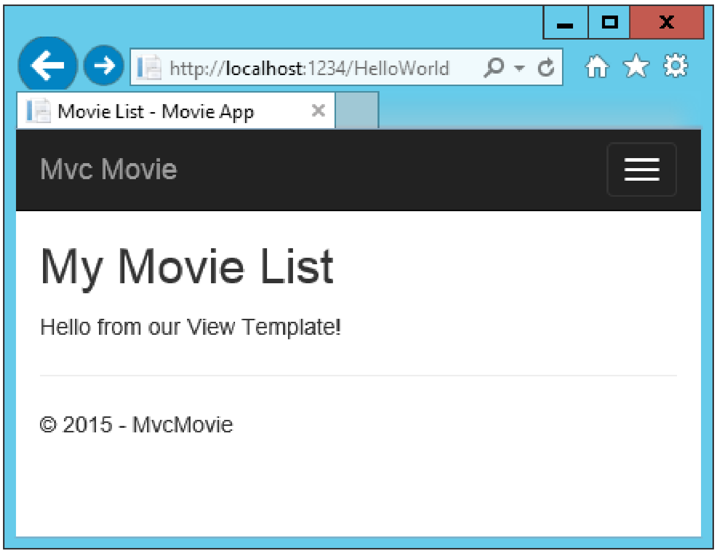
```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save your change and navigate to `http://localhost:xxxx/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *Views/Shared/_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see Layout.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

Return to the *HelloWorldController.cs* file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means you can put whatever you want in to it; the `ViewData` object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
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();
        }
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

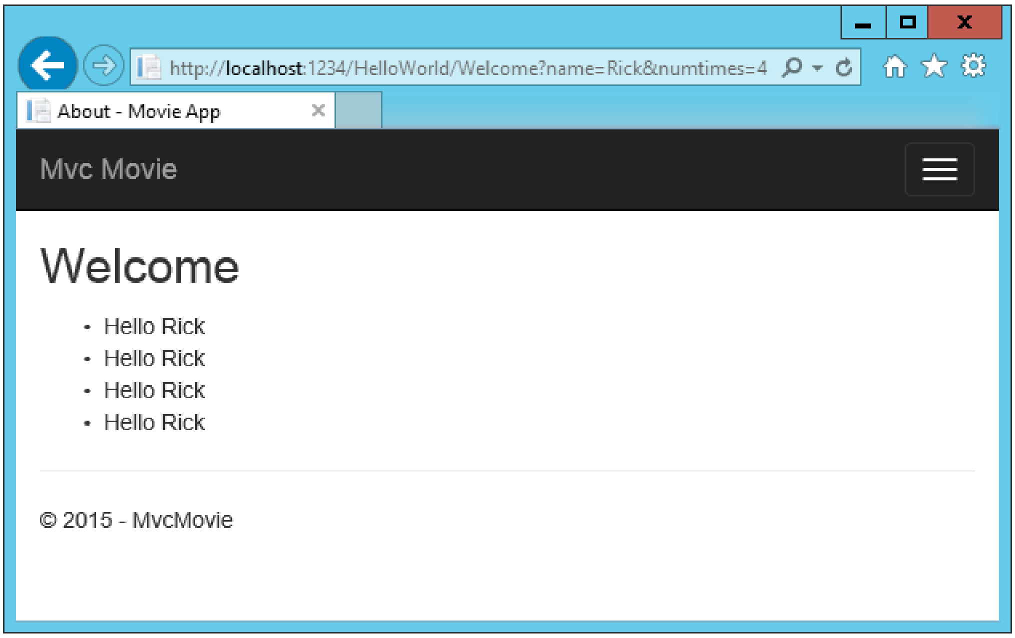
You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes`. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, we used the `ViewData` dictionary to pass data from the controller to a view. Later in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC for more information.

Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

上一节 NEXT

Adding a model

By Rick Anderson and Tom Dykstra

In this section you'll add some classes for managing movies in a database. These classes will be the "**M**odel" part of the **M**VC app.

You’ll use these classes with the Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data-access code that you have to write. For this tutorial you'll use SQLite, but EF Core supports many database engines.

The model classes you'll create are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial you'll write the model classes first, and EF Core will create the database. An alternate approach not covered here is to generate model classes from an already-existing database. For information about that approach, see ASP.NET Core - Existing Database.

Add a data model class

In Solution Explorer, right click the **MvcMovie** project > **Add** > **New Folder**. Name the folder *Models*.

In Solution Explorer, right click the *Models* folder > **Add** > **Class**. Name the class **Movie** and add the following properties:

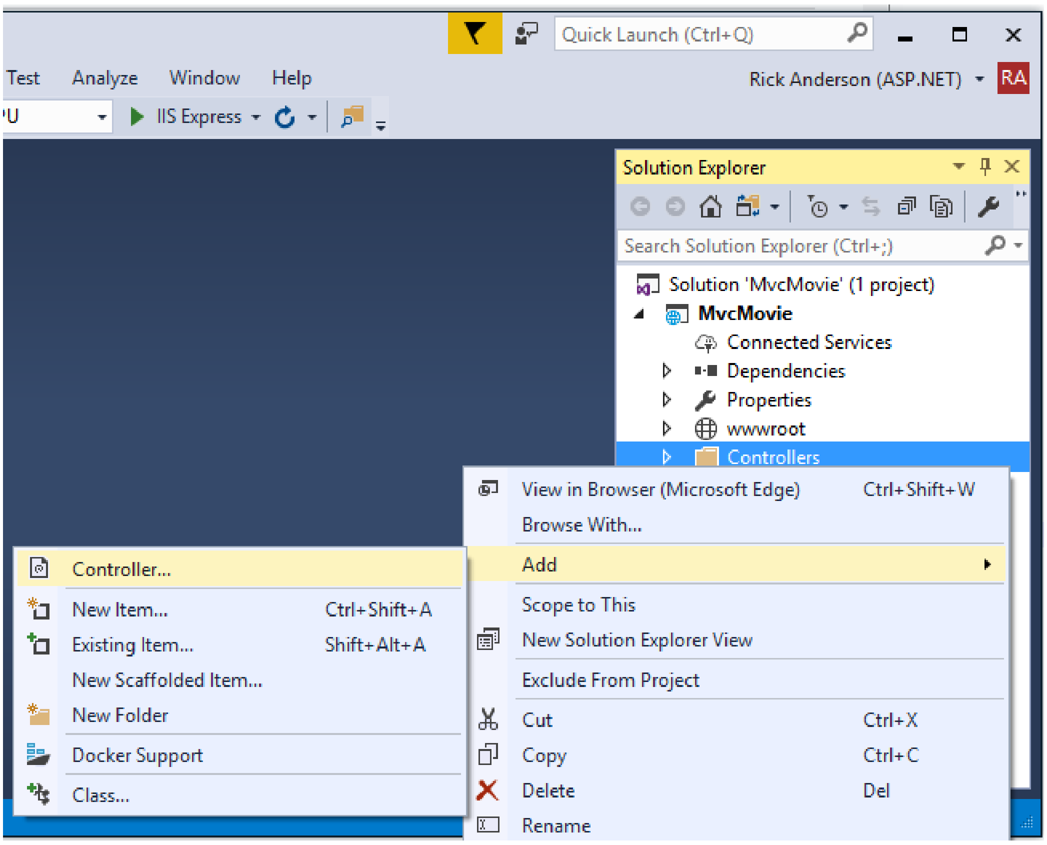
```
using System;

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; }
    }
}
```

The `ID` field is required by the database for the primary key.

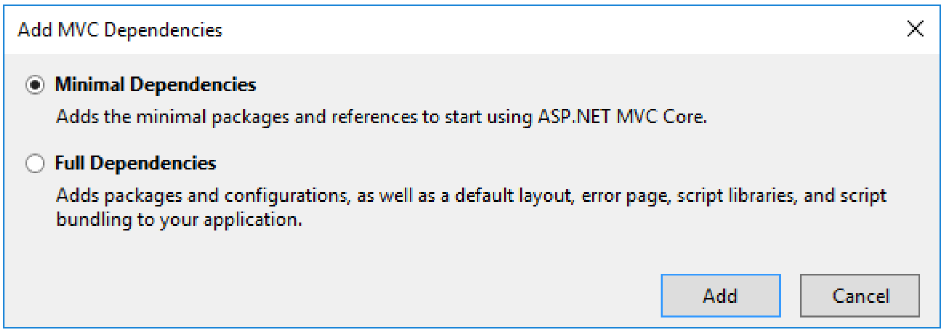
Build the project to verify you don't have any errors, and you've finally added a **M**odel to your **M**VC app.

Scaffolding a controller

In **Solution Explorer**, right-click the *Controllers* folder **> Add > Controller**.

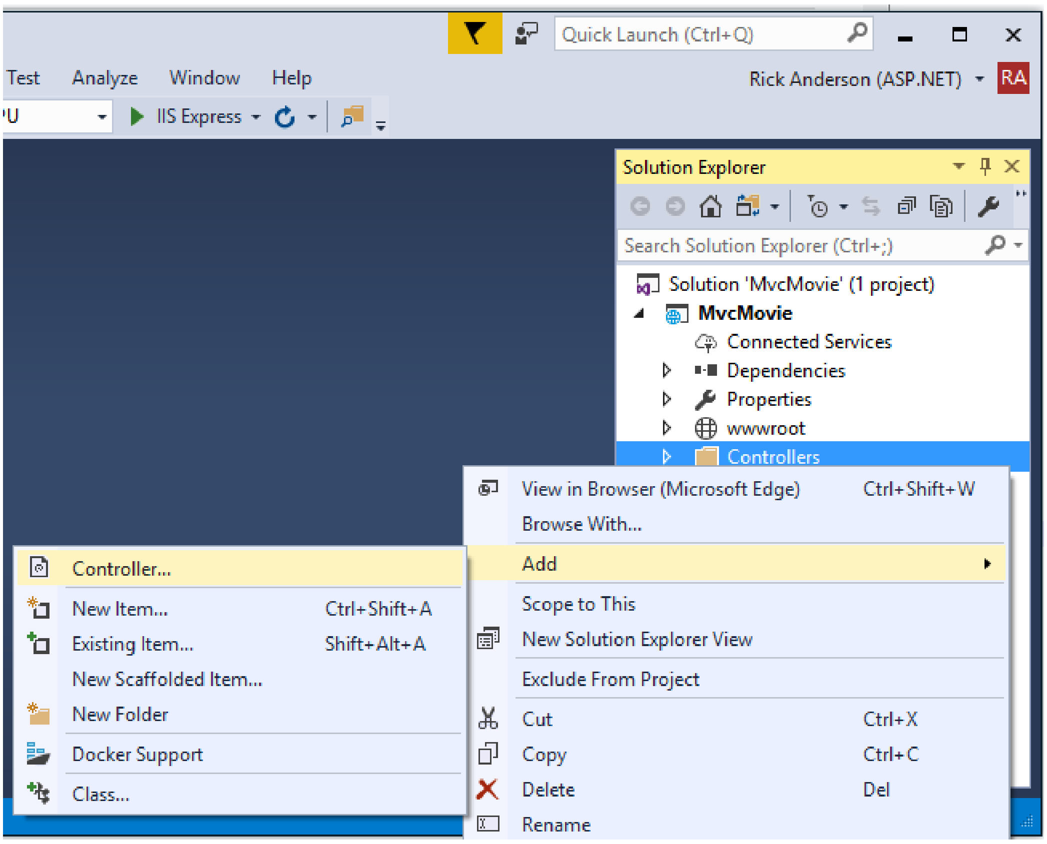


In the **Add MVC Dependencies** dialog, select **Minimal Dependencies**, and select **Add**.

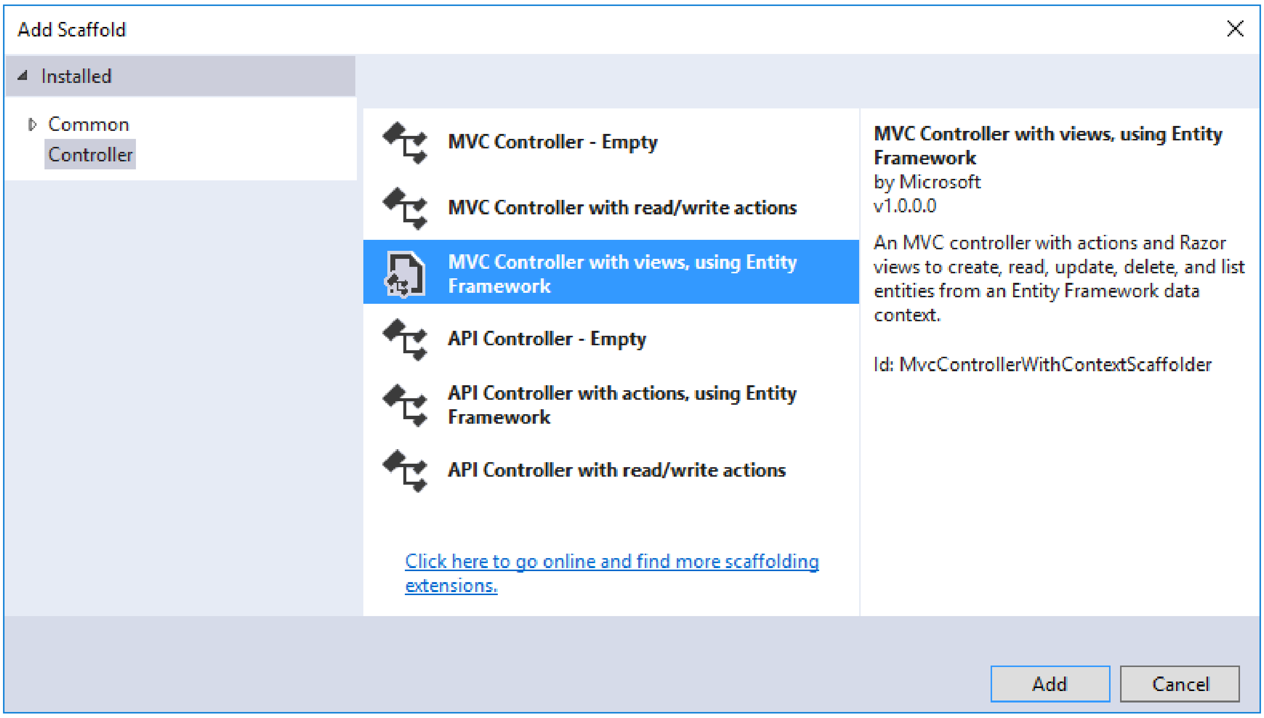


Visual Studio adds the dependencies needed to scaffold a controller, but the controller itself is not created. The next invoke of **> Add > Controller** creates the controller.

In **Solution Explorer**, right-click the *Controllers* folder **> Add > Controller**.

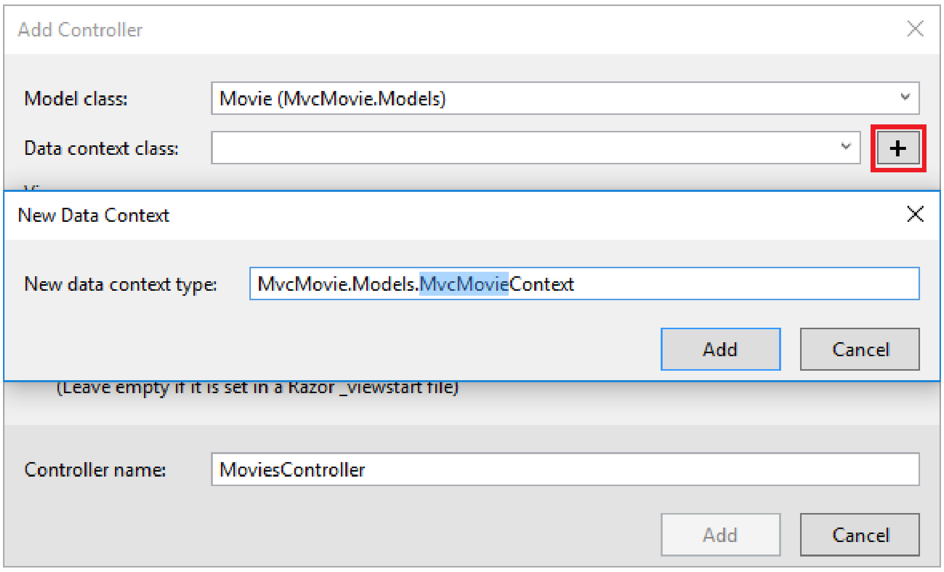


In the **Add Scaffold** dialog, tap **MVC Controller with views, using Entity Framework > Add**.

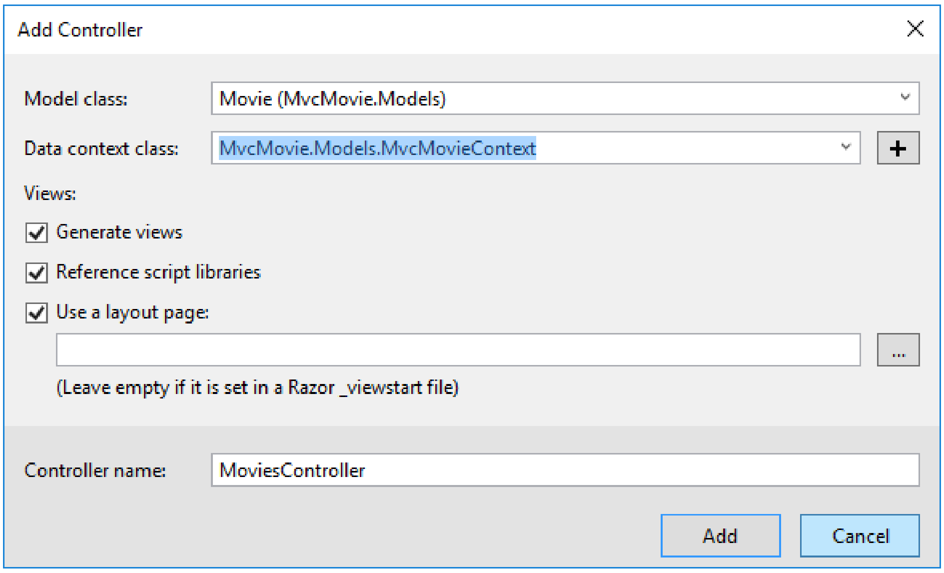


Complete the **Add Controller** dialog:

Model class: *Movie (MvcMovie.Models)*
Data context class: Select the **+** icon and add the default **MvcMovie.Models.MvcMovieContext**



Views: Keep the default of each option checked
Controller name: Keep the default *MoviesController*
Tap **Add**



Visual Studio creates:

An Entity Framework Core database context class (*Models/MvcMovieContext*)
A movies controller (*Controllers/MoviesController.cs*)
Razor view files for Create, Delete, Details, Edit and Index pages (*Views/Movies/*\.cshtml*)

The automatic creation of the database context and CRUD (create, read, update, and delete) action methods and views is known as *scaffolding*. You'll soon have a fully functional web application that lets you manage a movie database.

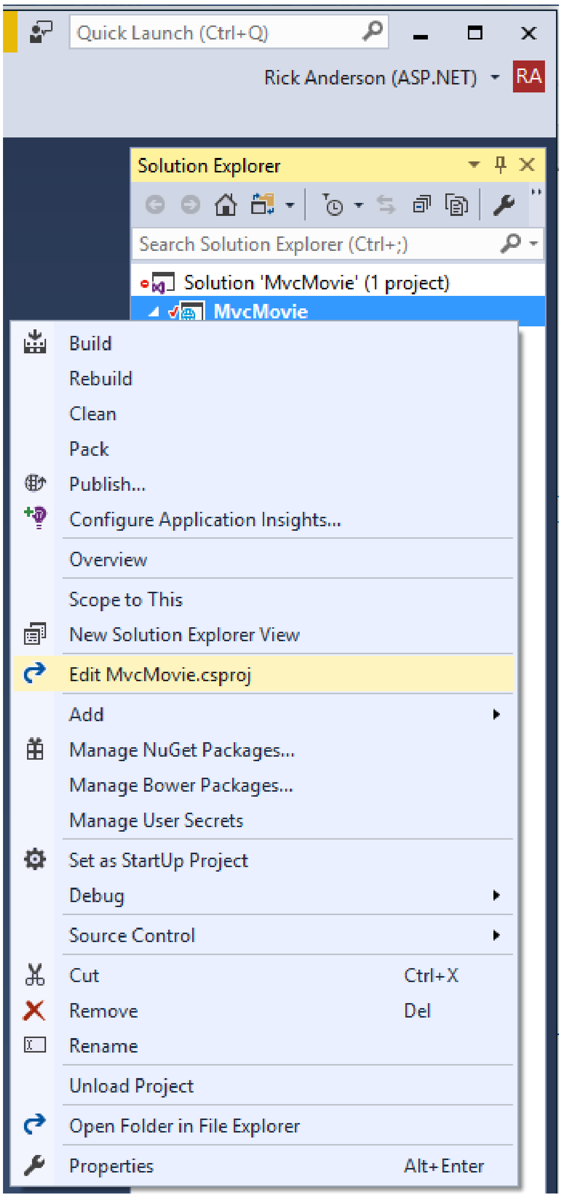
If you run the app and click on the **Mvc Movie** link, you'll get an error similar to the following:

```
An unhandled exception occurred while processing the request.
SqlException: Cannot open database "MvcMovieContext-<GUID removed>"
requested by the login. The login failed.
Login failed for user Rick
```

You need to create the database, and you'll use the EF Core Migrations feature to do that. Migrations lets you create a database that matches your data model and update the database schema when your data model changes.

Add EF tooling for Migrations

In Solution Explorer, right click the **MvcMovie** project > **Edit MvcMovie.csproj**.



Add the `"Microsoft.EntityFrameworkCore.Tools.DotNet"` NuGet package:

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="1.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
</ItemGroup>
```

Note: The version numbers shown above were correct at the time of writing.

Save your changes.

Add initial migration and update the database

Open a command prompt and navigate to the project directory. (The directory containing the *Startup.cs* file).

Run the following commands in the command prompt:

```
dotnet restore
dotnet ef migrations add Initial
dotnet ef database update
```

.NET Core is a cross-platform implementation of .NET. Here is what these commands do:

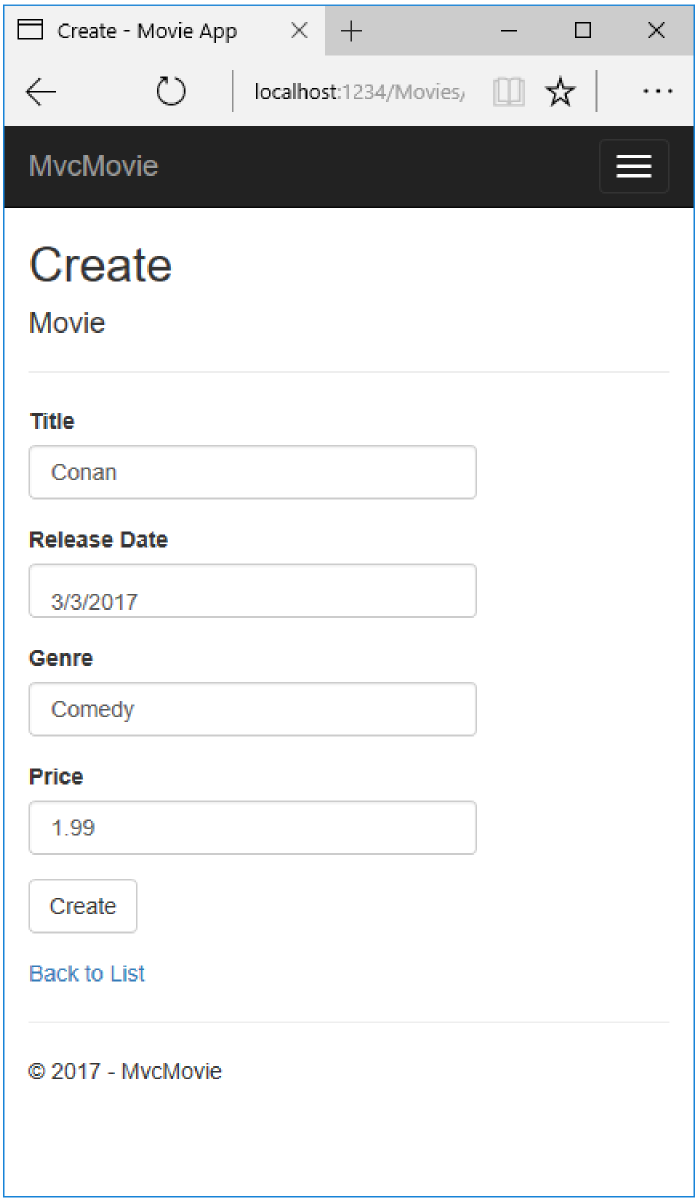
`dotnet restore`: Downloads the NuGet packages specified in the *.csproj* file.
`dotnet ef migrations add Initial` Runs the Entity Framework .NET Core CLI migrations command and creates the initial migration. The parameter after "add" is a name that you assign to the migration. Here you're naming the migration "Initial" because it's the initial database migration. This operation creates the *Data/Migrations/<date-time>_Initial.cs* file containing the migration commands to add the *Movie* table to the database.
`dotnet ef database update` Updates the database with the migration we just created.

You'll learn about the database and connection string in the next tutorial. You'll learn about data model changes in the Add a field tutorial.

Test the app

Run the app and tap the **Mvc Movie** link.

Tap the **Create New** link and create a movie.



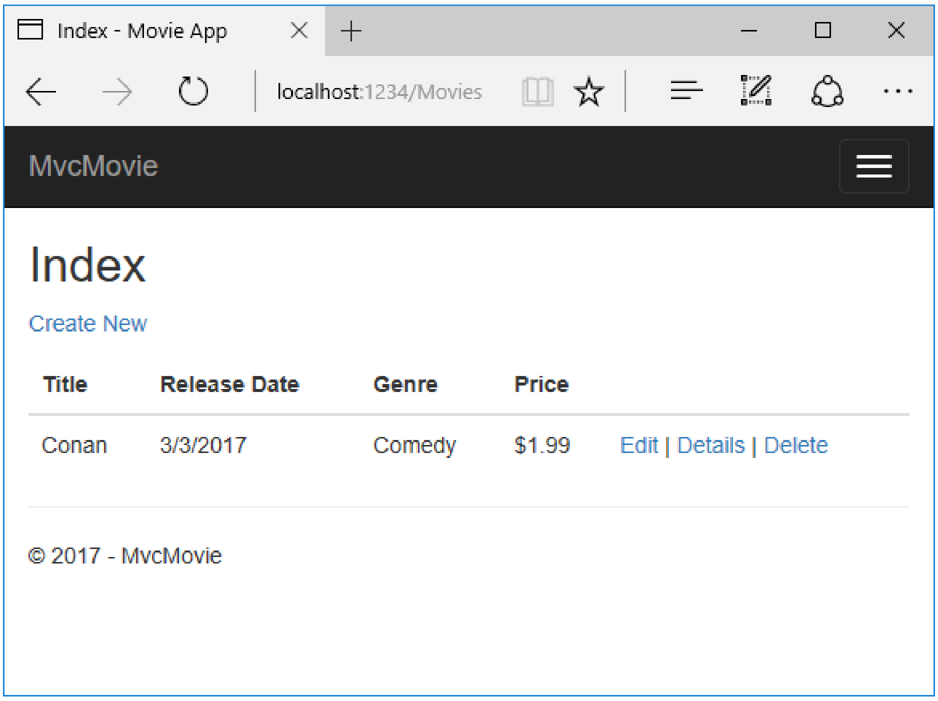
You may not be able to enter decimal points or commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See Additional resources for more information. For now, just enter whole numbers like 10.

In some locales you'll need to specify the date format. See the highlighted code below.

```
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; }
    }
}
```

We'll talk about `DataAnnotations` later in the tutorial.

Tapping **Create** causes the form to be posted to the server, where the movie information is saved in a database. You are then redirected to the */Movies* URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Dependency Injection

Open the *Startup.cs* file and examine `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
```

The highlighted code above shows the movie database context being added to the Dependency Injection container. The line following `services.AddDbContext<MvcMovieContext>(options =>` is not shown (see your code). It specifies the database to use and the connection string. `=>` is a lambda operator.

Open the *Controllers/MoviesController.cs* file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views when it created the methods and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
// 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);
}
```

The `id` parameter is generally passed as route data. For example `http://localhost:5000/movies/details/1` sets:

The controller to the `movies` controller (the first URL segment).
The action to `details` (the second URL segment).
The id to 1 (the last URL segment).

You could also pass in the `id` with a query string as follows:

`http://localhost:1234/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in case an ID value is not provided.

A lambda expression is passed in to `SingleOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .SingleOrDefaultAsync(m => m.ID == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

```
@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>
```

By including a `@model` statement at the top of the view file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Details.cshtml* file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

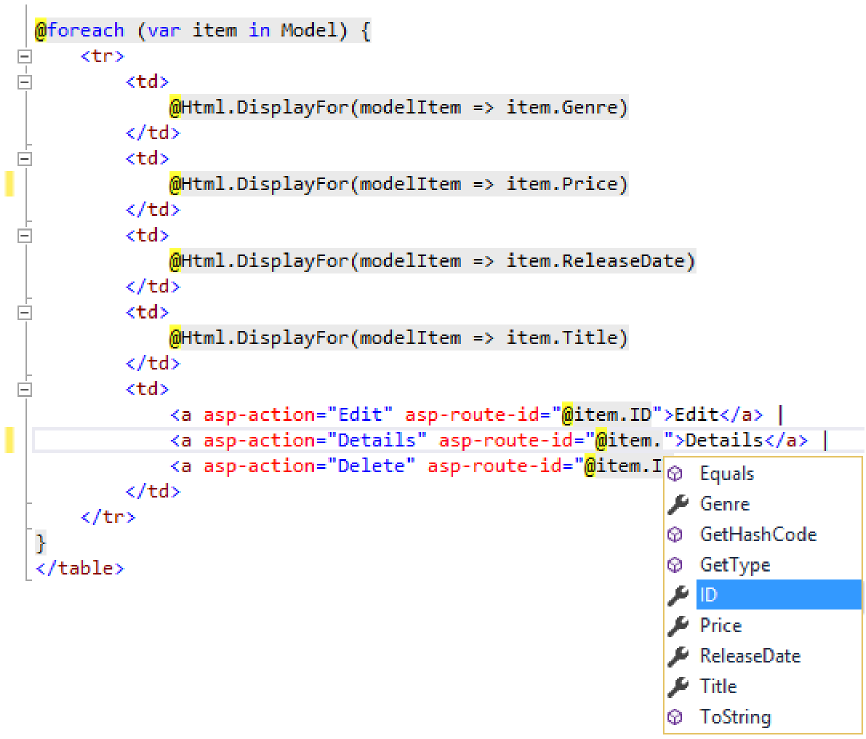
When you created the movies controller, scaffolding automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```
@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>
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full IntelliSense support in the code editor:



Additional resources

Tag Helpers
Globalization and localization

PREVIOUS ADDING A VIEW

NEXT WORKING WITH SQL

Working with SQL Server LocalDB

By Rick Anderson

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in the *Startup.cs* file:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddDbContext<MvcMovieContext>(options =>
```

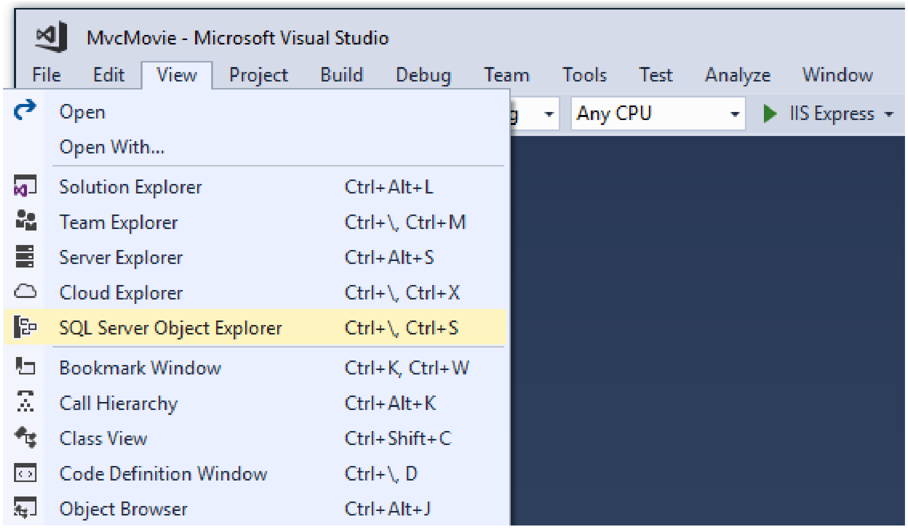
The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See Configuration for more information.

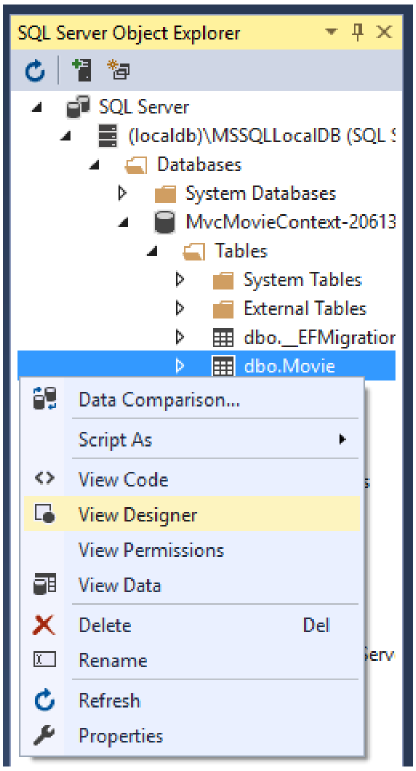
SQL Server Express LocalDB

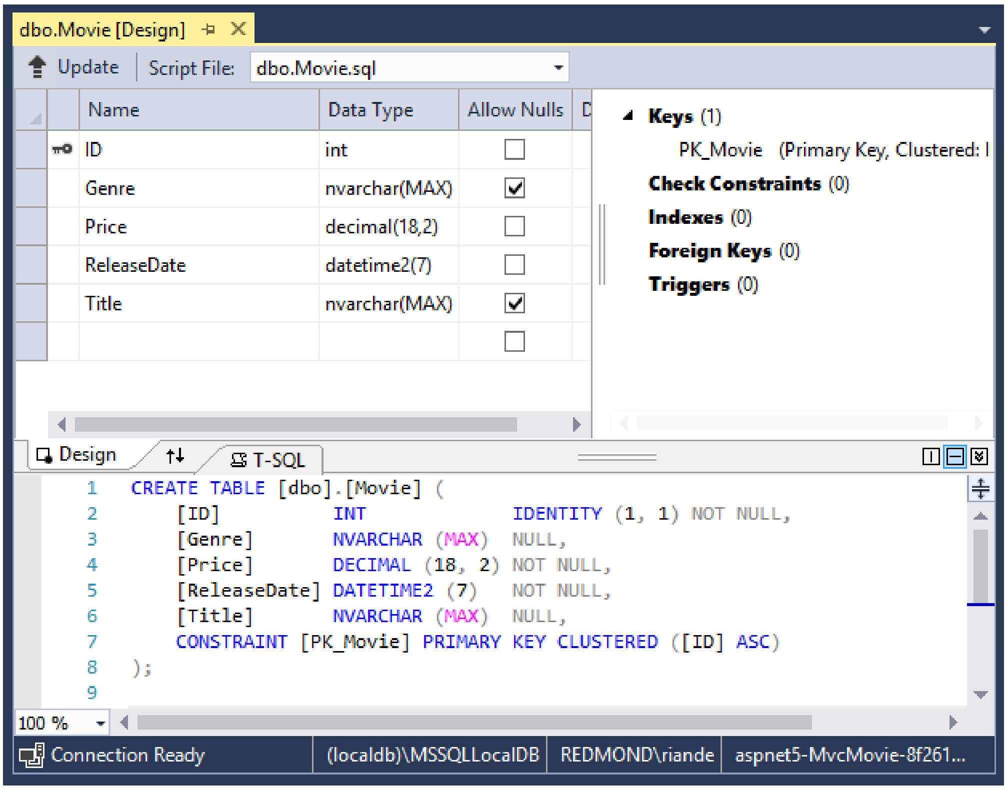
LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates "*.mdf" files in the *C:/Users/<user>* directory.

From the **View** menu, open **SQL Server Object Explorer** (SSOX).



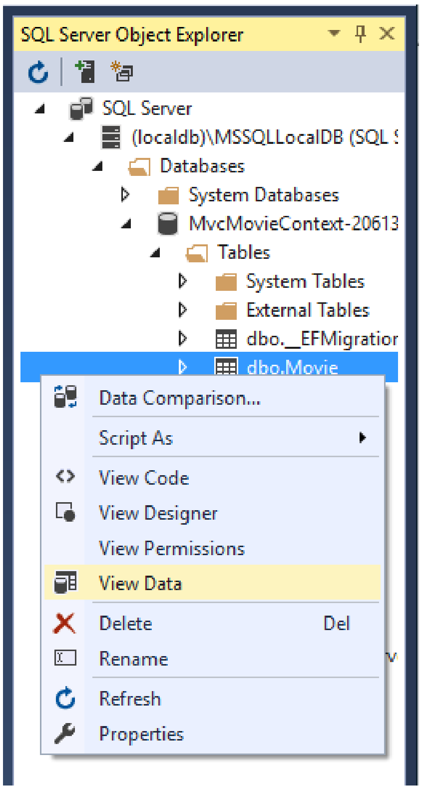
Right click on the `Movie` table **> View Designer**

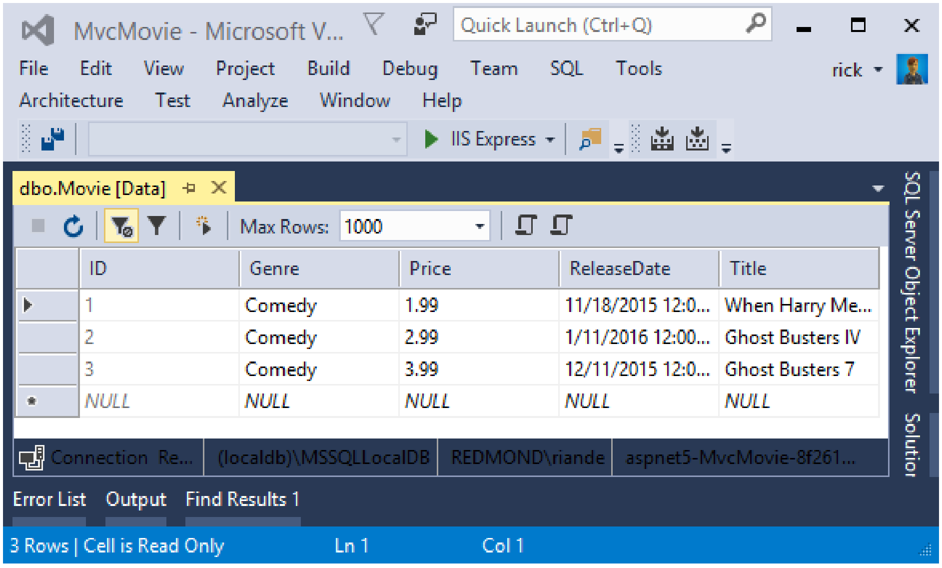




Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

Right click on the `Movie` table **> View Data**





Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
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();
            }
        }
    }
}
```

Notice if there are any movies in the DB, the seed initializer returns.

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

Add the seed initializer to the end of the `Configure` method in the *Startup.cs* file:

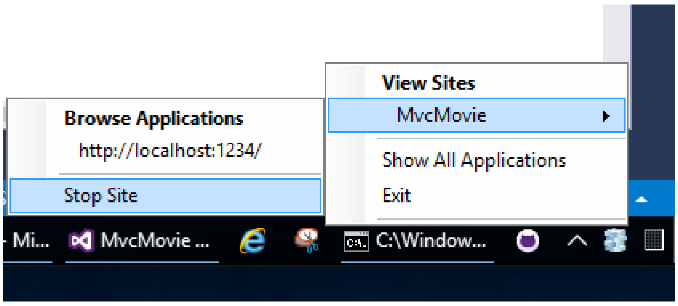
```
            app.UseStaticFiles();
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });

            SeedData.Initialize(app.ApplicationServices);
        }
    }
}
```

Test the app

Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.

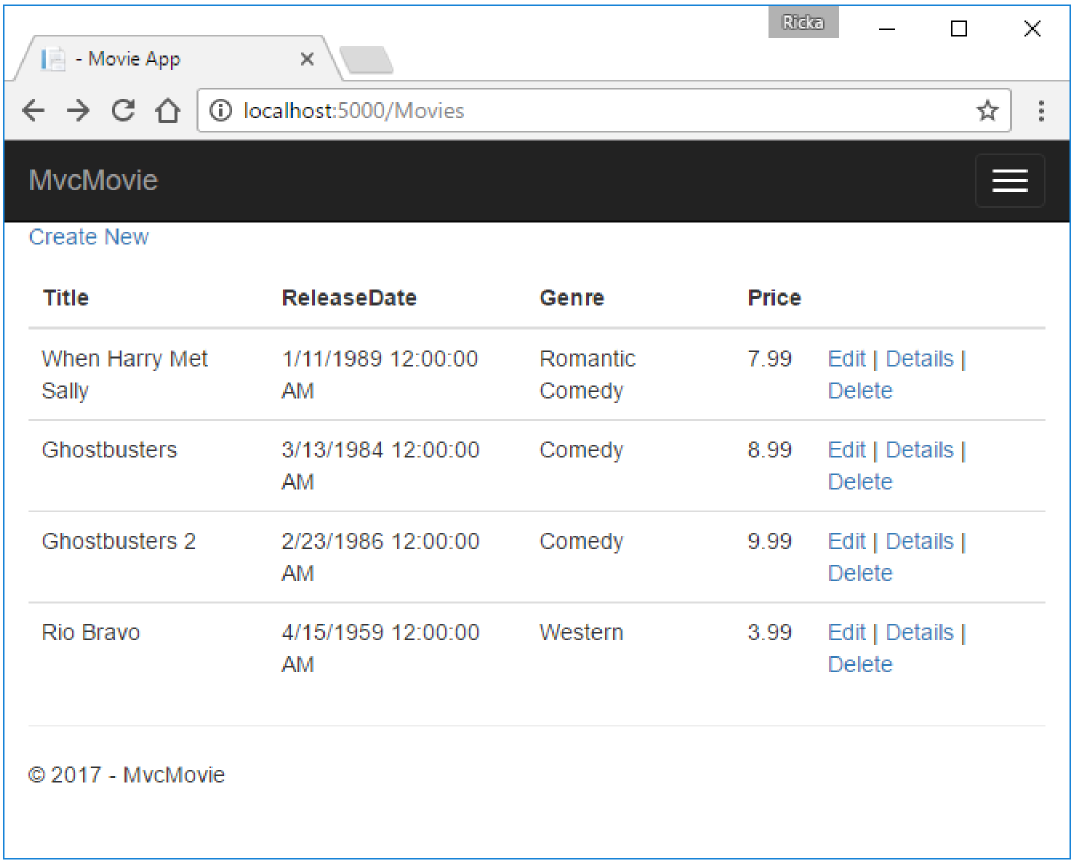
Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



If you were running VS in non-debug mode, press F5 to run in debug mode
If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.

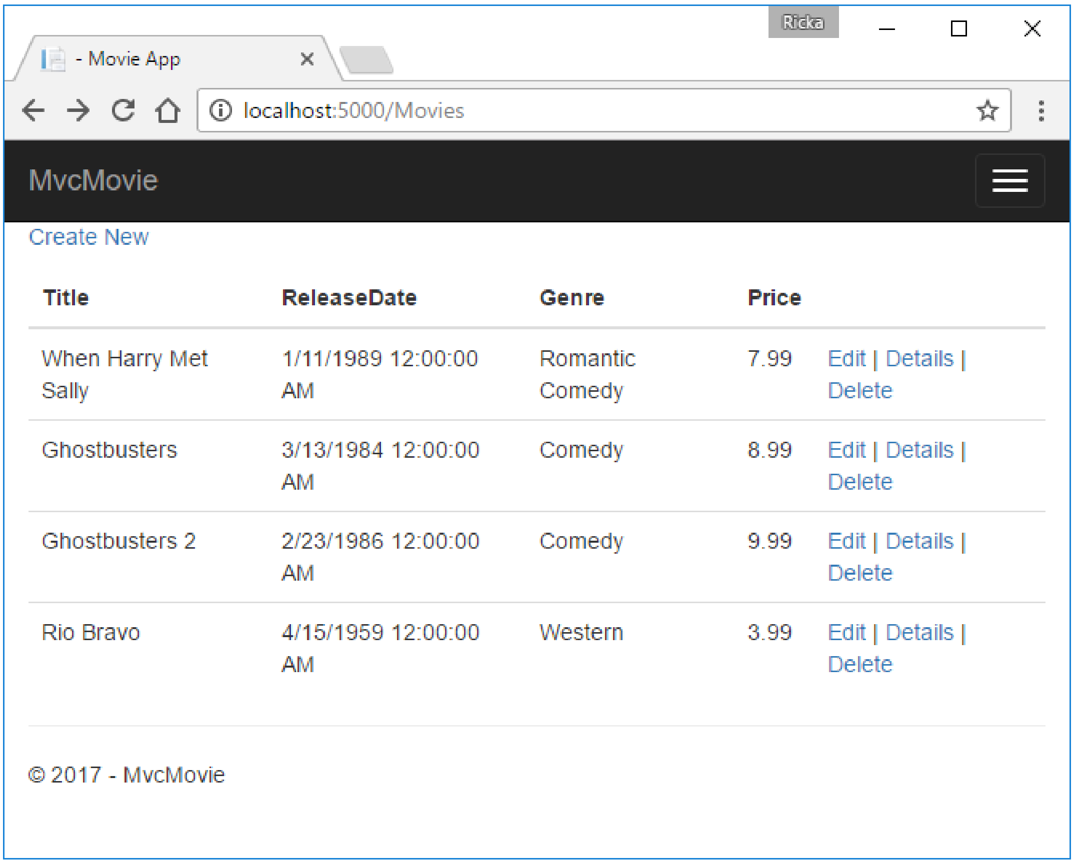


PREVIOUS 下一节

Controller methods and views

By Rick Anderson

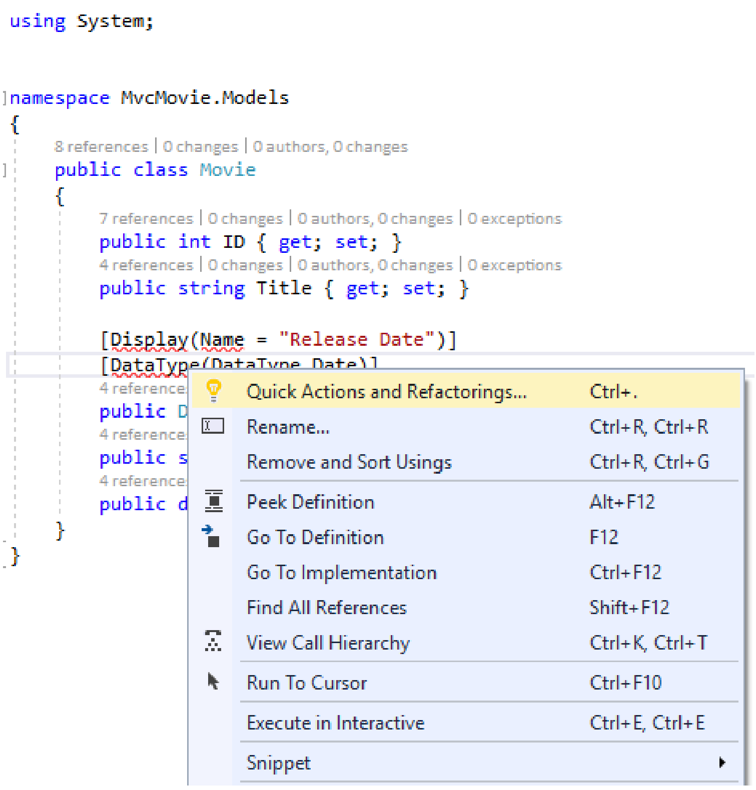
We have a good start to the movie app, but the presentation is not ideal. We don't want to see the time (12:00:00 AM in the image below) and **ReleaseDate** should be two words.



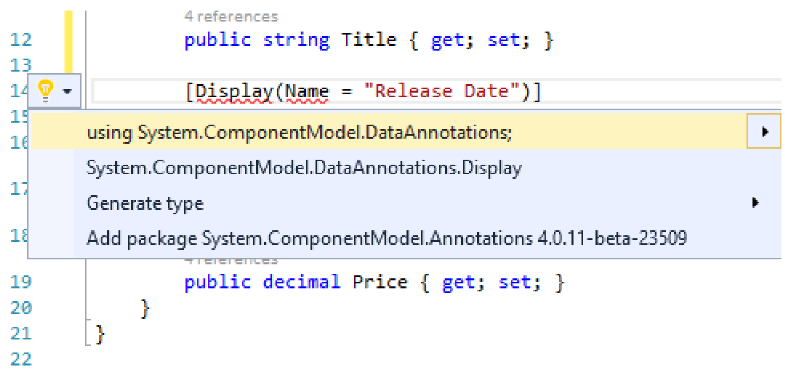
Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```
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; }
    }
}
```

Right click on a red squiggly line **> Quick Actions and Refactorings**.

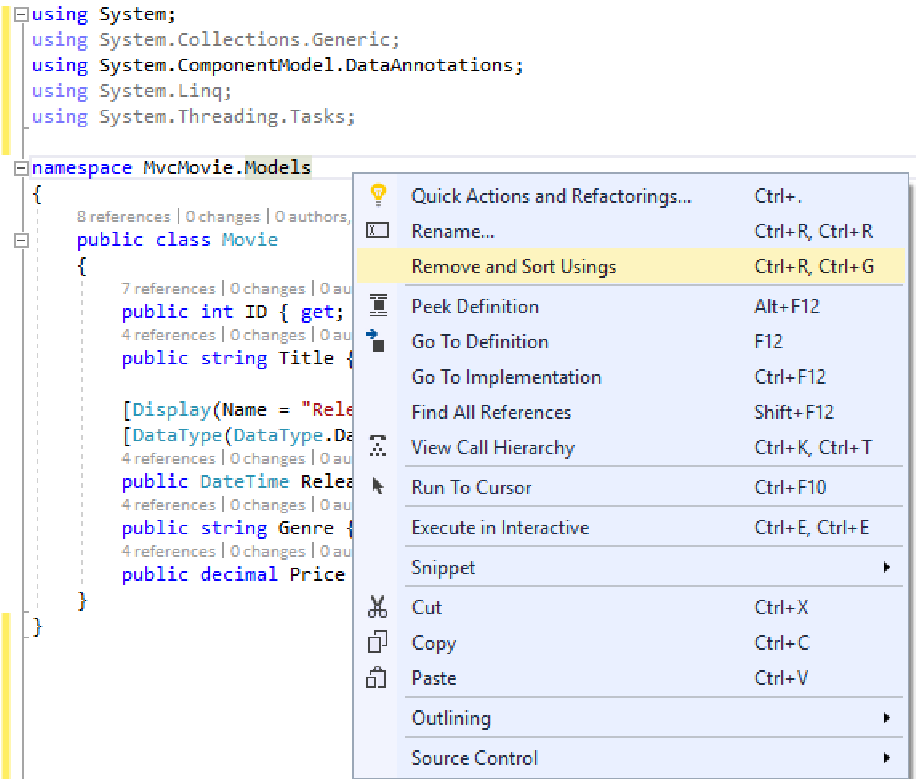


Tap `using System.ComponentModel.DataAnnotations;`



Visual studio adds `using System.ComponentModel.DataAnnotations;`.

Let's remove the `using` statements that are not needed. They show up by default in a light grey font. Right click anywhere in the *Movie.cs* file > **Remove and Sort Usings**.

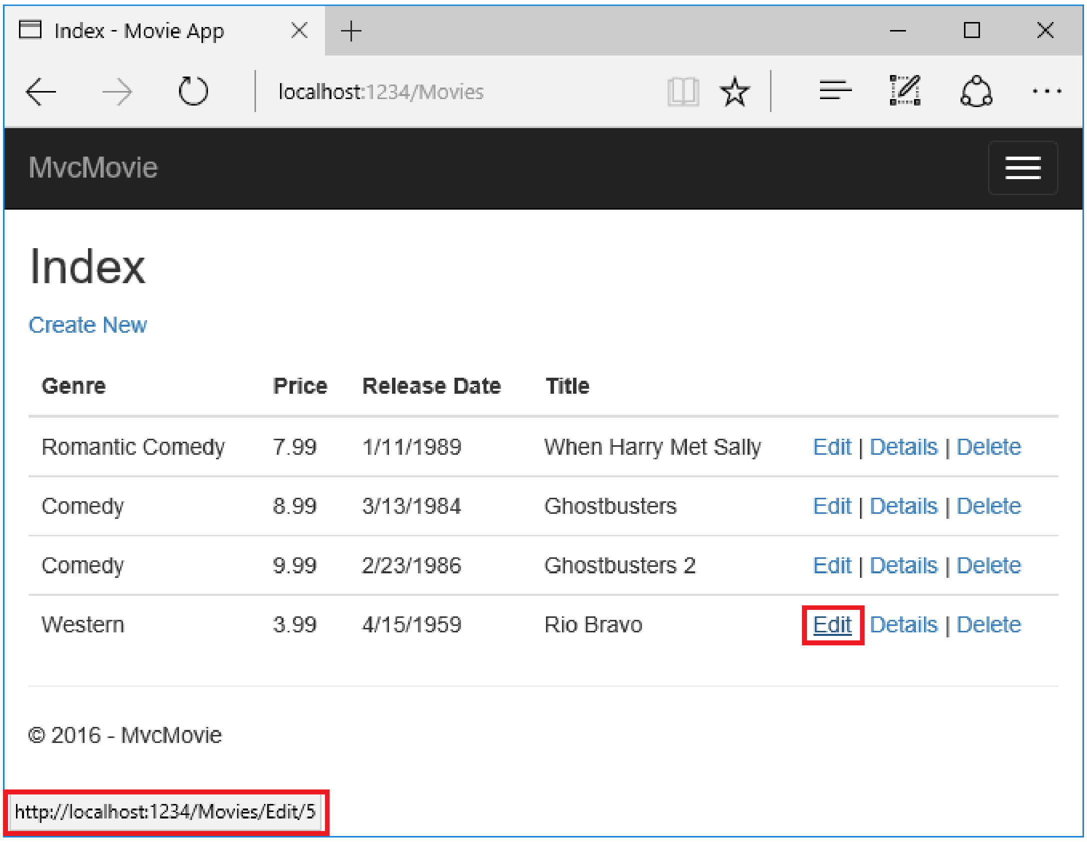


The updated code:

```
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; }
    }
}
```

We'll cover DataAnnotations in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field is not displayed.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the MVC Core Anchor Tag Helper in the *Views/Movies/Index.cshtml* file.

```
        <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>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

```
 <td>
    <a href = "/Movies/Edit/4" > Edit </ a > |
    < a href="/Movies/Details/4">Details</a> |
    <a href = "/Movies/Delete/4" > Delete </ a >
</ td >
```

Recall the format for routing set in the *Startup.cs* file:

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

ASP.NET Core translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. See Additional resources for more information.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the *Edit.cshtml* Razor file.

```
// 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);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// 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);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. See Protect your controller from over-posting for more information. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

```
// 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);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token with the Form Tag Helper.

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

The Form Tag Helper generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the Movies controller. For more information, see Anti-Request Forgery.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `SingleOrDefaultAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

```
// 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);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```
@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");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The - Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation

Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

```
<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="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

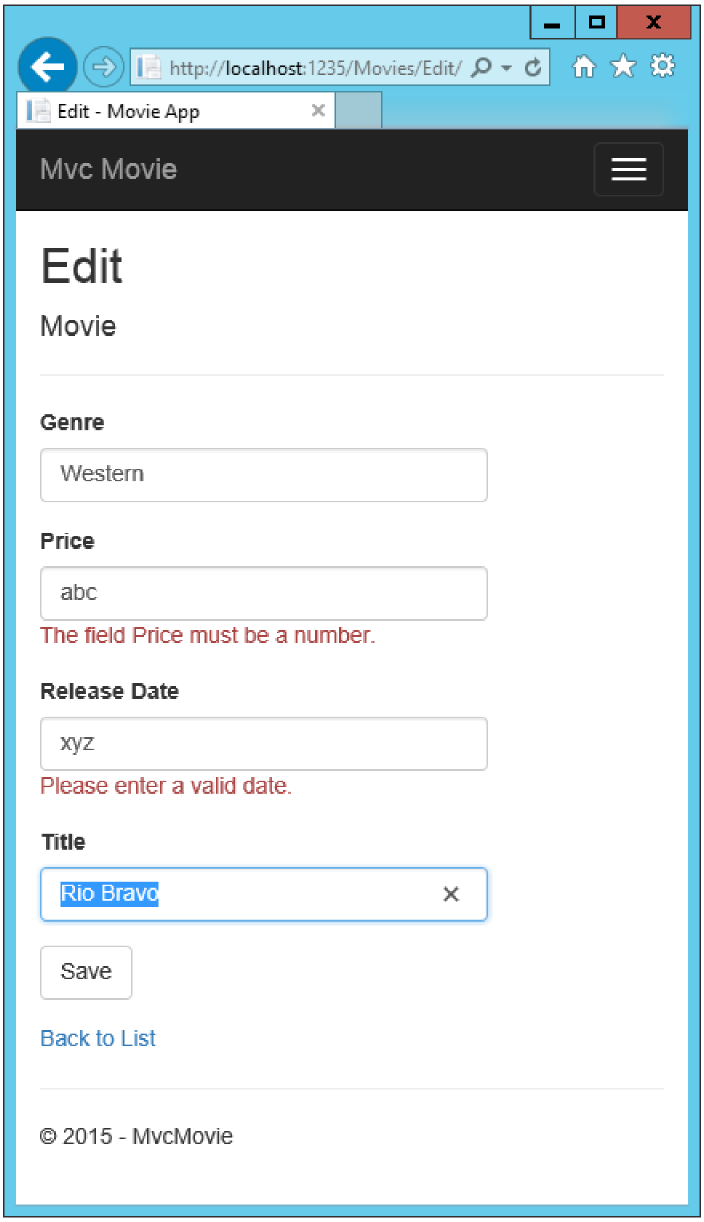
The following listing shows the `[HttpPost]` version of the `Edit` action method.

```
// 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);
}
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form is not posted. If JavaScript is disabled, you won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the *Views/Movies/Edit.cshtml* view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

Globalization and localization
Introduction to Tag Helpers
Authoring Tag Helpers
Anti-Request Forgery
Protect your controller from over-posting
ViewModels
Form Tag Helper
Input Tag Helper
Label Tag Helper
Select Tag Helper
Validation Tag Helper

Adding Search

By Rick Anderson

In this section you'll add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method with the following code:

```
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());
}
```

The first line of the `Index` action method creates a LINQ query to select the movies:

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

The query is *only* defined at this point, it has **not** been run against the database.

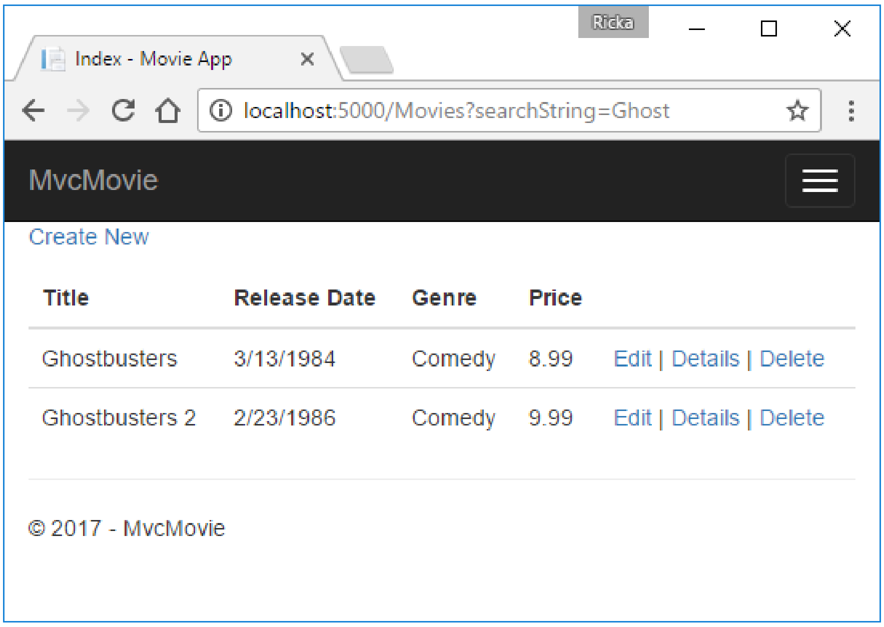
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

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

The `s => s.Title.Contains()` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where`, `Contains` or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

Note: The Contains method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, Contains maps to SQL LIKE, which is case insensitive. In SQLlite, with the default collation, it's case sensitive.

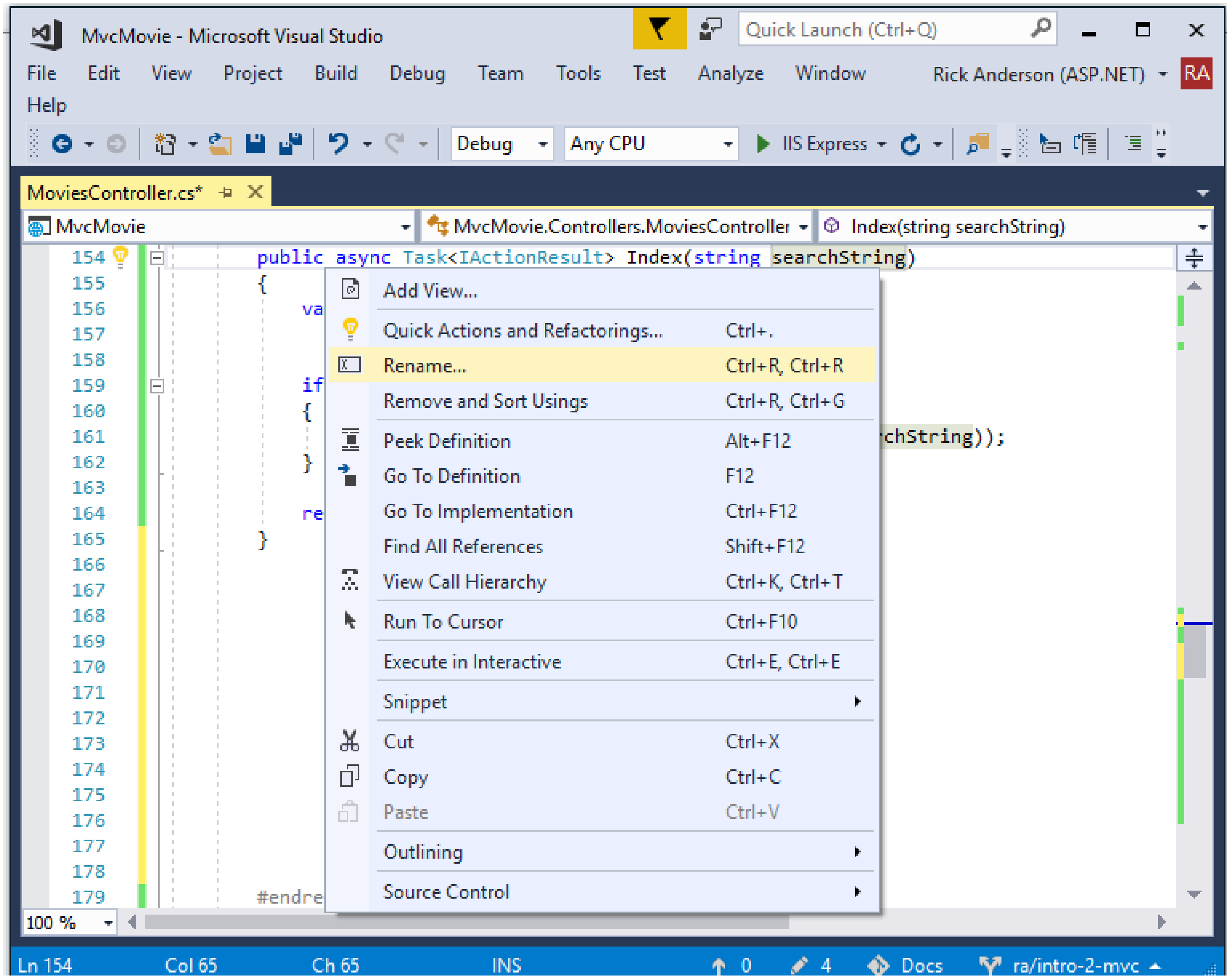
Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.

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

You can quickly rename the `searchString` parameter to `id` with the **rename** command. Right click on `searchString` > **Rename**.



The rename targets are highlighted.

```
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);
}
```

Change the parameter to `id` and all occurrences of `searchString` change to `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);
}
```

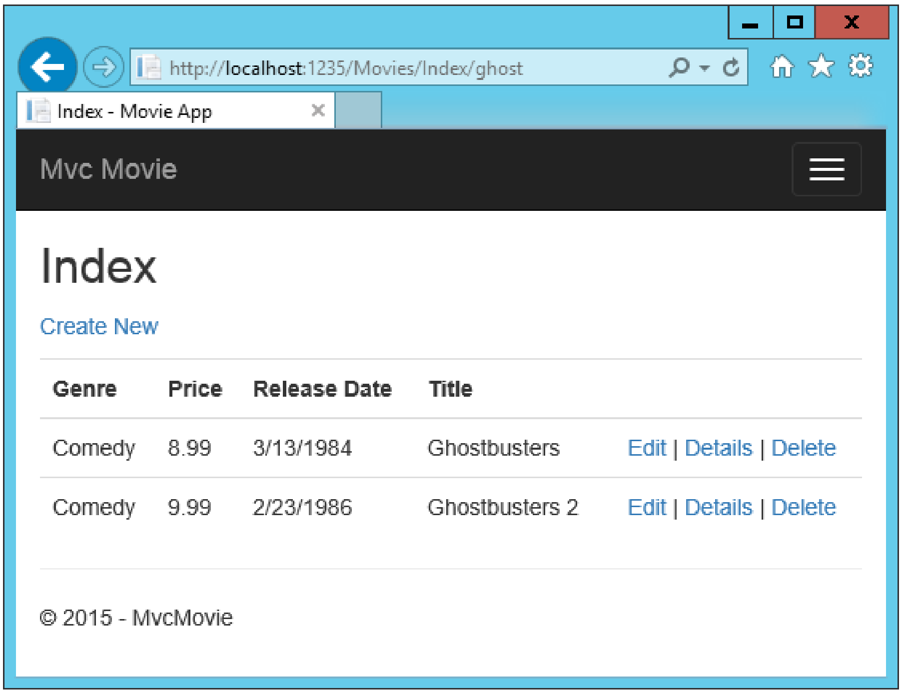
The previous `Index` method:

```
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());
}
```

The updated `Index` method with `id` parameter:

```
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());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



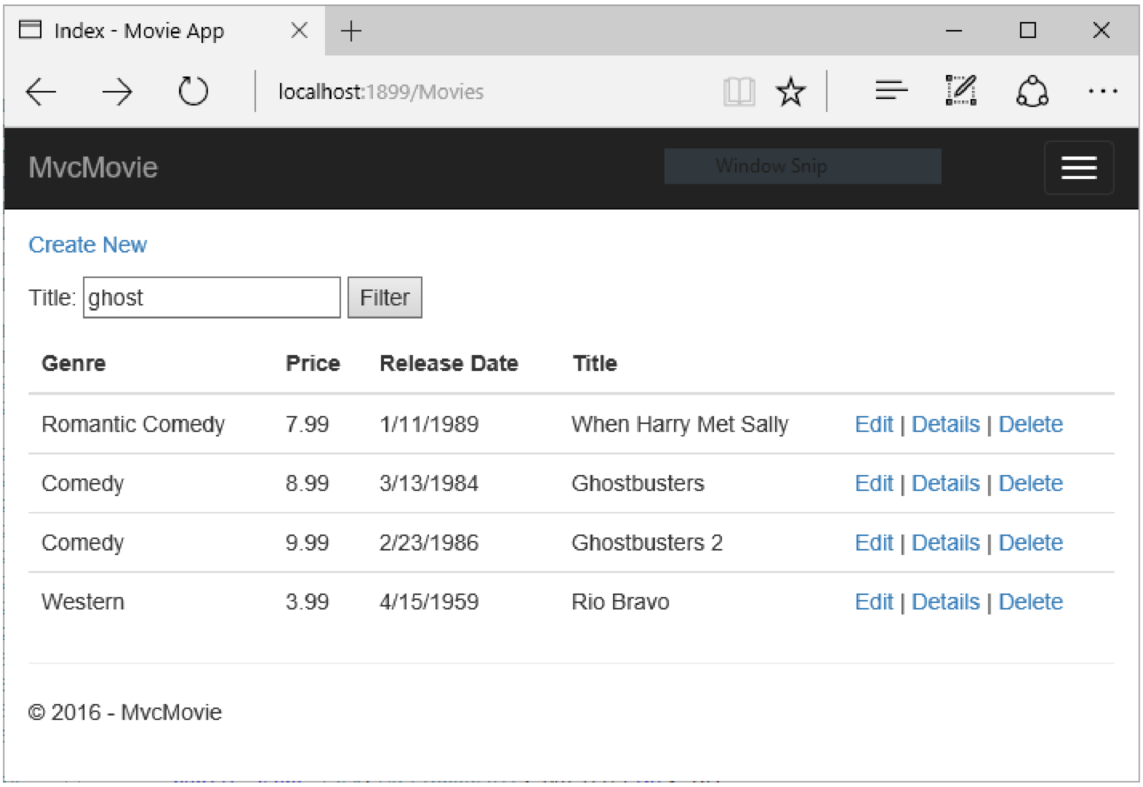
However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```
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());
}
```

Open the *Views/Movies/Index.cshtml* file, and add the `<form>` markup highlighted below:

```
    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>
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



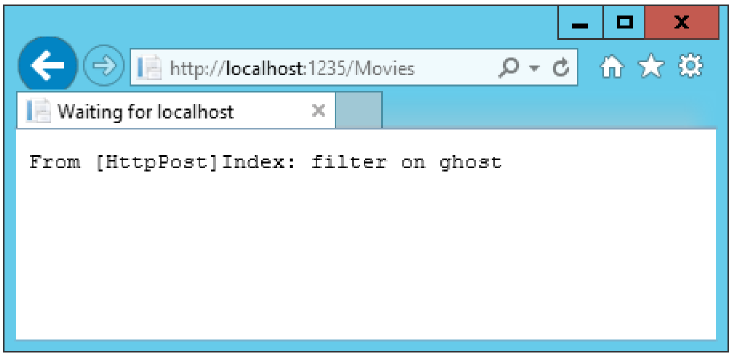
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

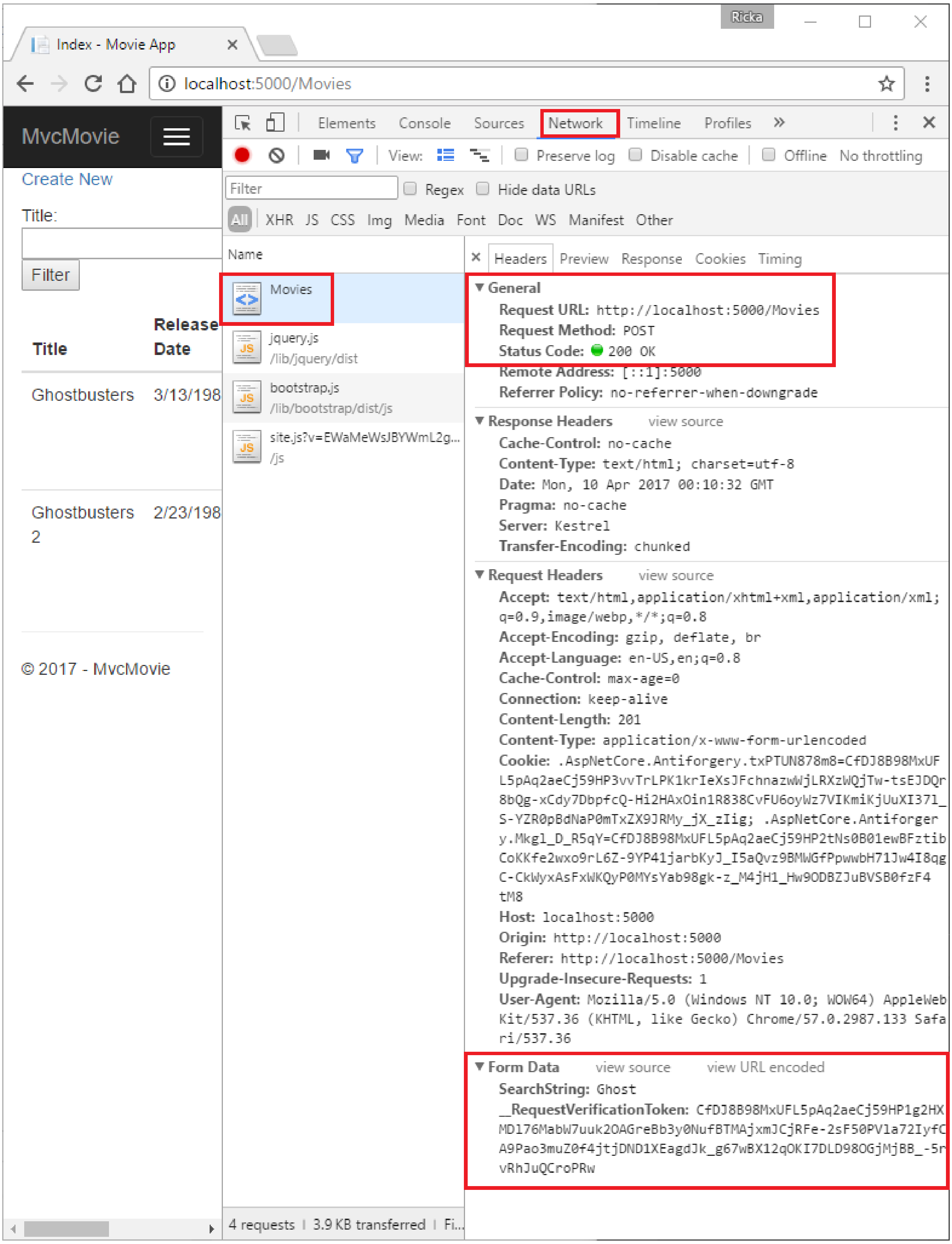
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



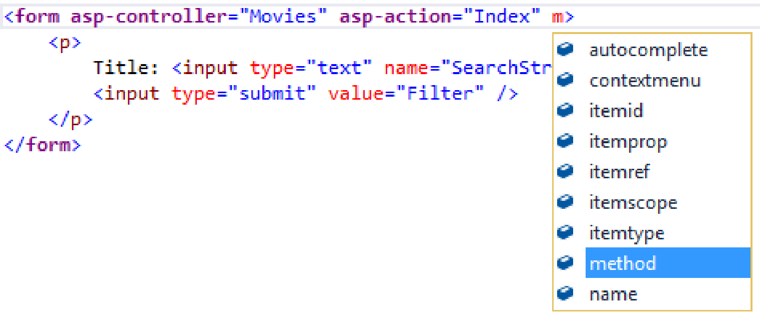
However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:

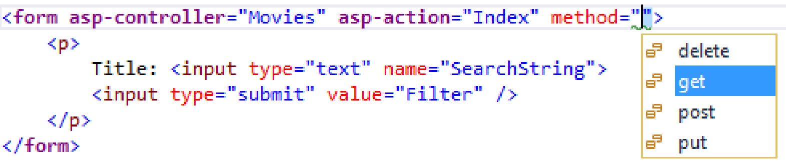


You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. We'll fix this by specifying the request should be `HTTP GET`.

Notice how intelliSense helps us update the markup.

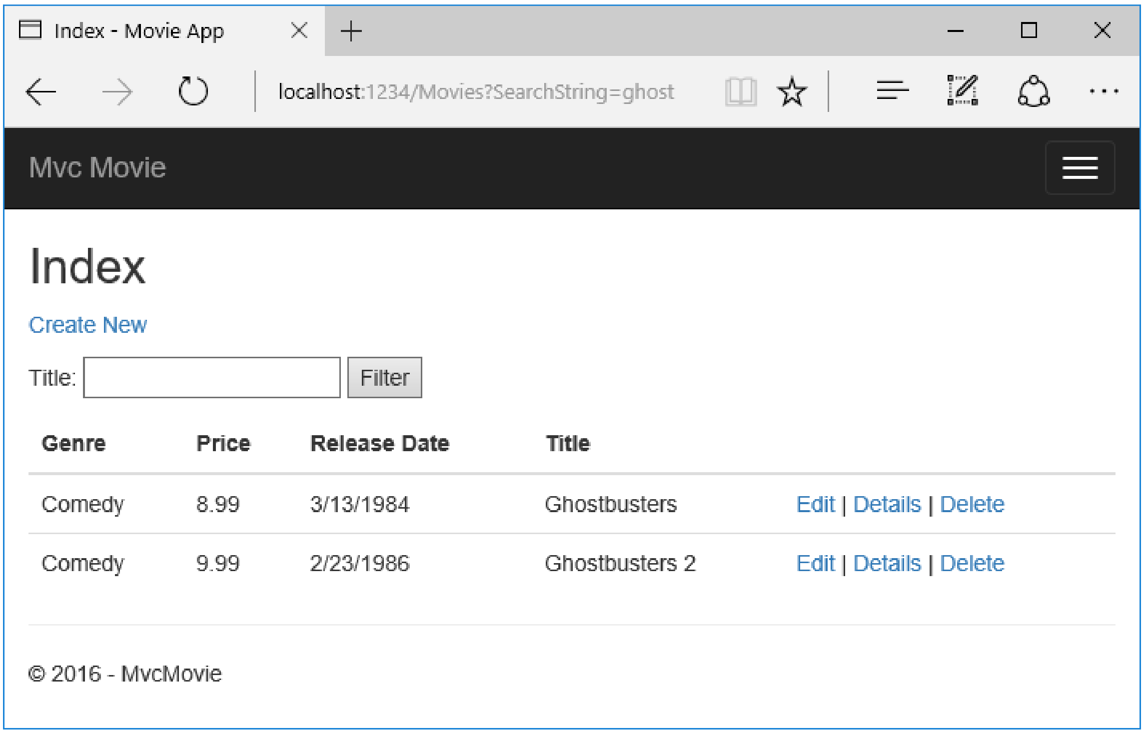




Notice the distinctive font in the `<form>` tag. That distinctive font indicates the tag is supported by Tag Helpers.

```
<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Adding Search by Genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

```
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; }
    }
}
```

The movie-genre view model will contain:

A list of movies.
A `SelectList` containing the list of genres. This will allow the user to select a genre from the list.
`movieGenre`, which contains the selected genre.

Replace the `Index` method in `MoviesController.cs` with the following code:

```
// 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);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

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

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

```
movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())
```

Adding search by genre to the Index view

Update `Index.cshtml` as follows:

```
@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>
```

Test the app by searching by genre, by movie title, and by both.

上一节 下一节

Adding a New Field

By Rick Anderson

In this section you'll use Entity Framework Code First Migrations to add a new field to the model and migrate that change to the database.

When you use EF Code First to automatically create a database, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
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; }
}
```

Build the app (Ctrl+Shift+B).

Because you've added a new field to the `Movie` class, you also need to update the binding white list so this new property will be included. In *MoviesController.cs*, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

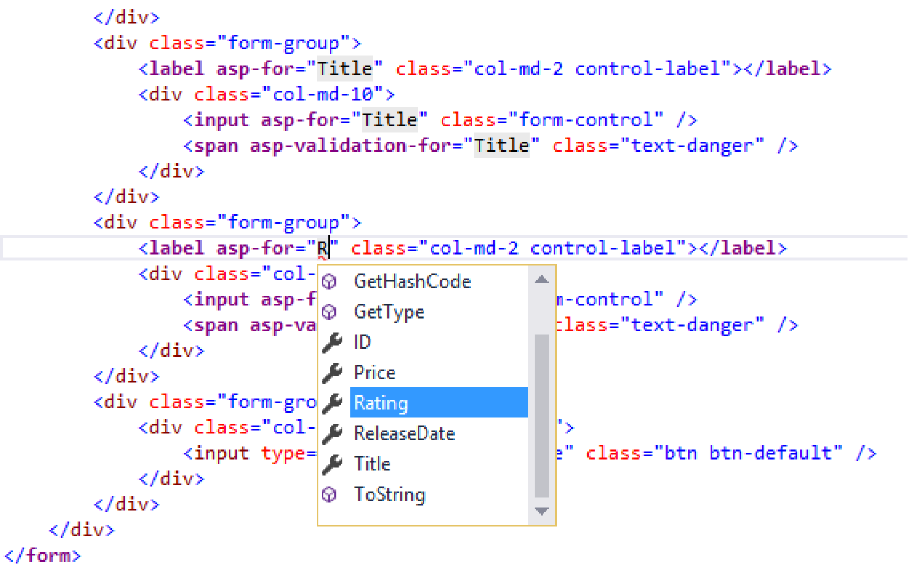
```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

You also need to update the view templates in order to display, create and edit the new `Rating` property in the browser view.

Edit the */Views/Movies/Index.cshtml* file and add a `Rating` field:

```
<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>
                @Html.DisplayNameFor(model => model.movies[0].Rating)
            </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>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
```

Update the */Views/Movies/Create.cshtml* with a `Rating` field. You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers. Note: In the RTM verison of Visual Studio 2017 you need to install the Razor Language Services for Razor intelliSense. This will be fixed in the next release.



The app won't work until we update the DB to include the new field. If you run it now, you'll get the following `SqlException`:

`SqlException: Invalid column name 'Rating'.`

You're seeing this error because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

There are a few approaches to resolving the error:

Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application.

Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

Use Code First Migrations to update the database schema.

For this tutorial, we'll use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

Build the solution then open a command prompt in the project folder. Enter the following commands:

```
dotnet ef migrations add Rating
dotnet ef database update
```

Note: If you get the error message `No executable found matching command "dotnet-ef"`:

Verify you're in the project folder (which contains the *.csproj* file).
Verify the *.csproj* file contains the "Microsoft.EntityFrameworkCore.Tools.DotNet" NuGet package.
See this blog post for help troubleshooting.

The `migrations add` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model. The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration step.

If you delete all the records in the DB, the initialize will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from SSOX.

Run the app and verify you can create/edit/display movies with a `Rating` field. You should also add the `Rating` field to the `Edit`, `Details`, and `Delete` view templates.

Adding validation

By Rick Anderson

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Adding validation rules to the movie model

Open the *Movie.cs* file. DataAnnotations provides a built-in set of validation attributes that you apply declaratively to any class or property. (It also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.)

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
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; }
}
```

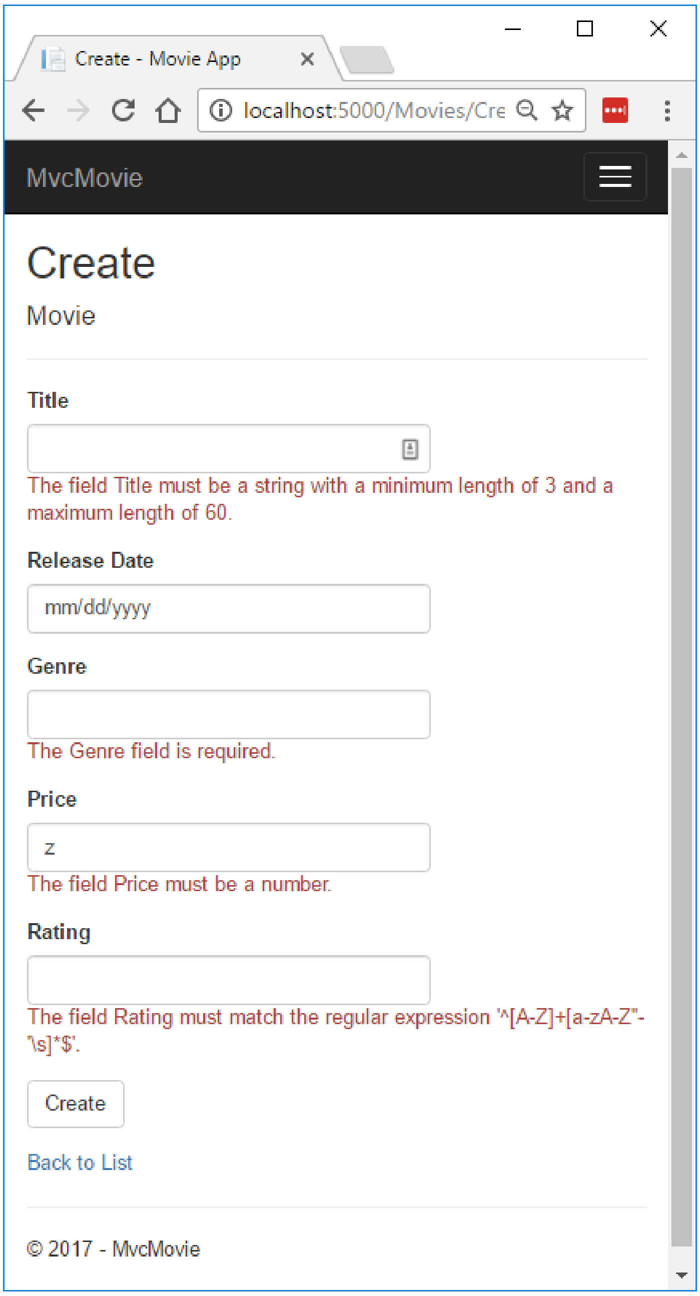
The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` and `MinimumLength` attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The `RegularExpression` attribute is used to limit what characters can be input. In the code above, `Genre` and `Rating` must use only letters (white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in MVC

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



Note

You may not be able to enter decimal points or commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See Additional resources for more information. For now, just enter whole numbers like 10.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool , or the F12 Developer tools.

How validation works

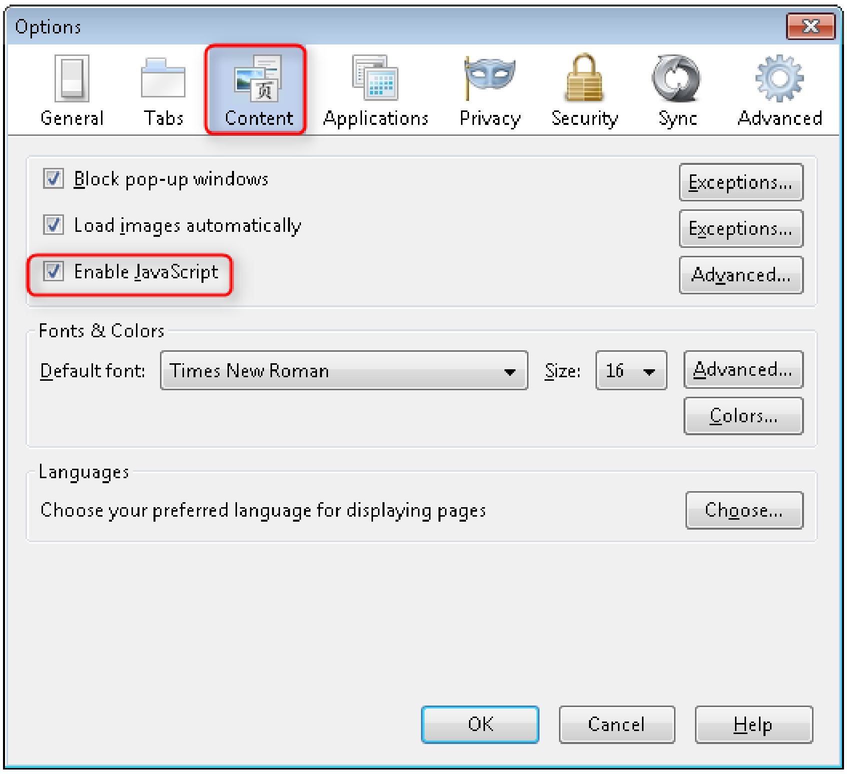
You might wonder how the validation UI was generated without any updates to the code in the controller or views. The follow code shows the two `Create` methods.

```
// 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);
}
```

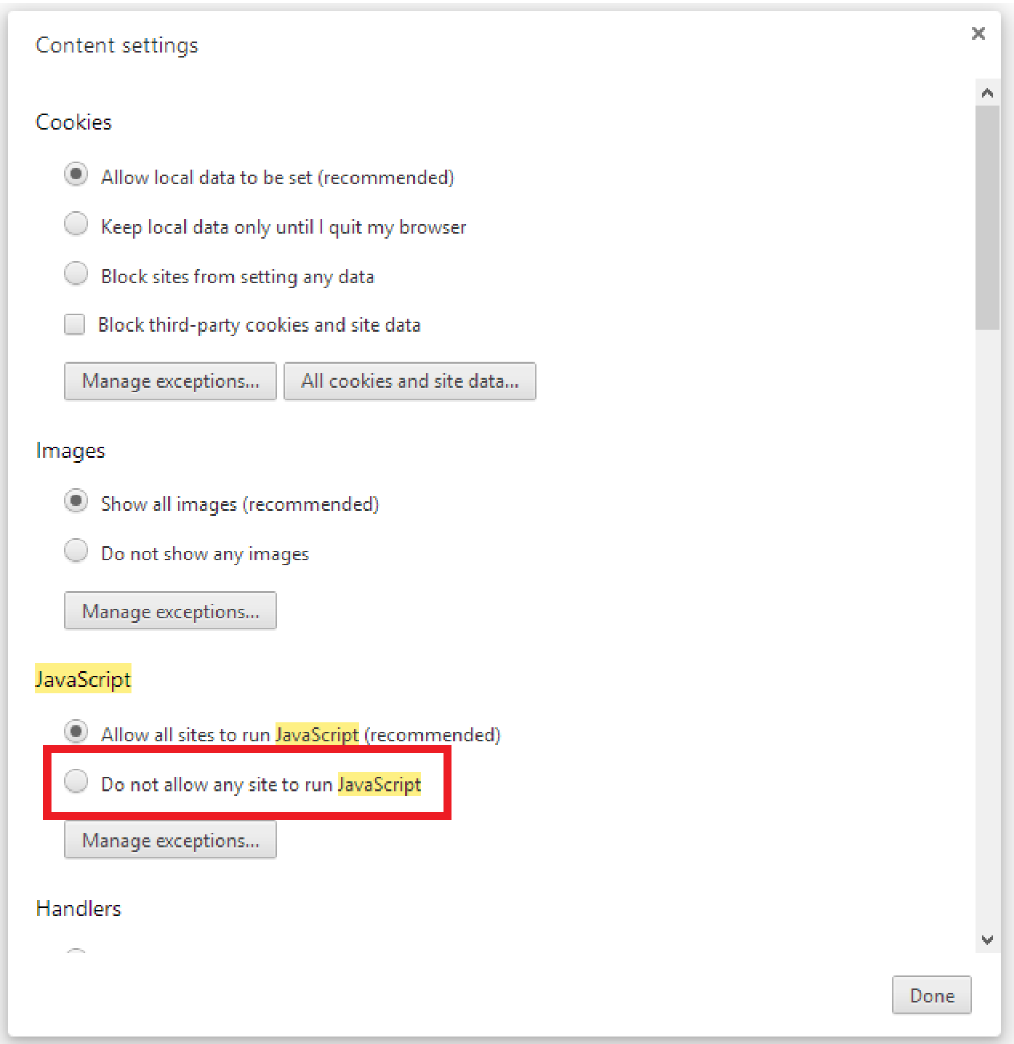
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form is not posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

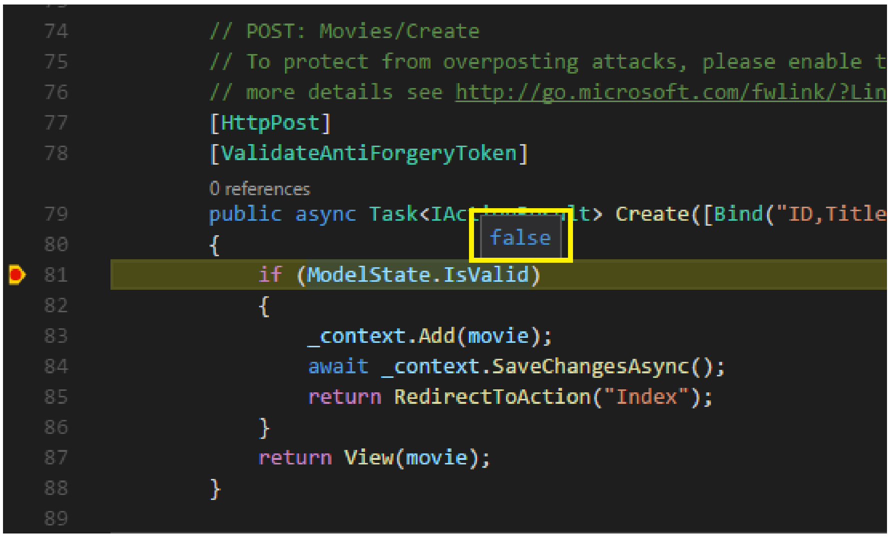
The following image shows how to disable JavaScript in the FireFox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.



Below is portion of the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
<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>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

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

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

The `DataType` attributes only provide hints for the view engine to format the data (and supply attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type, they are not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

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

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably do not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)

By default, the browser will render data using the correct format based on your locale

The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

Note

jQuery validation does not work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:

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

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
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; }
}
```

In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

Working with Forms
Globalization and localization
Introduction to Tag Helpers
Authoring Tag Helpers

PREVIOUS 下一节

Examining the Details and Delete methods

By Rick Anderson

Open the Movie controller and examine the `Details` method:

```
// 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);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method and an `id` value. Recall these segments are defined in *Startup.cs*.

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

EF makes it easy to search for data using the `SingleOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:xxxx/Movies/Details/1` to something like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you did not check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// 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");
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// 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)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

```
// POST: Movies/Delete/6
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Thanks for completing this introduction to ASP.NET Core MVC. We appreciate any comments you leave. Getting started with MVC and EF Core is an excellent follow up to this tutorial.

PREVIOUS

Create a Web API with ASP.NET Core MVC and Visual Studio for Mac

By Rick Anderson and Mike Wasson

HTTP is not just for serving up web pages. It's also a powerful platform for building APIs that expose services and data. HTTP is flexible and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop apps.

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

ASP.NET Core has built-in support for MVC creating Web APIs.

There are 3 versions of this tutorial:

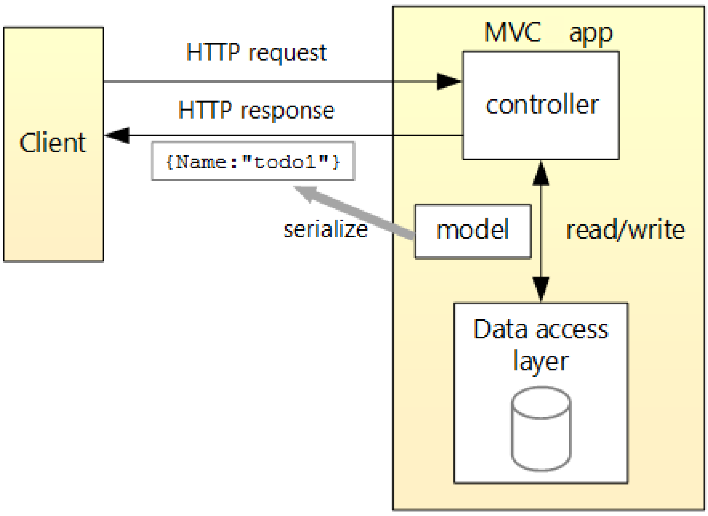
macOS: Web API with Visual Studio for Mac
Windows: Web API with Visual Studio for Windows
macOS, Linux, Windows: Web API with Visual Studio Code

Overview

Here is the API that you'll create:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



The client is whatever consumes the web API (mobile app, browser, etc). We aren't writing a client in this tutorial. We'll use

Postman or curl to test the app.

A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as C# classes, also know as **P**lain **O**ld **C#** **O**bject (POCOs).

A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.

To keep the tutorial simple, the app doesn't use a persistent database. Instead, it stores to-do items in an in-memory database.

See Introduction to ASP.NET Core MVC on Mac or Linux for an example that uses a persistent database.

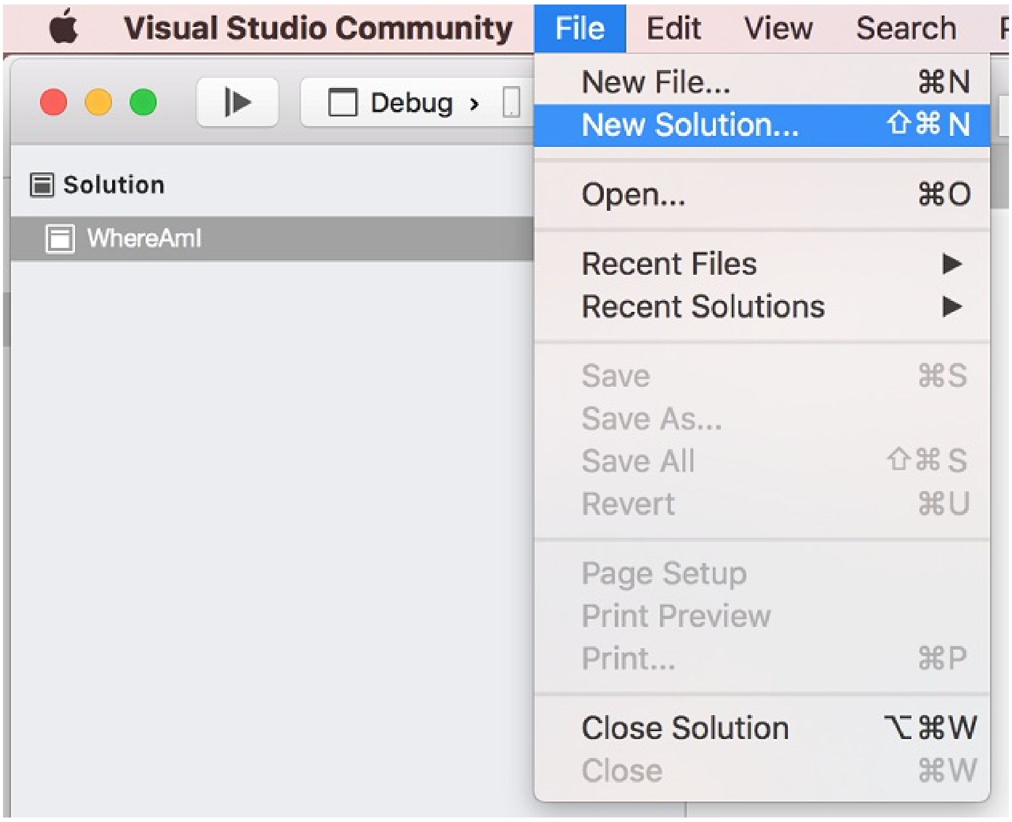
Prerequisites

Install the following:

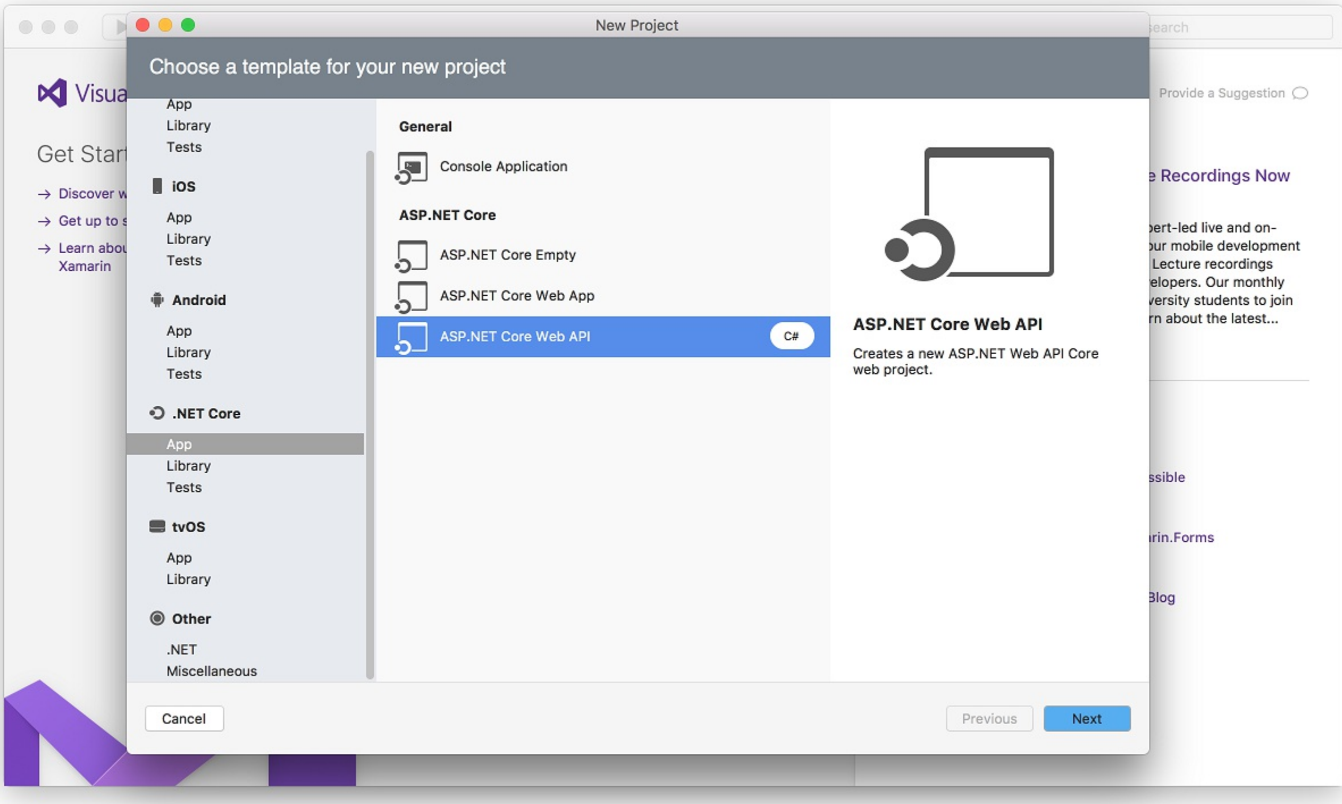
.NET Core SDK
Visual Studio for Mac

Create the project

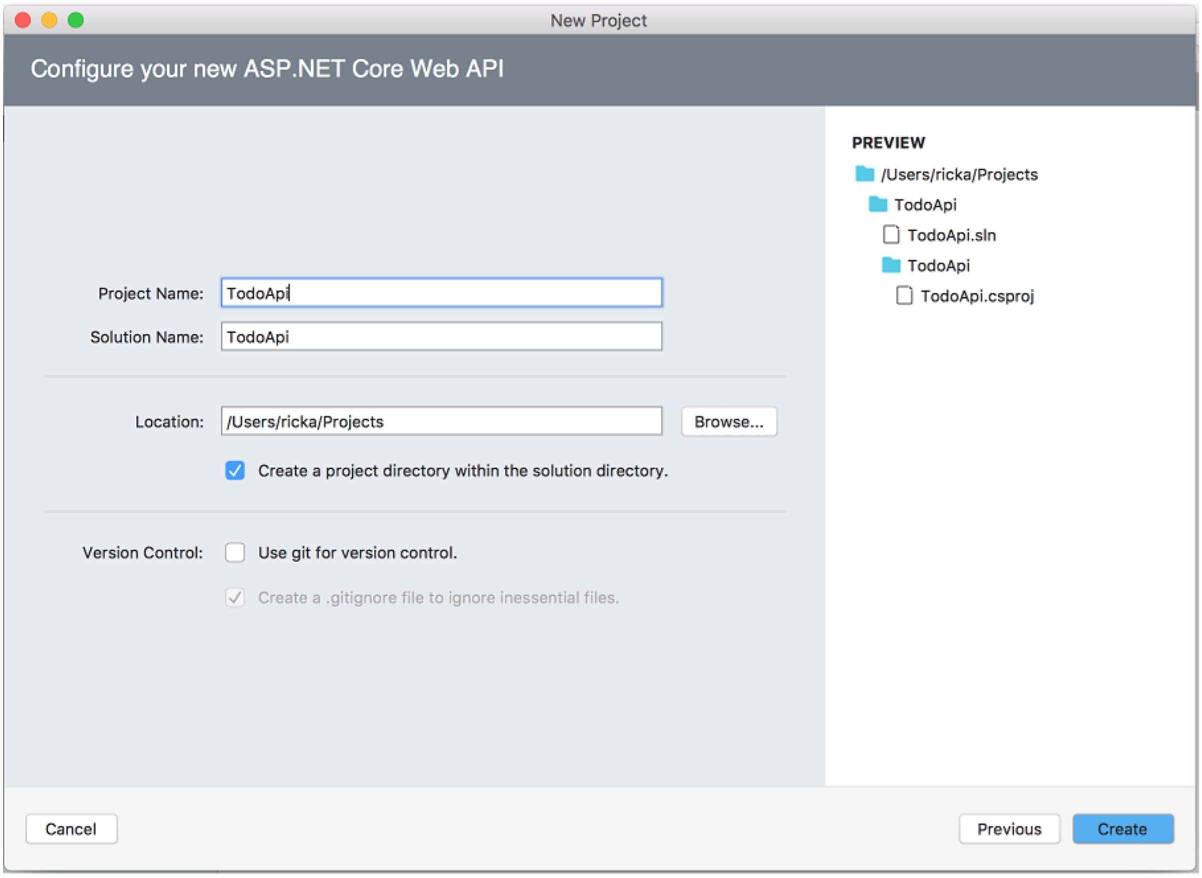
From Visual Studio, select **File > New Solution**.



Select **.NET Core App > ASP.NET Core Web API > Next**.



Enter **TodoApi** for the **Project Name**, and then select Create.



Launch the app

In Visual Studio, select **Run > Start With Debugging** to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port`, where *port* is a randomly chosen port number. You get an HTTP 404 (Not Found) error. Change the

URL to `http://localhost:port/api/values`. The `ValuesController` data will be displayed:

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

Add support for Entity Framework Core

Install the Entity Framework Core InMemory database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

From the **Project** menu, select **Add NuGet Packages**.

Alternately, you can right-click **Dependencies**, and then select **Add Packages**.

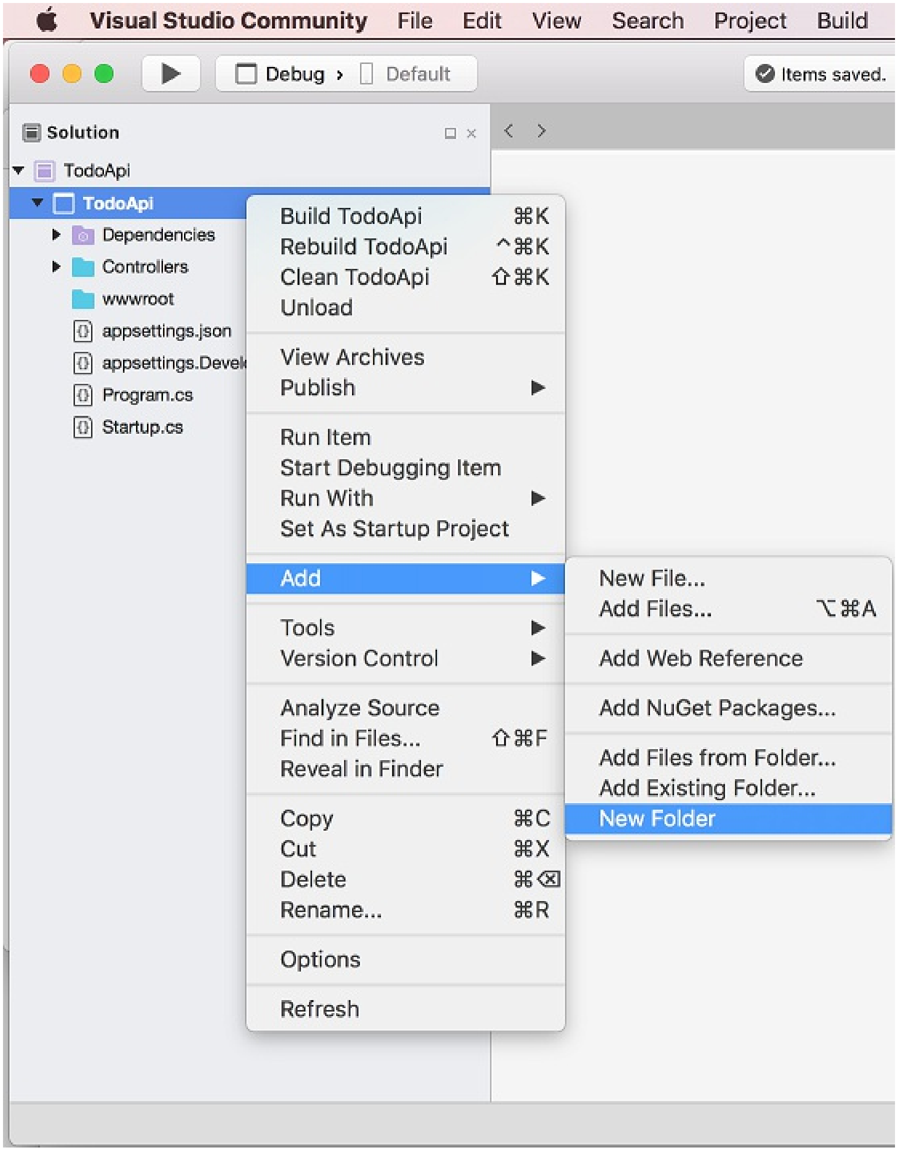
Enter `EntityFrameworkCore.InMemory` in the search box.

Select `Microsoft.EntityFrameworkCore.InMemory`, and then select **Add Package**.

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named *Models*. In Solution Explorer, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.



Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Add a `TodoItem` class. Right-click the *Models* folder and select **Add > New File > General > Empty Class**. Name the class `TodoItem`, and then select **New**.

Replace the generated code with:

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

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class to the *Models* folder.

```
using Microsoft.EntityFrameworkCore;

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

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

    }
}
```

Register the database context

In order to inject the database context into the controller, we need to register it with the dependency injection container. Register the database context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());
            services.AddMvc();
        }

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

The preceding code:

Removes the code we're not using.
Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, in the *Controllers* folder, add the class `TodoController`.

Replace the generated code with the following (and add closing braces):

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

Defines an empty controller class. In the next sections, we'll add methods to implement the API.
The constructor uses Dependency Injection to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

```
HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Server: Microsoft-IIS/10.0
   Date: Thu, 18 Jun 2015 20:51:10 GMT
   Content-Length: 82

   [{"Key":"1", "Name":"Item1","IsComplete":false}]
```

Later in the tutorial I'll show how you can view the HTTP response using Postman or or curl.

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

Take the template string in the controller's route attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller and the root name is "todo". ASP.NET Core routing is not case sensitive. If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`, append that to the path. This sample doesn't use a template. See Attribute routing with Http[Verb] attributes for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route and allows you to link to this route in an HTTP Response. I'll explain it with an example

later. See Routing to Controller Actions for detailed information.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.

Otherwise, the method returns 200 with a JSON response body. This is done by returning an `ObjectResult`

Launch the app

In Visual Studio, select **Run > Start With Debugging** to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port`, where *port* is a randomly chosen port number. You get an HTTP 404 (Not Found) error. Change the URL to `http://localhost:port/api/values`. The `ValuesController` data will be displayed:

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

Navigate to the `Todo` controller at`http://localhost:port/api/todo`:

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

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

Create

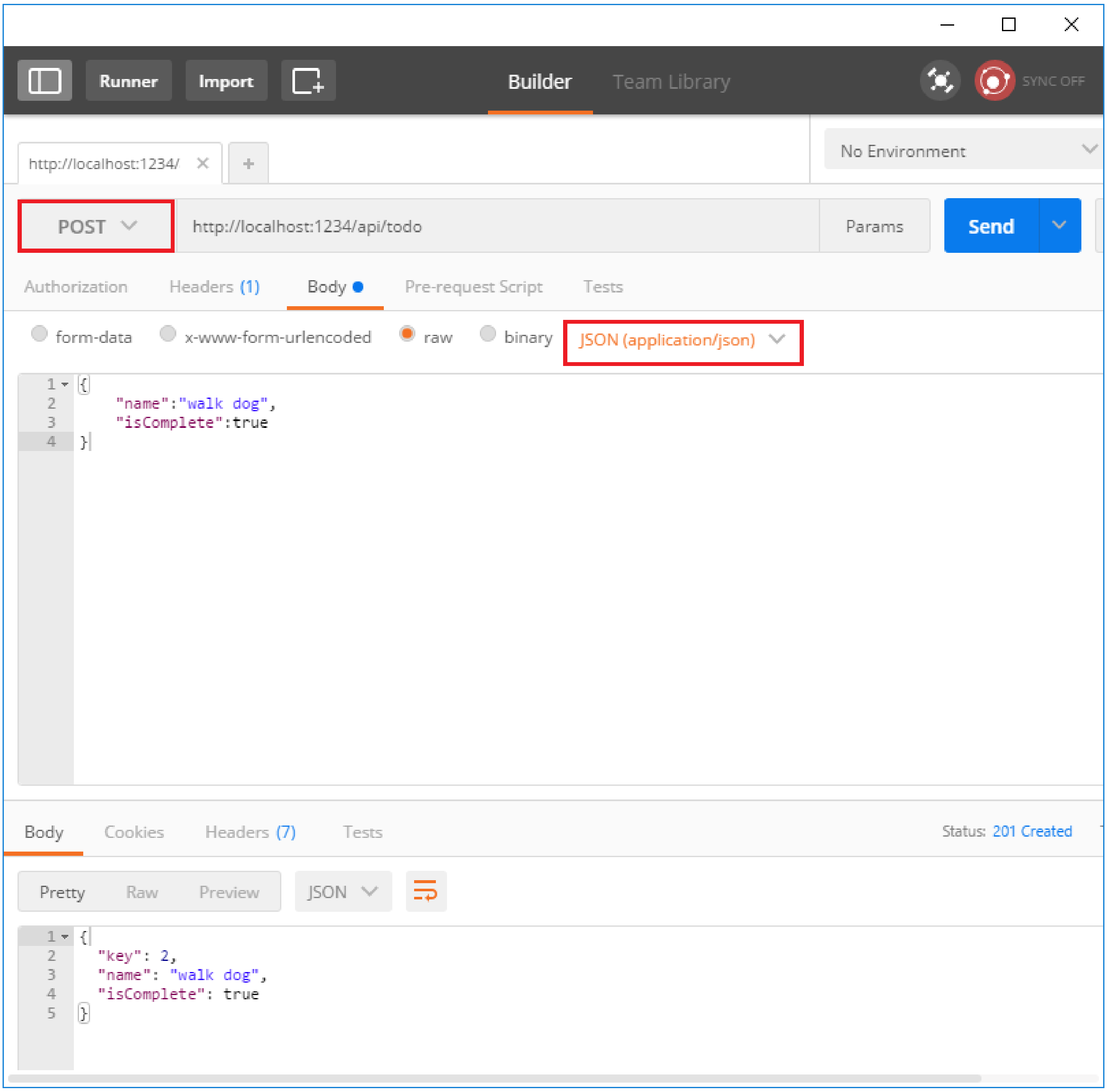
```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.

Use Postman to send a Create request

Start the app (**Run > Start With Debugging**).
Start Postman.



Set the HTTP method to `POST`

Select the **Body** radio button

Select the **raw** radio button

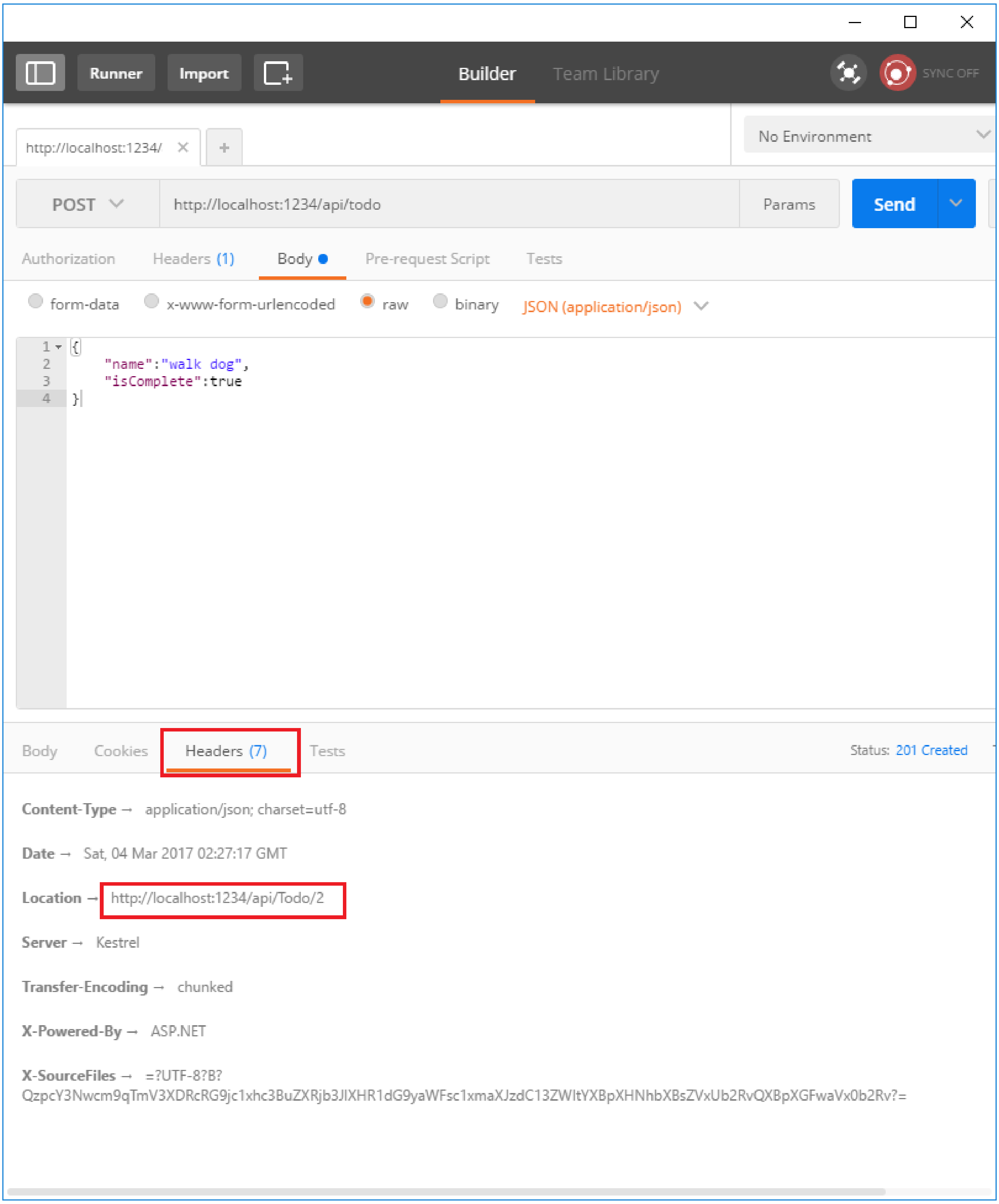
Set the type to JSON

In the key-value editor, enter a Todo item such as

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

Select **Send**

Select the Headers tab in the lower pane and copy the **Location** header:



You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

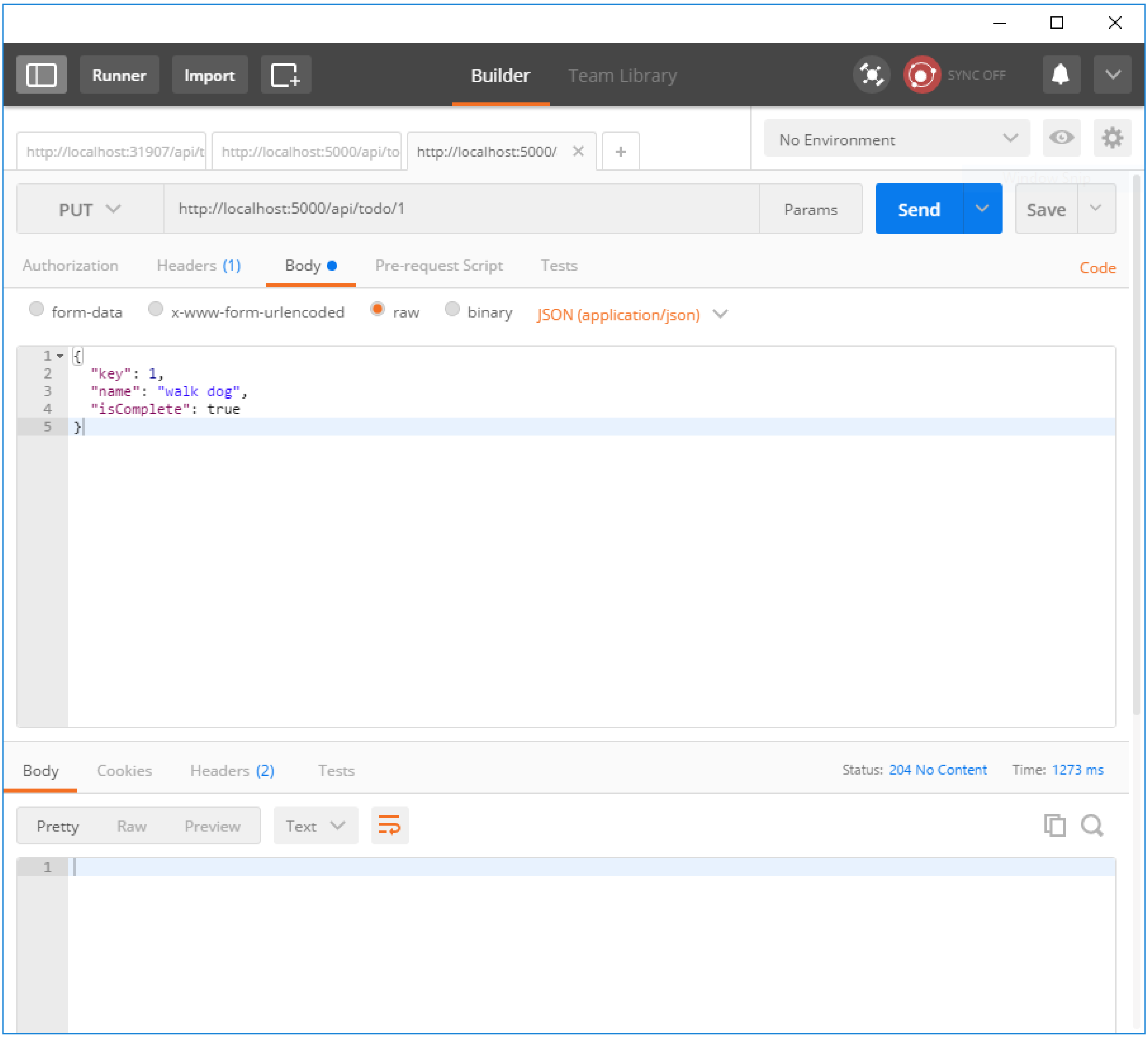
```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

`Update` is similar to `Create`, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.

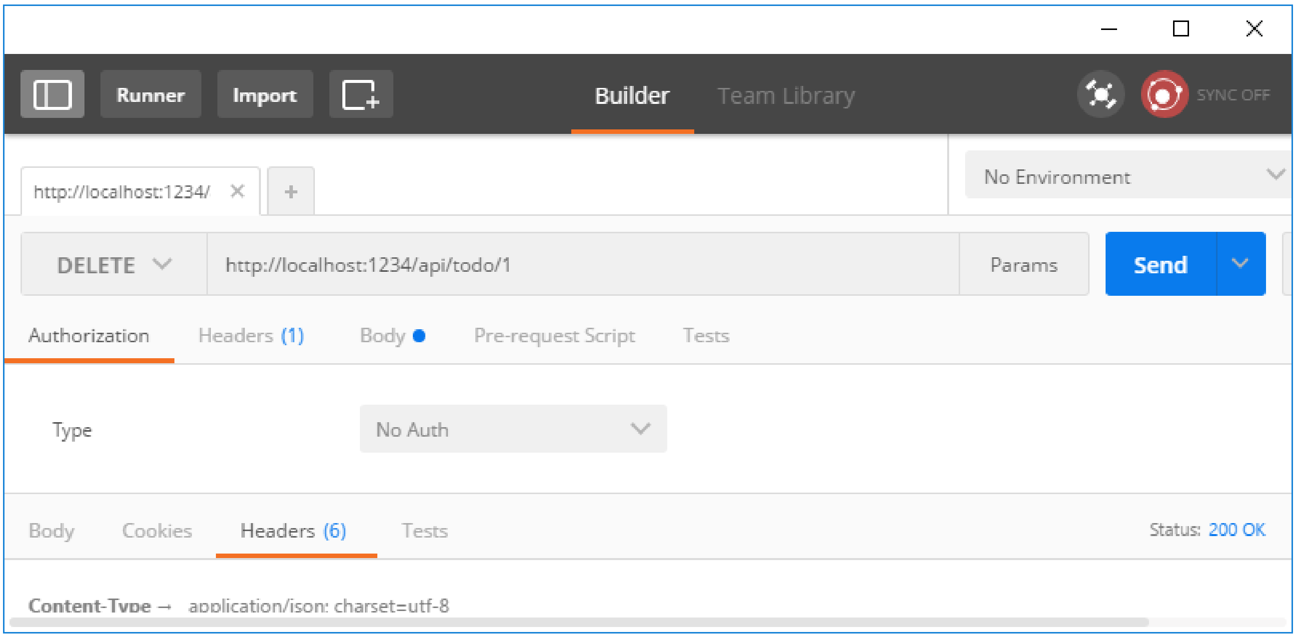
```
{
  "key": 1,
  "name": "walk dog",
  "isComplete": true
}
```



Delete

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

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

The response is 204 (No Content).



Next steps

Routing to Controller Actions

For information about deploying your API, see Publishing and Deployment.

View or download sample code

Postman

Fiddler

Create a web API with ASP.NET Core MVC and Visual Studio for Windows

By Rick Anderson and Mike Wasson

HTTP is not just for serving up web pages. It's also a powerful platform for building APIs that expose services and data. HTTP is flexible and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop apps.

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

ASP.NET Core has built-in support for MVC creating Web APIs.

There are 3 versions of this tutorial:

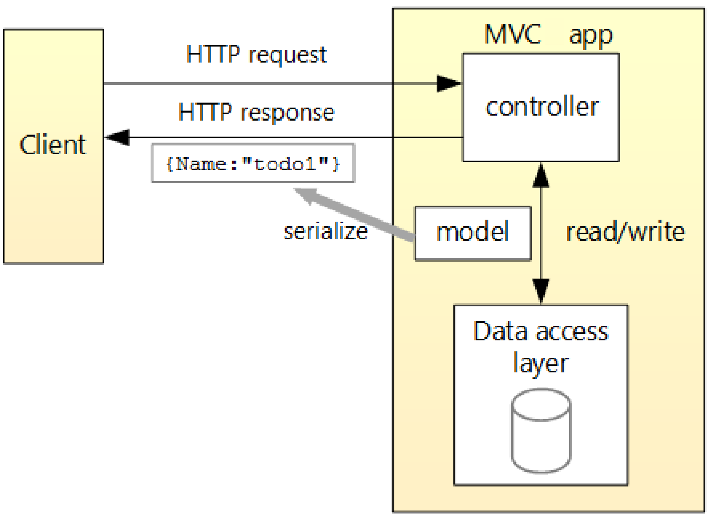
macOS: Web API with Visual Studio for Mac
Windows: Web API with Visual Studio for Windows
macOS, Linux, Windows: Web API with Visual Studio Code

Overview

Here is the API that you'll create:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



The client is whatever consumes the web API (mobile app, browser, etc). We aren't writing a client in this tutorial. We'll use

Postman or curl to test the app.

A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as C# classes, also know as **P**lain **O**ld **C#** **O**bject (POCOs).

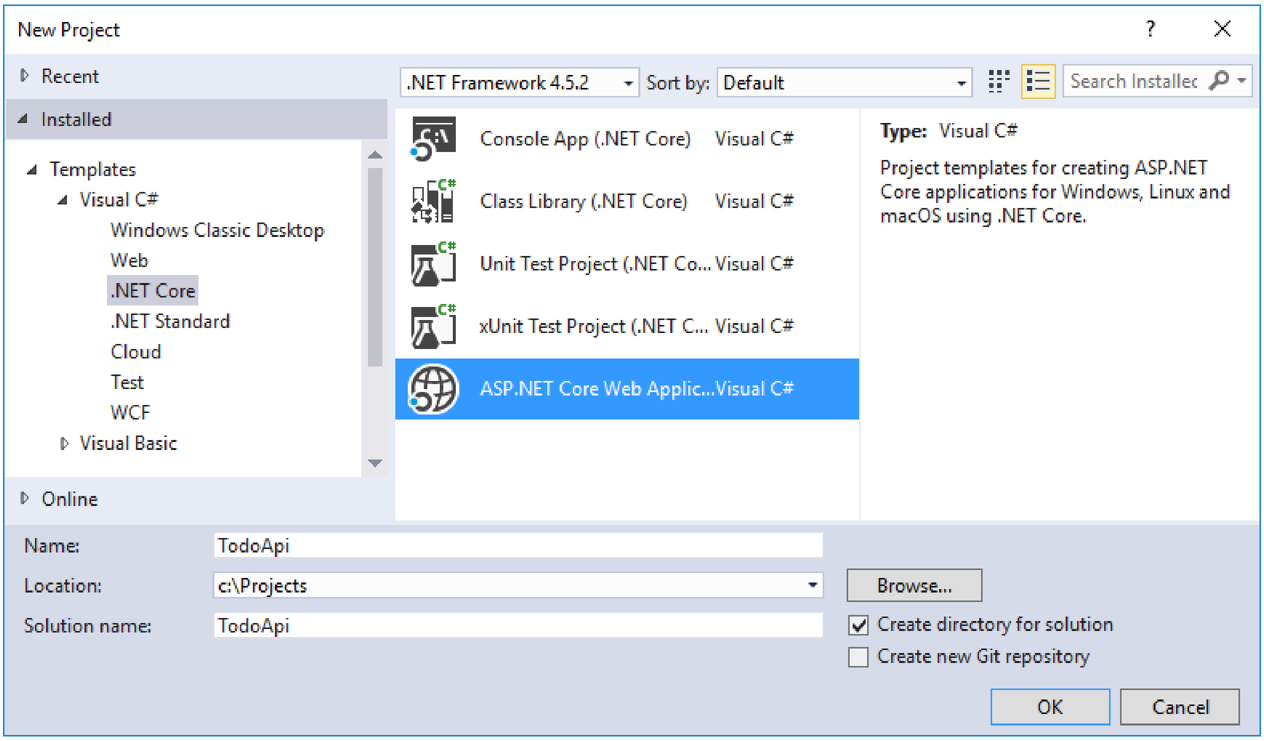
A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.

To keep the tutorial simple, the app doesn't use a persistent database. Instead, it stores to-do items in an in-memory database.

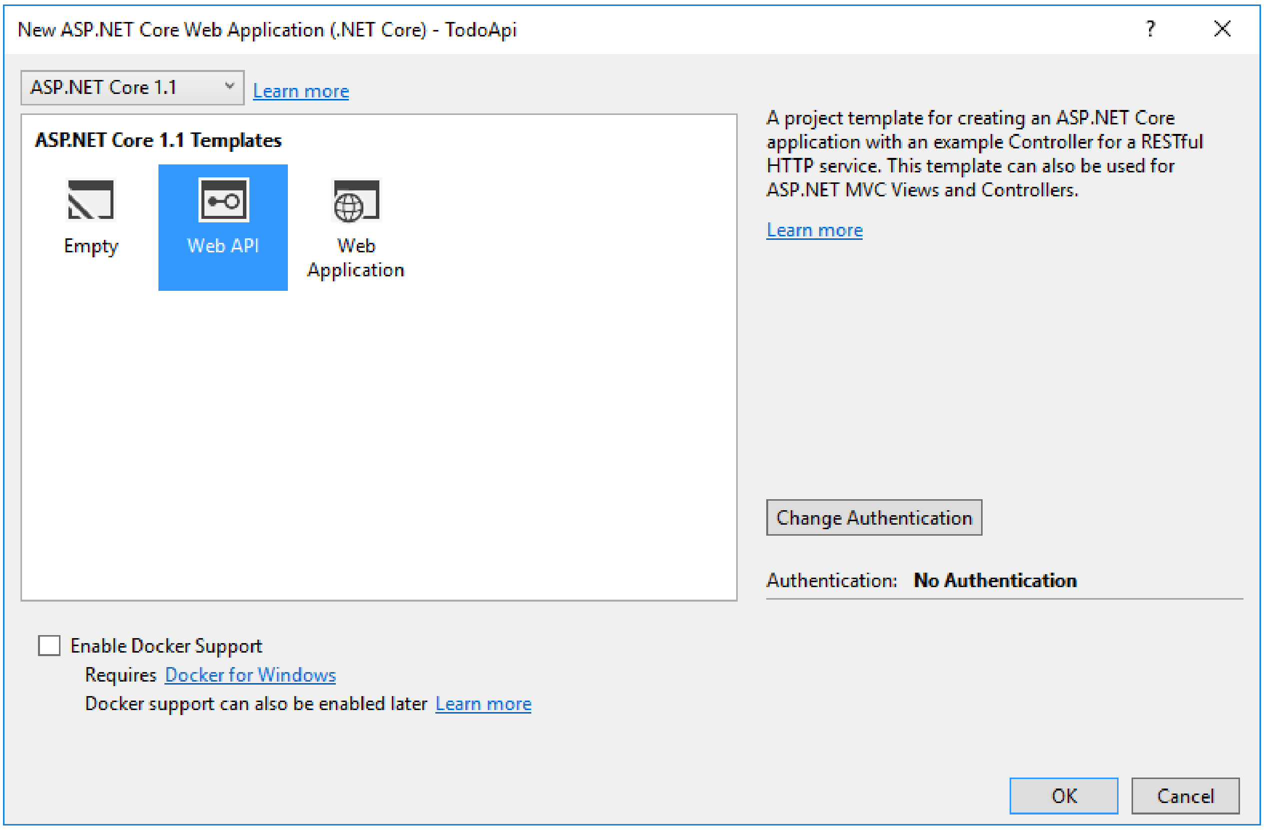
Create the project

From Visual Studio, select **File** menu, > **New** > **Project**.

Select the **ASP.NET Core Web Application (.NET Core)** project template. Name the project `TodoApi` and select **OK**.



In the **New ASP.NET Core Web Application (.NET Core) - TodoApi** dialog, select the **Web API** template. Select **OK**. Do **not** select **Enable Docker Support**.



Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the `ValuesController` data will be displayed:

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

If you're using IE, you are prompted to open or save the *values.json* file.

Add support for Entity Framework Core

Install the Entity Framework Core InMemory database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

Edit the *TodoApi.csproj* file. In Solution Explorer, right-click the project. Select **Edit TodoApi.csproj**. In the `ItemGroup` element, add "Microsoft.EntityFrameworkCore.InMemory":

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="1.1.1" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="1.0.0" />
  </ItemGroup>

</Project>
```

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named "Models". In Solution Explorer, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.

Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Add a `TodoItem` class. Right-click the *Models* folder and select **Add** > **Class**. Name the class `TodoItem` and select **Add**.

Replace the generated code with:

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

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class. Right-click the *Models* folder and select **Add** > **Class**. Name the class `TodoContext` and select **Add**.

```
using Microsoft.EntityFrameworkCore;

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

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

    }
}
```

Register the database context

In order to inject the database context into the controller, we need to register it with the dependency injection container. Register the database context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following:

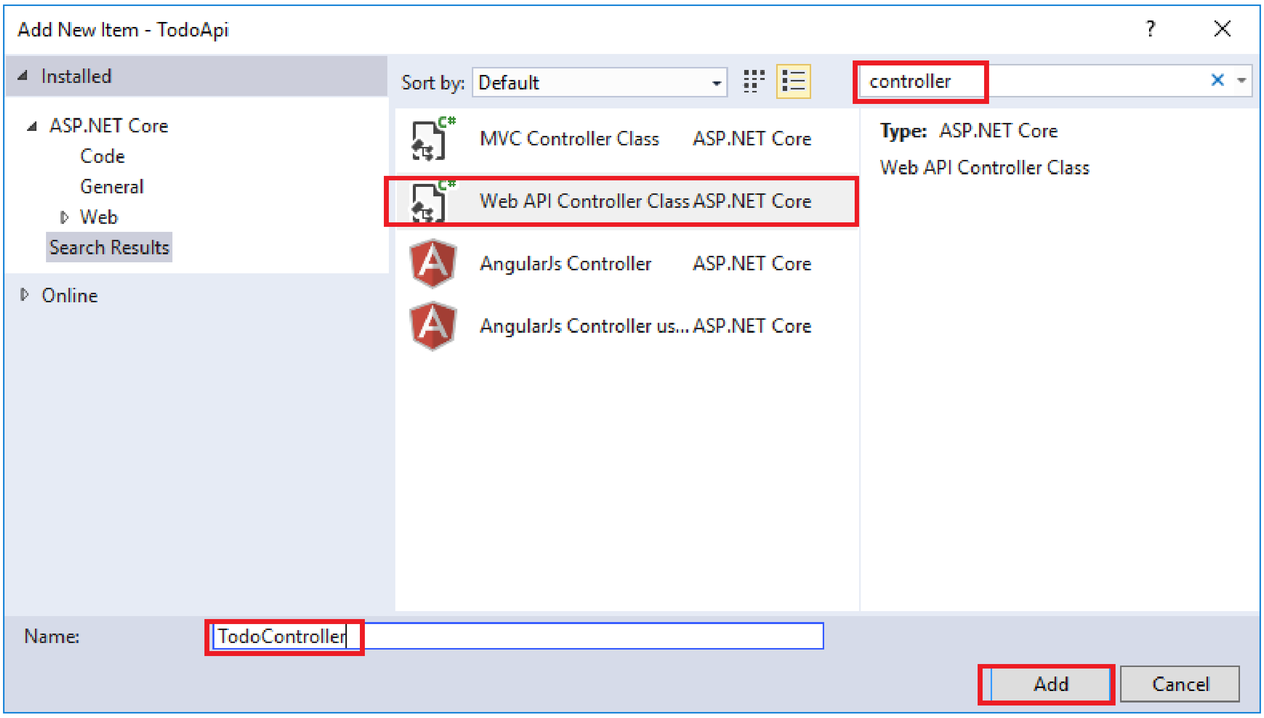
```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());
            services.AddMvc();
        }

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

The preceding code:

Removes the code we're not using.
Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, right-click the *Controllers* folder. Select **Add** > **New Item**. In the **Add New Item** dialog, select the **Web API Controller Class** template. Name the class `TodoController`.



Replace the generated code with the following:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

Defines an empty controller class. In the next sections, we'll add methods to implement the API.
The constructor uses Dependency Injection to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

```
HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Server: Microsoft-IIS/10.0
   Date: Thu, 18 Jun 2015 20:51:10 GMT
   Content-Length: 82

   [{"Key":"1", "Name":"Item1","IsComplete":false}]
```

Later in the tutorial I'll show how you can view the HTTP response using Postman or or curl.

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

Take the template string in the controller's route attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller and the root name is "todo". ASP.NET Core routing is not case sensitive. If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`, append that to the path. This sample doesn't use a template. See Attribute routing with Http[Verb] attributes for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route and allows you to link to this route in an HTTP Response. I'll explain it with an example

later. See Routing to Controller Actions for detailed information.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.

Otherwise, the method returns 200 with a JSON response body. This is done by returning an `ObjectResult`

Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the data will be displayed. If you're using IE, IE will prompt to you open or save the *values.json* file. Navigate to the `Todo` controller we just created `http://localhost:port/api/todo`.

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

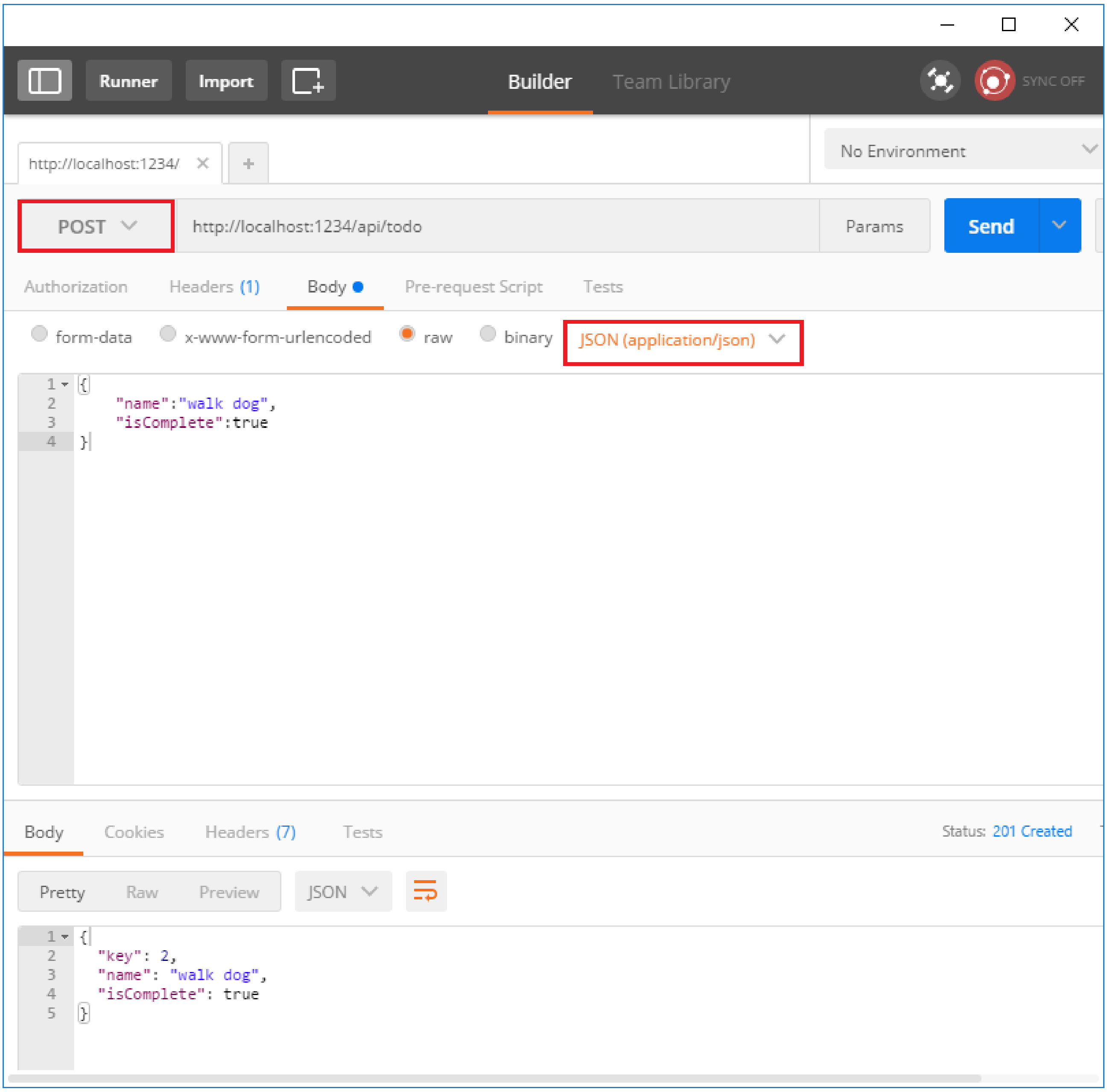
Create

```
[HttpPost]
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);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.

Use Postman to send a Create request



Set the HTTP method to `POST`

Select the **Body** radio button

Select the **raw** radio button

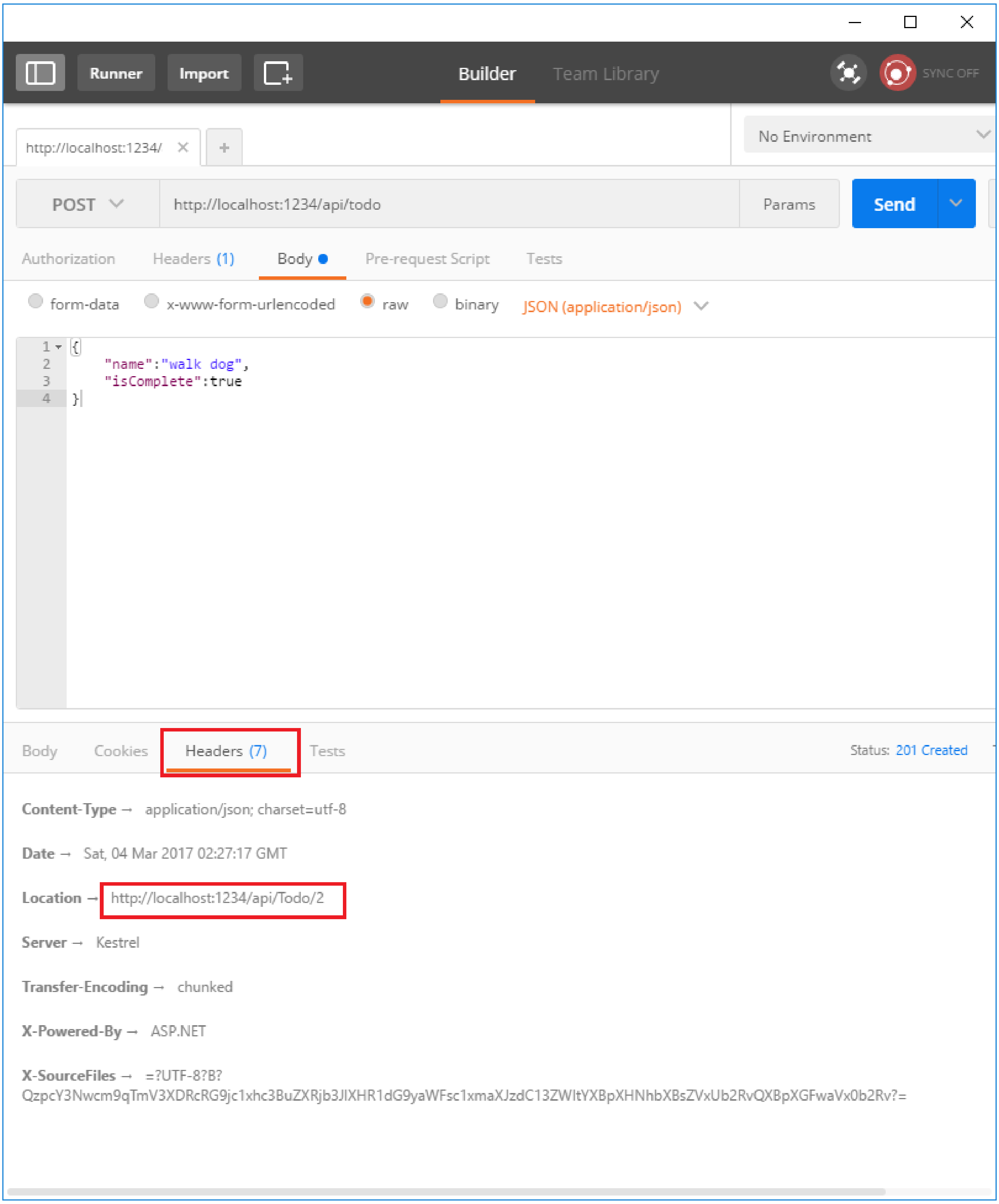
Set the type to JSON

In the key-value editor, enter a Todo item such as

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

Select **Send**

Select the Headers tab in the lower pane and copy the **Location** header:



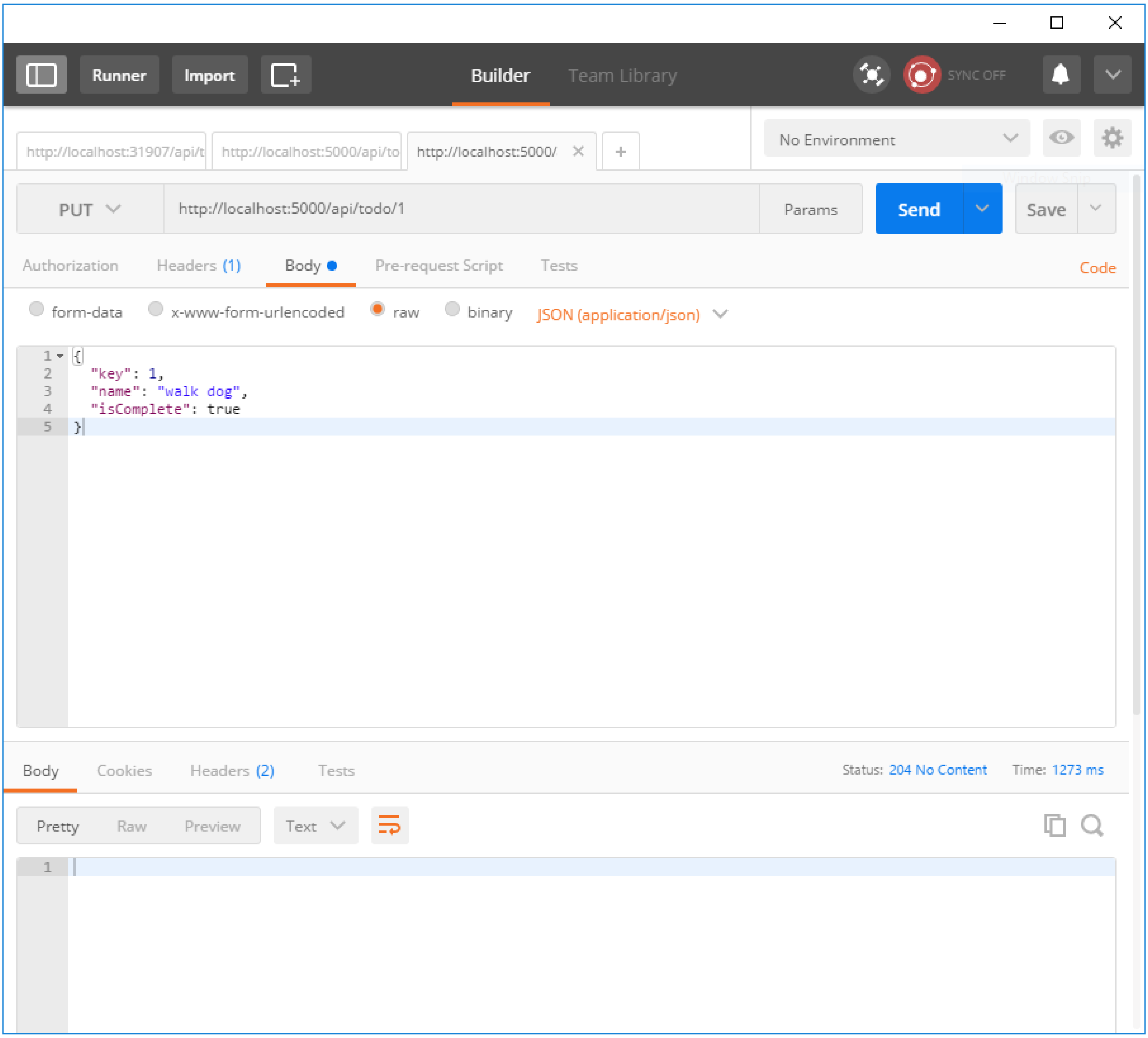
You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

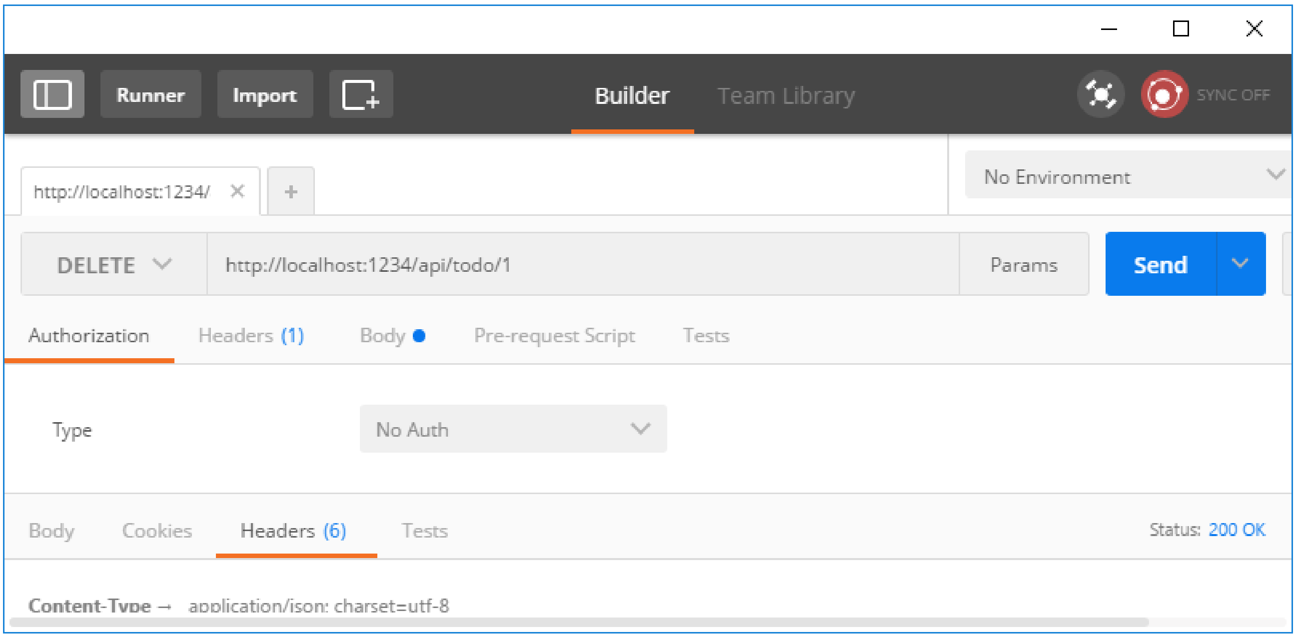
`Update` is similar to `Create`, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



Delete

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

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

The response is 204 (No Content).



Next steps

Routing to Controller Actions
For information about deploying your API, see Publishing and Deployment.
View or download sample code. See how to download.
Postman

Create a Web API with ASP.NET Core MVC and Visual Studio Code on Linux, macOS, and Windows

By Rick Anderson and Mike Wasson

HTTP is not just for serving up web pages. It's also a powerful platform for building APIs that expose services and data. HTTP is flexible and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop apps.

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

ASP.NET Core has built-in support for MVC creating Web APIs.

There are 3 versions of this tutorial:

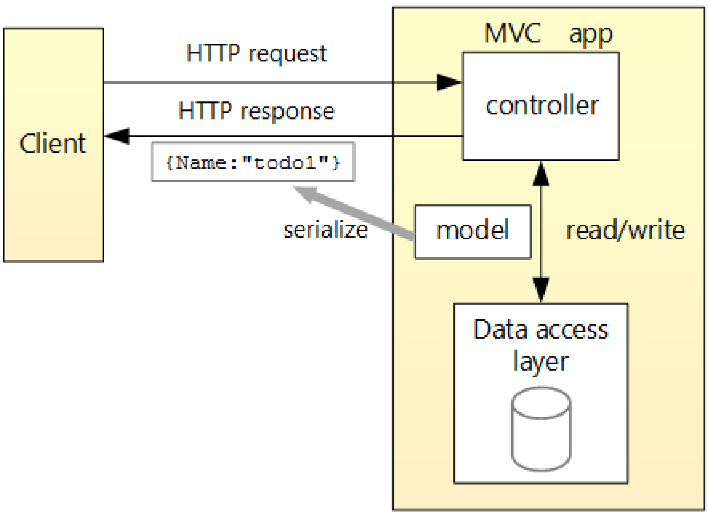
macOS: Web API with Visual Studio for Mac
Windows: Web API with Visual Studio for Windows
macOS, Linux, Windows: Web API with Visual Studio Code

Overview

Here is the API that you'll create:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



The client is whatever consumes the web API (mobile app, browser, etc). We aren't writing a client in this tutorial. We'll use

Postman or curl to test the app.

A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as C# classes, also know as **P**lain **O**ld **C#** **O**bject (POCOs).

A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.

To keep the tutorial simple, the app doesn't use a persistent database. Instead, it stores to-do items in an in-memory database.

Set up your development environment

Download and install:

.NET Core
Visual Studio Code
Visual Studio Code C# extension

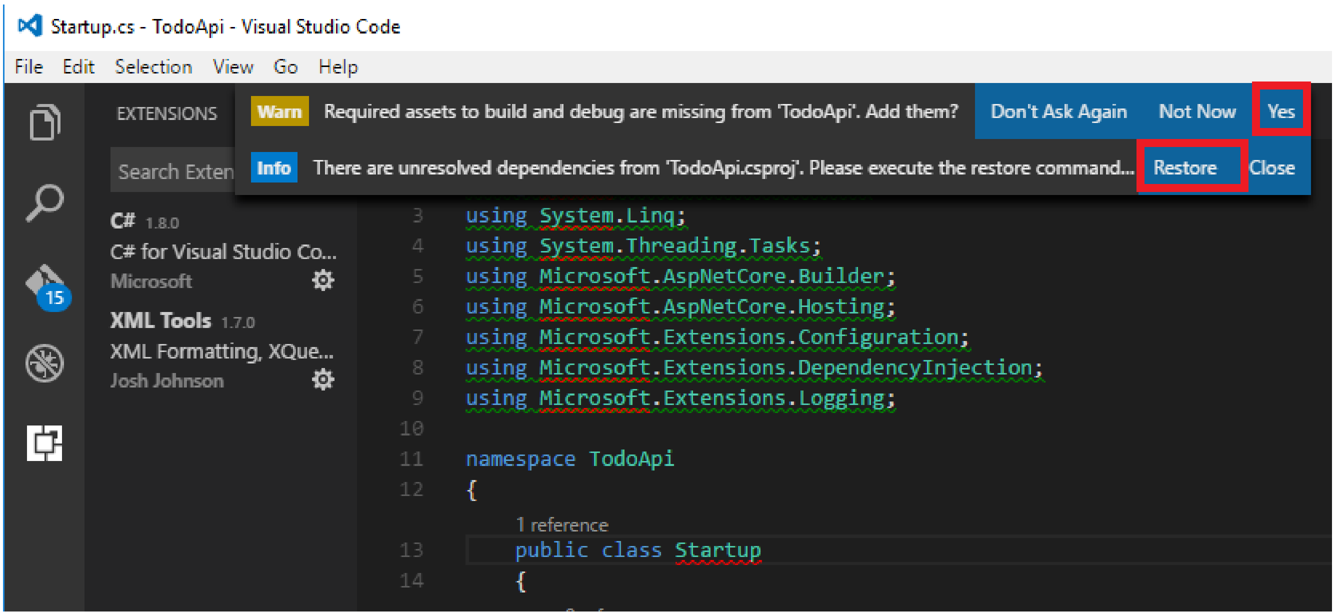
Create the project

From a console, run the following commands:

```
mkdir TodoApi
cd TodoApi
dotnet new webapi
```

Open the *TodoApi* folder in Visual Studio Code (VS Code) and select the *Startup.cs* file.

Select **Yes** to the **Warn** message "Required assets to build and debug are missing from 'TodoApi'. Add them?"
Select **Restore** to the **Info** message "There are unresolved dependencies".



Press **Debug** (F5) to build and run the program. In a browser navigate to http://localhost:5000/api/values . The following is displayed:

`["value1","value2"]`

See Visual Studio Code help for tips on using VS Code.

Add support for Entity Framework Core

Edit the *TodoApi.csproj* file to install the Entity Framework Core InMemory database provider. This database provider allows Entity

Framework Core to be used with an in-memory database.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="1.1.1" />
  </ItemGroup>
</Project>
```

Run `dotnet restore` to download and install the EF Core InMemory DB provider. You can run `dotnet restore` from the terminal or enter `⌘⇧P` (macOS) or `Ctrl+Shift+P` (Linux) in VS Code and then type **.NET**. Select **.NET: Restore Packages**.

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named *Models*. You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Add a `TodoItem` class with the following code:

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

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class in the *Models* folder:

```
using Microsoft.EntityFrameworkCore;

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

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

    }
}
```

Register the database context

In order to inject the database context into the controller, we need to register it with the dependency injection container. Register the database context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());
            services.AddMvc();
        }

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

The preceding code:

Removes the code we're not using.
Specifies an in-memory database is injected into the service container.

Add a controller

In the *Controllers* folder, create a class named `TodoController`. Add the following code:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

- Defines an empty controller class. In the next sections, we'll add methods to implement the API.
- The constructor uses Dependency Injection to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
- The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

```
HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Server: Microsoft-IIS/10.0
   Date: Thu, 18 Jun 2015 20:51:10 GMT
   Content-Length: 82

   [{"Key":"1", "Name":"Item1","IsComplete":false}]
```

Later in the tutorial I'll show how you can view the HTTP response using Postman or or curl.

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

- Take the template string in the controller's route attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

- Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller and the root name is "todo". ASP.NET Core routing is not case sensitive.

If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`, append that to the path. This sample doesn't use a template. See Attribute routing with Http[Verb] attributes for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route and allows you to link to this route in an HTTP Response. I'll explain it with an example later. See Routing to Controller Actions for detailed information.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.

Otherwise, the method returns 200 with a JSON response body. This is done by returning an `ObjectResult`

Launch the app

In VS Code, press F5 to launch the app. Navigate to http://localhost:5000/api/todo (The `Todo` controller we just created).

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

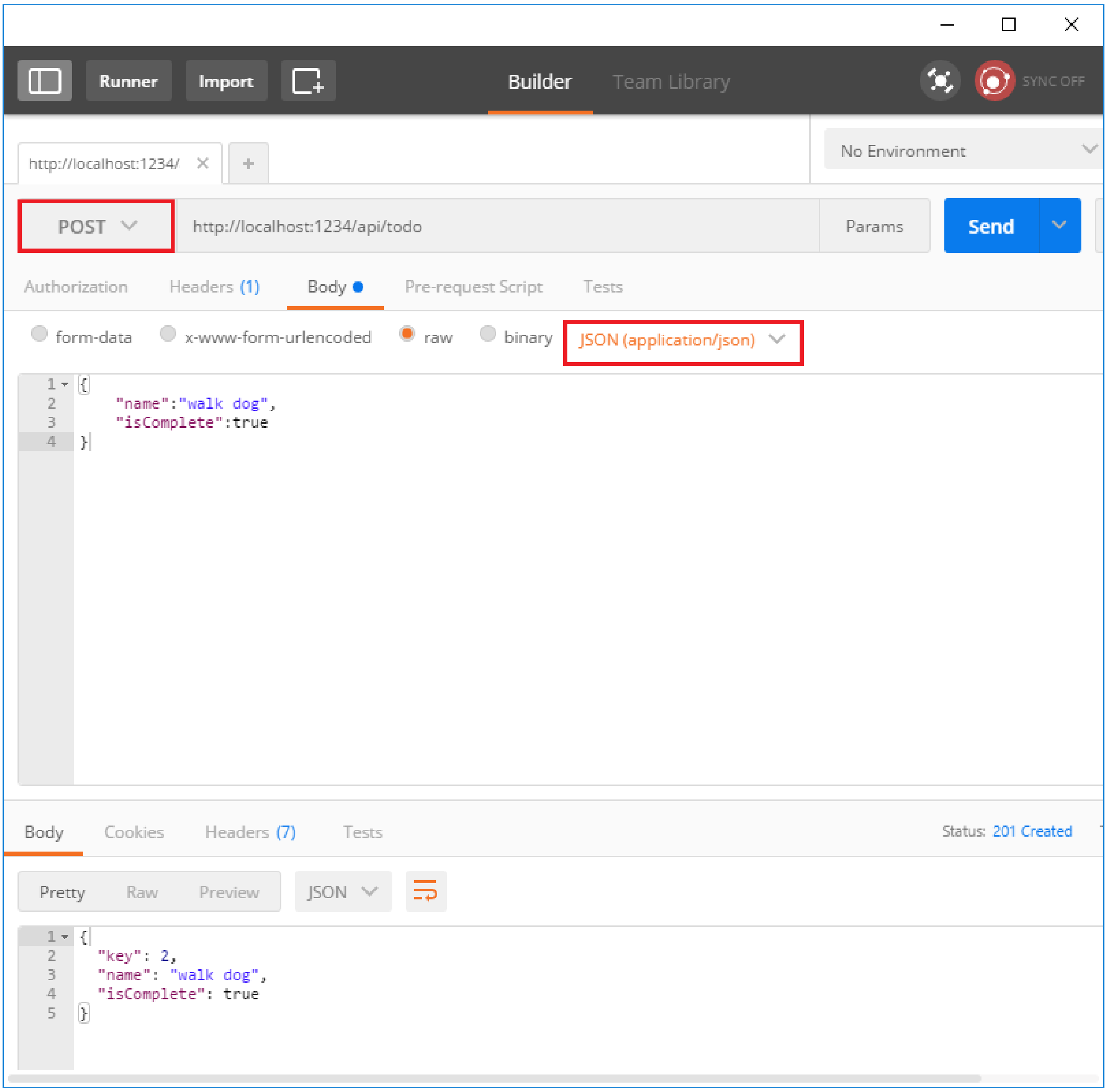
Create

```
[HttpPost]
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);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.

Use Postman to send a Create request



Set the HTTP method to `POST`

Select the **Body** radio button

Select the **raw** radio button

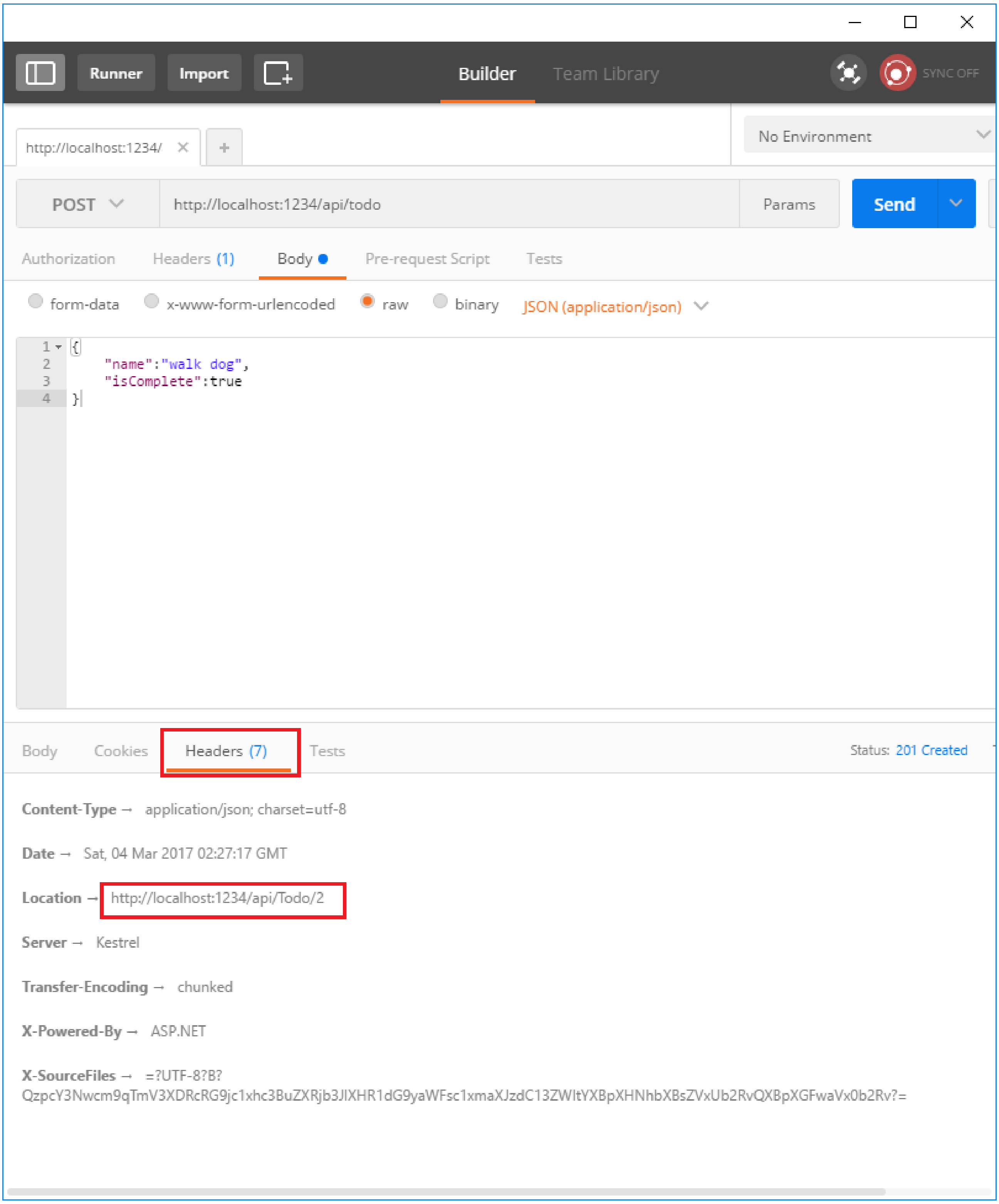
Set the type to JSON

In the key-value editor, enter a Todo item such as

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

Select **Send**

Select the Headers tab in the lower pane and copy the **Location** header:



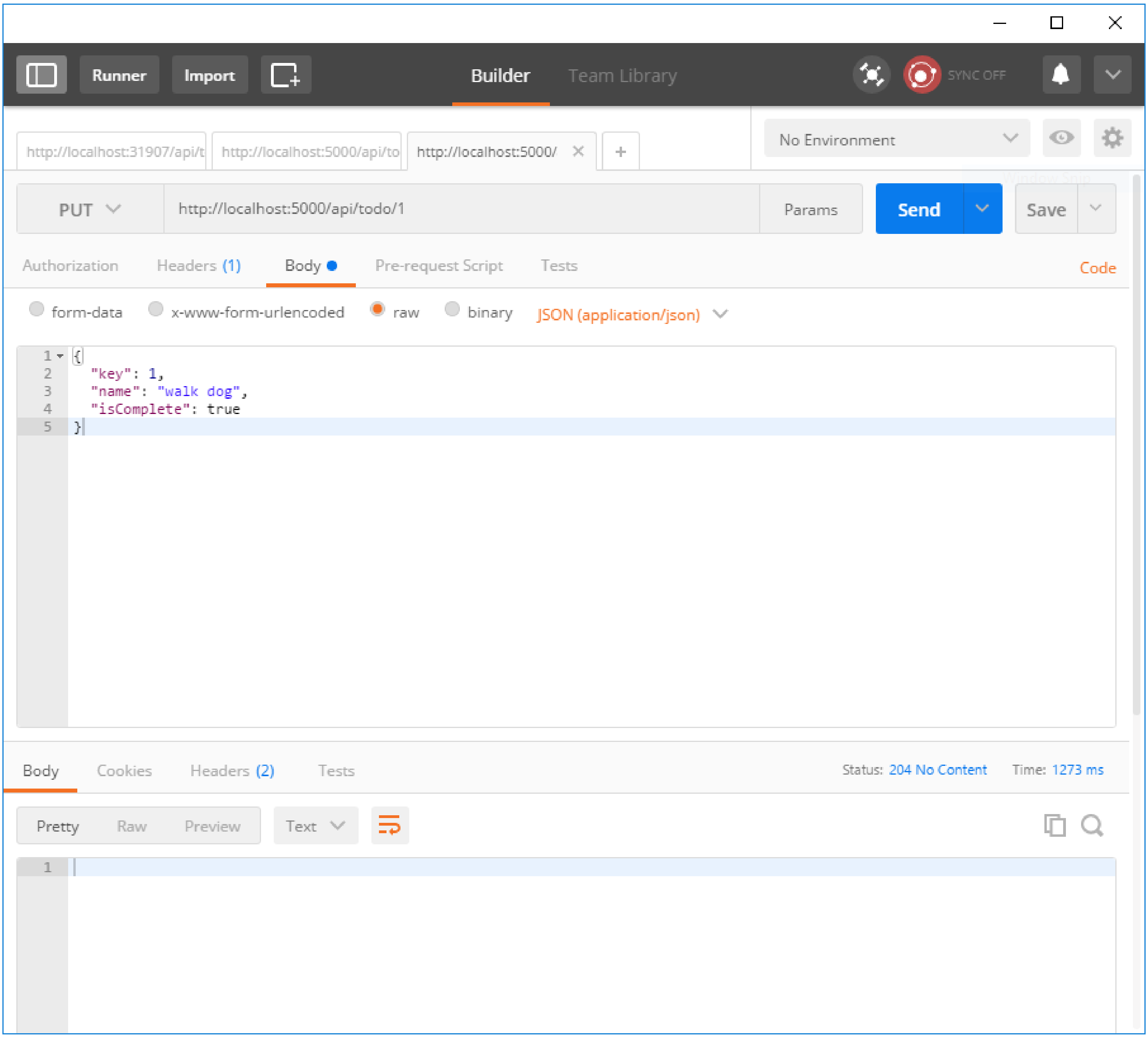
You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

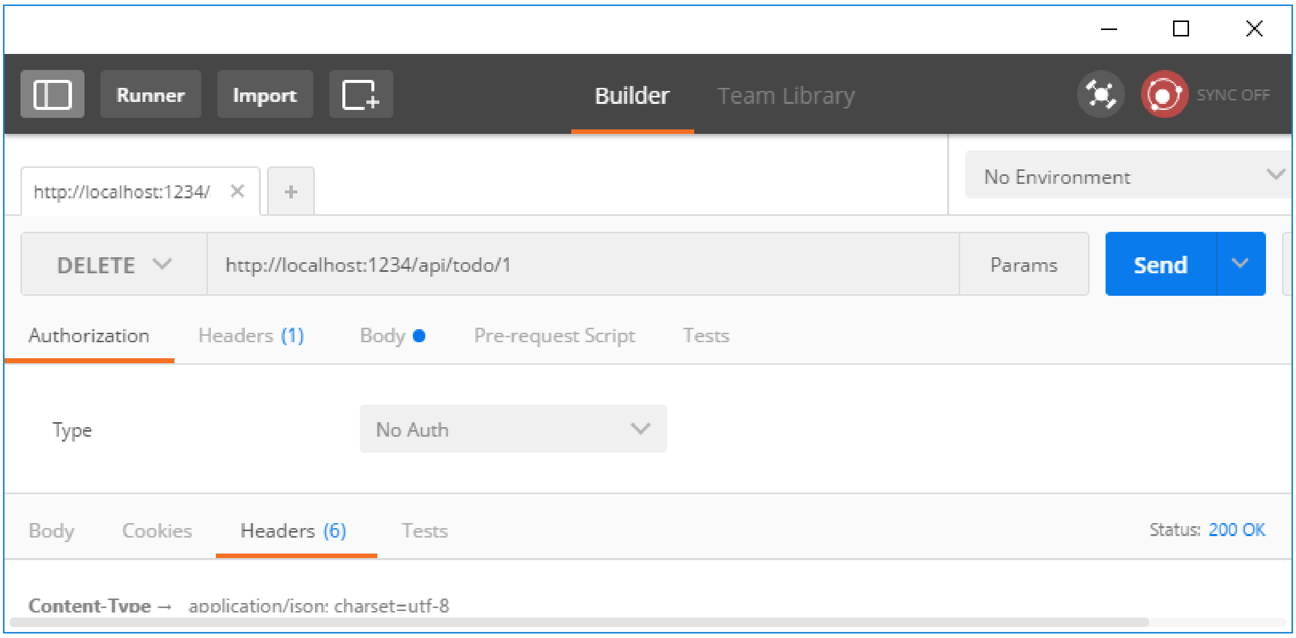
`Update` is similar to `Create`, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



Delete

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

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

The response is 204 (No Content).



Visual Studio Code help

Getting started
Debugging
Integrated terminal

Keyboard shortcuts

Mac keyboard shortcuts
Linux keyboard shortcuts
Windows keyboard shortcuts

Next steps

Routing to Controller Actions
For information about deploying your API, see Publishing and Deployment.
View or download sample code. See how to download.
Postman

Getting started with ASP.NET Core and Entity Framework Core using Visual Studio

This series of tutorials teaches you how to create ASP.NET Core MVC web applications that use Entity Framework Core for data access. The tutorials require Visual Studio 2017.

Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio (1 of 10)

By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017.

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample application from scratch.

Download or view the completed application.

EF Core 1.1 is the latest version of EF but does not yet have all the features of EF 6.x. For information about how to choose between EF 6.x and EF Core 1.0, see EF Core vs. EF6.x. If you choose EF 6.x, see the previous version of this tutorial series.

 Note

For the Visual Studio 2015 version of this tutorial, see the VS 2015 version of ASP.NET Core documentation in PDF format.

Prerequisites

Visual Studio 2017 with the **ASP.NET and web development** and **.NET Core cross-platform development workloads** installed.

Troubleshooting

If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. For a list of common errors and how to solve them, see the Troubleshooting section of the last tutorial in the series. If you don't find what you need there, you can post a question to StackOverflow.com for ASP.NET Core or EF Core.

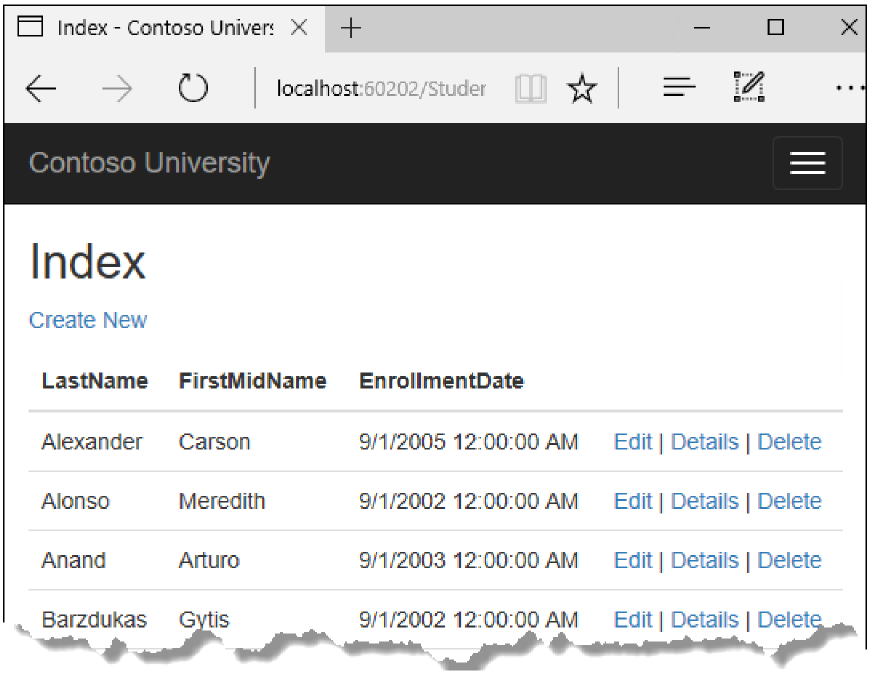
 Tip

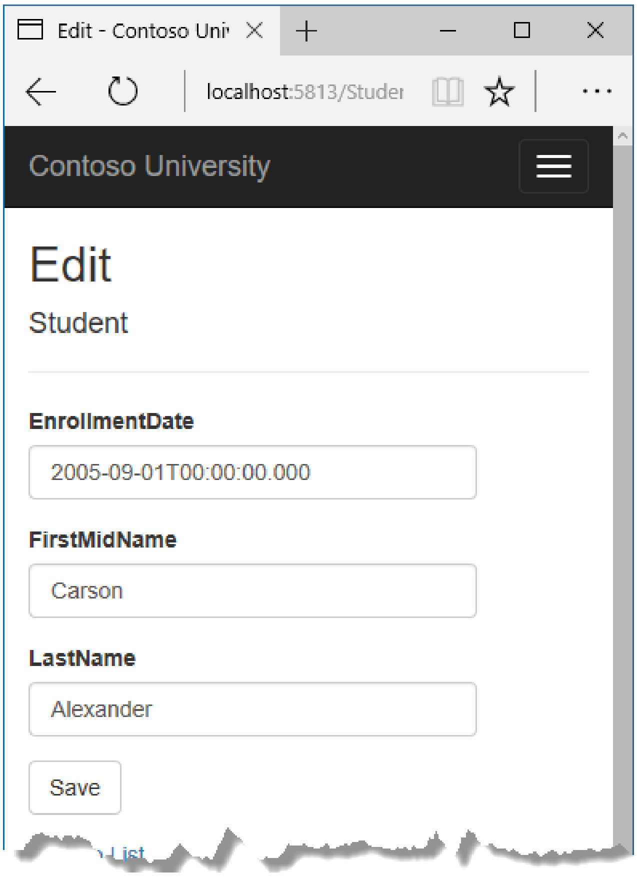
This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

The Contoso University web application

The application you'll be building in these tutorials is a simple university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.





The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Create an ASP.NET Core MVC web application

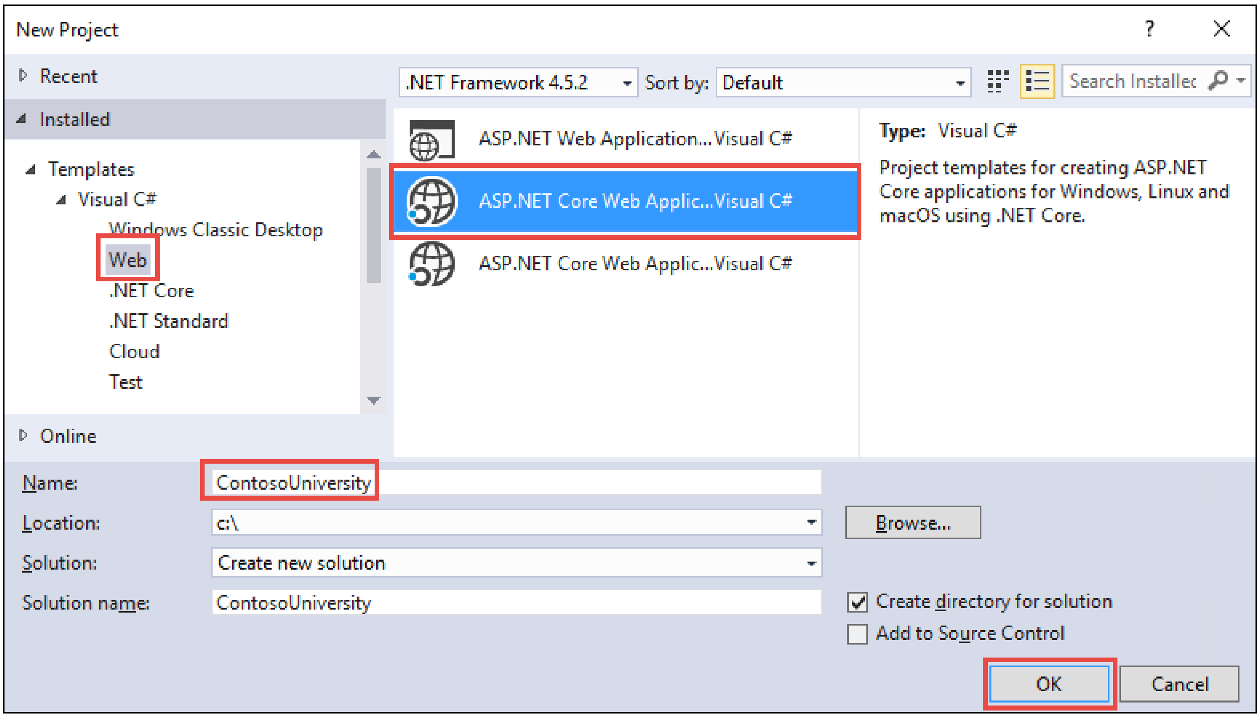
Open Visual Studio and create a new ASP.NET Core C# web project named "ContosoUniversity".

From the **File** menu, select **New > Project**.

From the left pane, select **Templates > Visual C# > Web**.

Select the **ASP.NET Core Web Application (.NET Core)** project template.

Enter **ContosoUniversity** as the name and click **OK**.



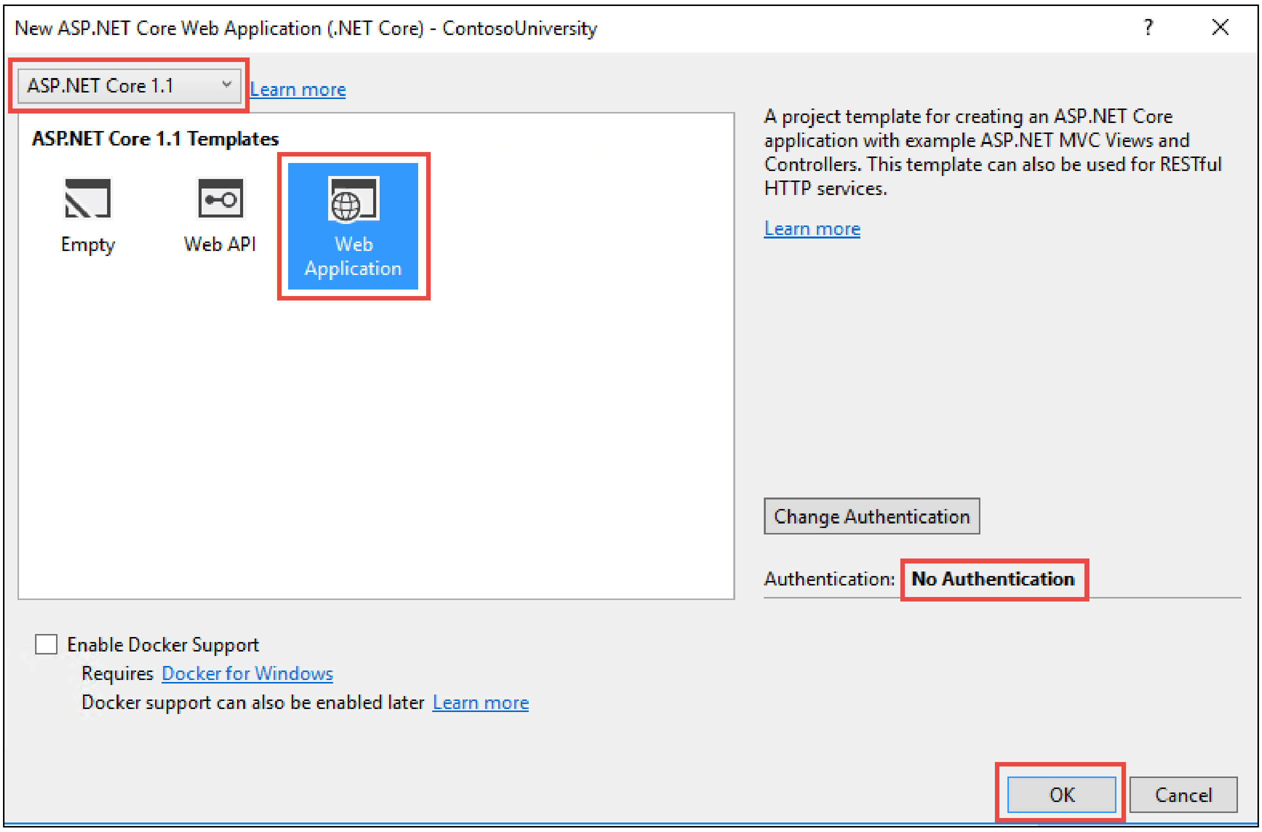
Wait for the **New ASP.NET Core Web Application (.NET Core)** dialog to appear

Select **ASP.NET Core 1.1** and the **Web Application** template.

Note: This tutorial requires ASP.NET Core 1.1 and EF Core 1.1 or later -- make sure that **ASP.NET Core 1.0** is not selected.

Make sure **Authentication** is set to **No Authentication**.

Click **OK**



Set up the site style

A few simple changes will set up the site menu, layout, and home page.

Open *Views/Shared/_Layout.cshtml* and make the following changes:

Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.

Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted.

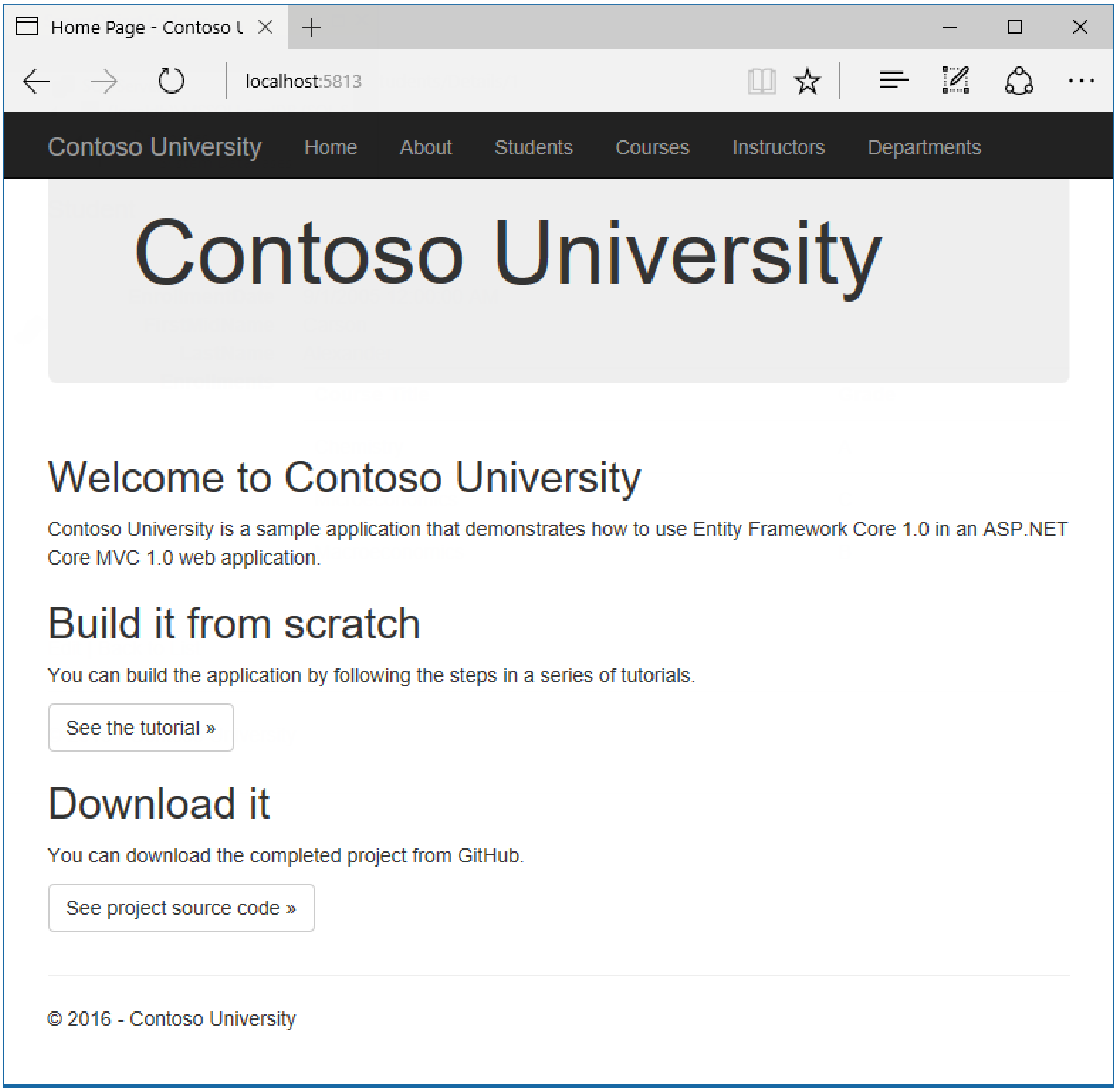
```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <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>
    @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="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>
    </nav>
    <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-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Views/Home/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

```
@{
    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/aspnet/data/ef-
mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu. You see the home page with tabs for the pages you'll create in these tutorials.



Entity Framework Core NuGet packages

To add EF Core support to a project, install the database provider that you want to target. For this tutorial, install the SQL Server provider: Microsoft.EntityFrameworkCore.SqlServer.

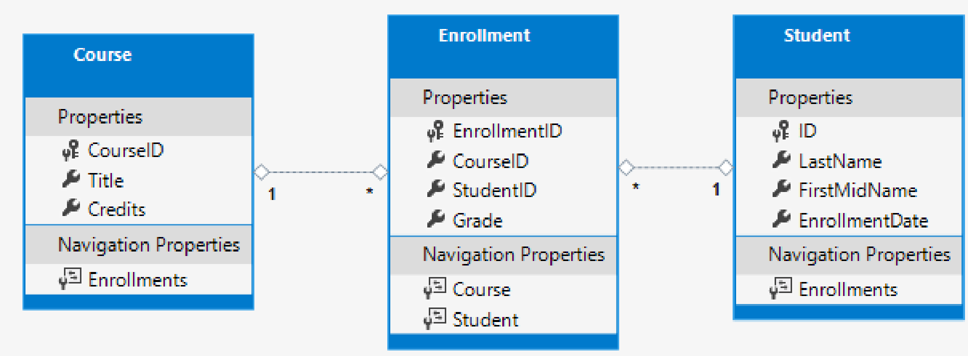
To install the package, enter the following command in **Package Manager Console (PMC)**. (From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**.)

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

This package and its dependencies (`Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`) provide run-time support for EF. You'll add a tooling package later, in the Migrations tutorial.

Create the data model

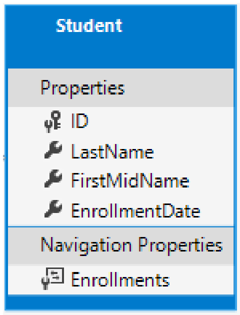
Next you'll create entity classes for the Contoso University application. You'll start with the following three entities.



There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

The Student entity



In the project folder, create a folder named *Models*.

In the *Models* folder, create a class file named *Student.cs* and replace the template code with the following code.

```
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; }
    }
}
```

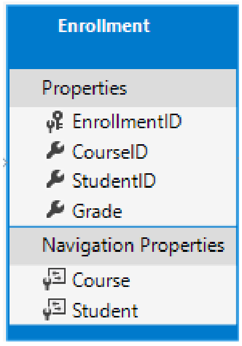
The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student entity` will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given Student row in the database has two related Enrollment rows (rows that contain that student's primary key value in their StudentID foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection<T>`. You can specify `ICollection<T>` or a type such as

`List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
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; }
    }
}
```

The `EnrollmentID` property will be the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using ID without classname makes it easier to implement inheritance in the data model.

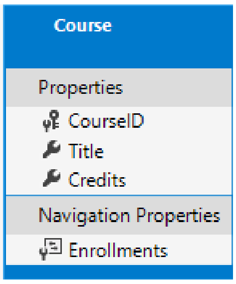
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply `<primary key property name>` (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity



In the *Models* folder, create *Course.cs* and replace the existing code with the following code:

```
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; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `DatabaseGenerated` attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

The main class that coordinates Entity Framework functionality for a given data model is the database context class. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

```
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; }
    }
}
```

This code creates a `DbSet` property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

You could have omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student), but developers disagree about whether table names should be pluralized or not. For these tutorials you'll override the default behavior by specifying singular table names in the DbContext. To do that, add the following highlighted code after the last DbSet property.

```
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");
        }
    }
}
```

Register the context with dependency injection

ASP.NET Core implements dependency injection by default. Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You'll see the controller constructor code that gets a context instance later in this tutorial.

To register `SchoolContext` as a service, open *Startup.cs*, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces, and then build the project.

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

Open the *appsettings.json* file and add a connection string as shown in the following example.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB database. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for application development, not production use. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB creates *.mdf* database files in the `C:/Users/<user>` directory.

Add code to initialize the database with test data

The Entity Framework will create an empty database for you. In this section, you write a method that is called after the database is created in order to populate it with test data.

Here you'll use the `EnsureCreated` method to automatically create the database. In a later tutorial you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class file named *DbInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
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[]
            {
            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();
        }
    }
}
```

The code checks if there are any students in the database, and if not, it assumes the database is new and needs to be seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

In *Startup.cs*, modify the `Configure` method to call this seed method on application startup. First, add the context to the method signature so that ASP.NET dependency injection can provide it to your `DbInitializer` class.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory,
SchoolContext context)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
```

Then call your `DbInitializer.Initialize` method at the end of the `Configure` method.

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

DbInitializer.Initialize(context);
```

Now the first time you run the application the database will be created and seeded with test data. Whenever you change your data model, you can delete the database, update your seed method, and start afresh with a new database the same way. In later tutorials you'll see how to modify the database when the data model changes, without deleting and re-creating it.

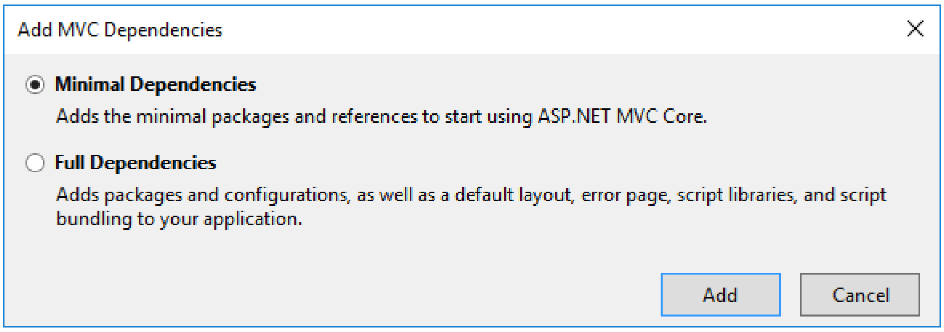
Create a controller and views

Next, you'll use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding. Scaffolding differs from code generation in that the scaffolded code is a starting point that you can modify to suit your own requirements, whereas you typically don't modify generated code. When you need to customize generated code, you use partial classes or you regenerate the code when things change.

Right-click the **Controllers** folder in **Solution Explorer** and select **Add > New Scaffolded Item**.

In the **Add MVC Dependencies** dialog, select **Minimal Dependencies**, and select **Add**.



Visual Studio adds the dependencies needed to scaffold a controller, including a package with design-time EF functionality (`Microsoft.EntityFrameworkCore.Design`). A package that is needed only for scaffolding a DbContext from an existing database is also included (`Microsoft.EntityFrameworkCore.SqlServer.Design`). A *ScaffoldingReadMe.txt* file is created which you can delete.

Once again, right-click the **Controllers** folder in **Solution Explorer** and select **Add > New Scaffolded Item**.

In the **Add Scaffold** dialog box:

Select **MVC controller with views, using Entity Framework**.

Click **Add**.

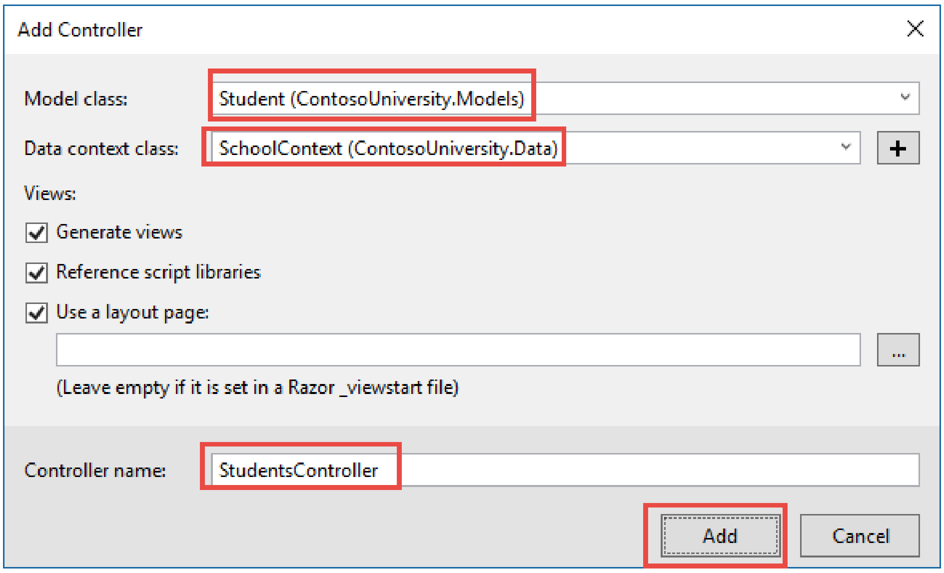
In the **Add Controller** dialog box:

In **Model class** select **Student**.

In **Data context class** select **SchoolContext**.

Accept the default **StudentsController** as the name.

Click **Add**.



When you click **Add**, the Visual Studio scaffolding engine creates a *StudentsController.cs* file and a set of views (*.cshtml* files) that work with the controller.

(The scaffolding engine can also create the database context for you if you don't create it manually first as you did earlier for this tutorial. You can specify a new context class in the **Add Controller** box by clicking the plus sign to the right of **Data context class**. Visual Studio will then create your `DbContext` class as well as the controller and views.)

You'll notice that the controller takes a `SchoolContext` as a constructor parameter.

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET dependency injection will take care of passing an instance of `SchoolContext` into the controller. You configured that in the *Startup.cs* file earlier.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

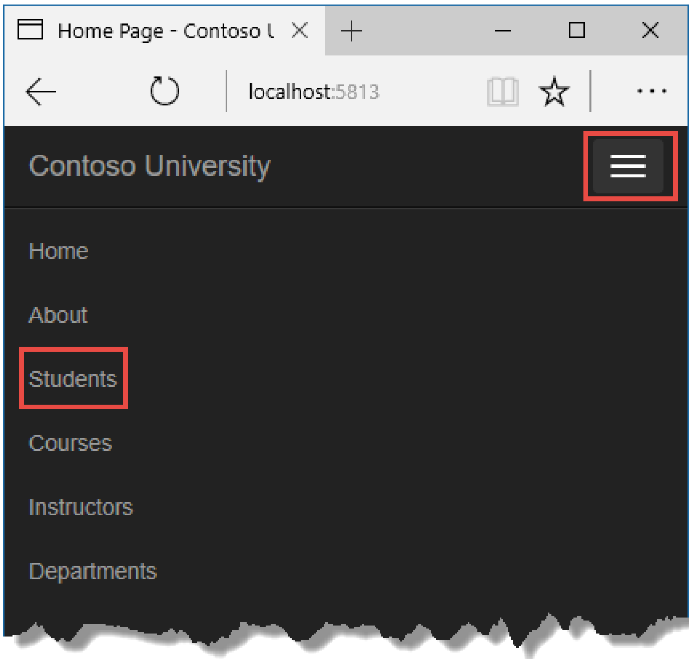
You'll learn about the asynchronous programming elements in this code later in the tutorial.

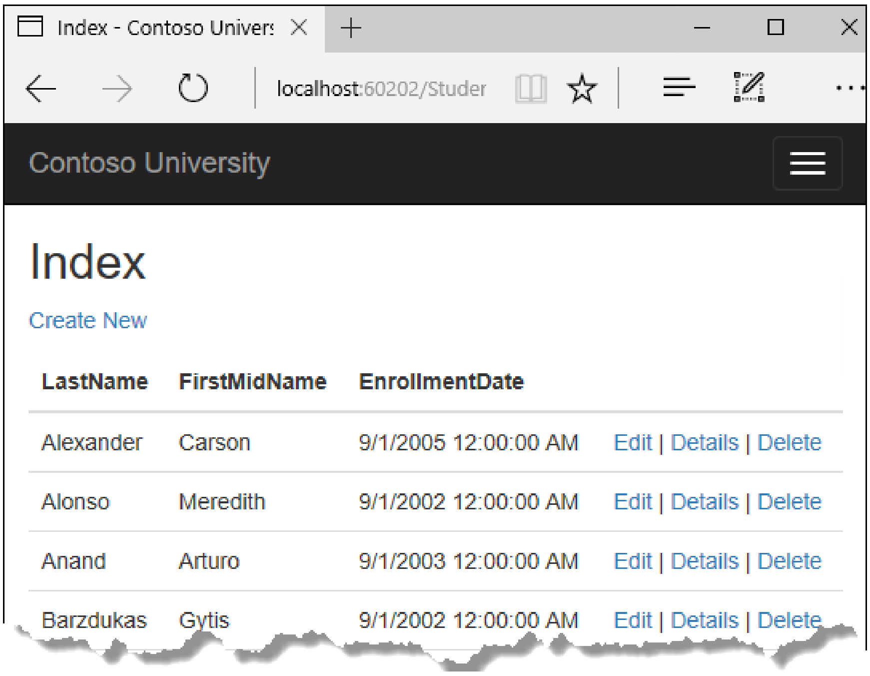
The *Views/Students/Index.cshtml* view displays this list in a table:

```
@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>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Student` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.





View the Database

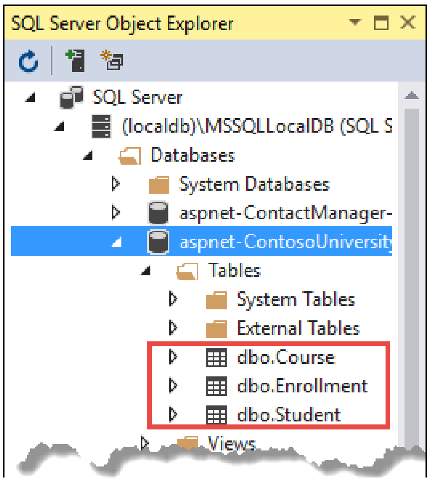
When you started the application, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database and so it created one, then the remainder of the `Initialize` method code populated the database with data. You can use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio.

Close the browser.

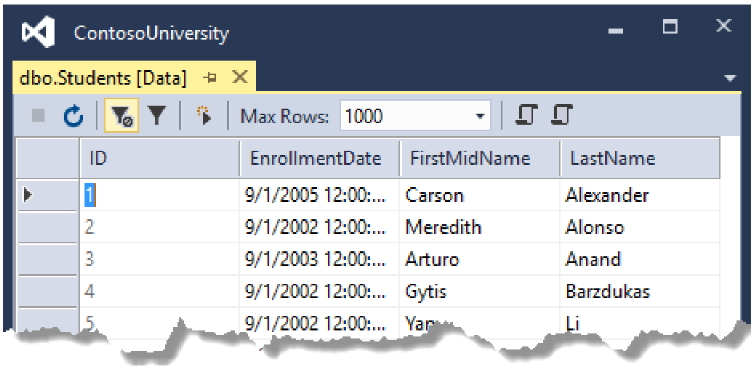
If the SSOX window isn't already open, select it from the **View** menu in Visual Studio.

In SSOX, click **(localdb)\MSSQLLocalDB > Databases**, and then click the entry for the database name that is in the connection string in your *appsettings.json* file.

Expand the **Tables** node to see the tables in your database.



Right-click the **Student** table and click **View Data** to see the columns that were created and the rows that were inserted into the table.



The *.mdf* and *.ldf* database files are in the *C:Users* folder.

Because you're calling `EnsureCreated` in the initializer method that runs on app start, you could now make a change to the `Student class`, delete the database, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, you'll see a new `EmailAddress` column in the re-created table.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of conventions, or assumptions that the Entity Framework makes.

The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.

Entity property names are used for column names.

Entity properties that are named ID or classnameID are recognized as primary key properties.

A property is interpreted as a foreign key property if it's named (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, you can explicitly specify table names, as you saw earlier in this tutorial. And you can set column names and set any property as primary key or foreign key, as you'll see in a later tutorial in this series.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that is returned.

The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.

The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.

`ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when you are writing asynchronous code that uses the Entity Framework:

Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It does not include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.

An EF context is not thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.

If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see Async Overview.

Summary

You've now created a simple application that uses the Entity Framework Core and SQL Server Express LocalDB to store and display data. In the following tutorial, you'll learn how to perform basic CRUD (create, read, update, delete) operations.

NEXT

Create, Read, Update, and Delete - EF Core with ASP.NET Core MVC tutorial (2 of 10)

By Tom Dykstra and Rick Anderson

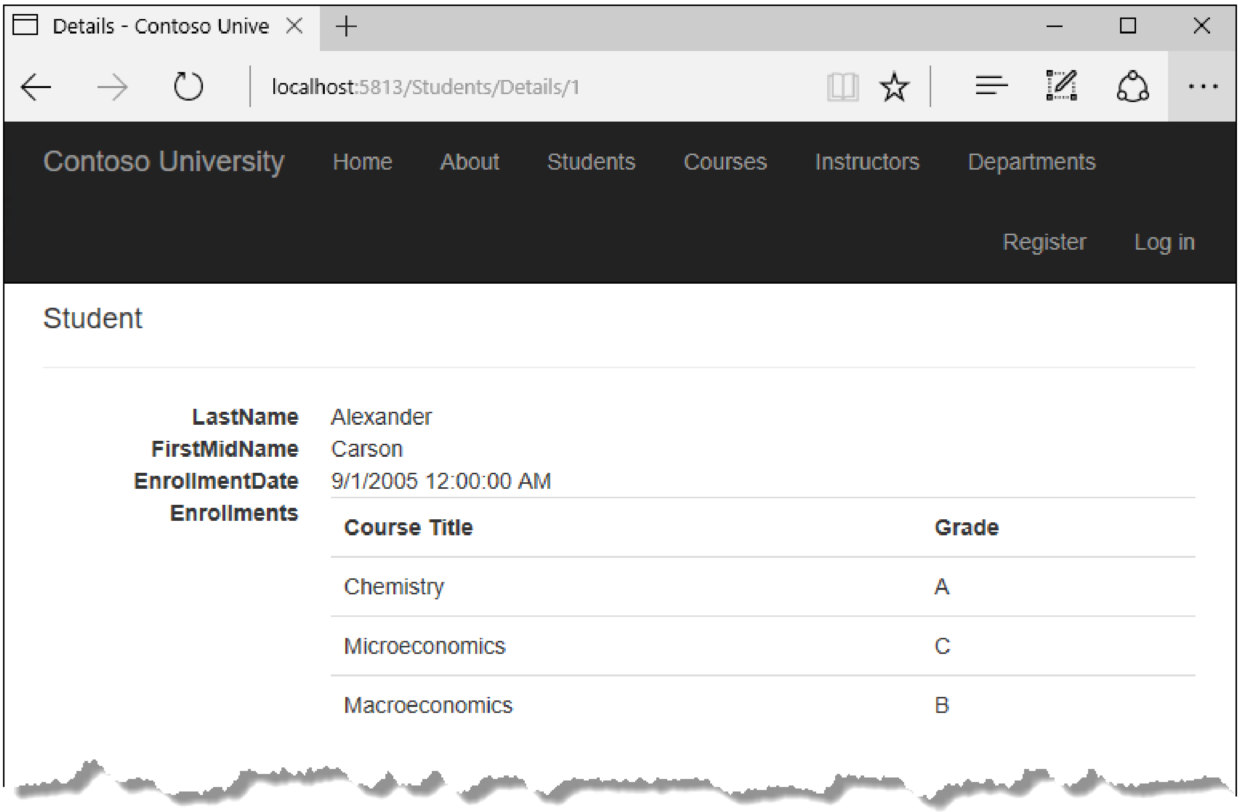
The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

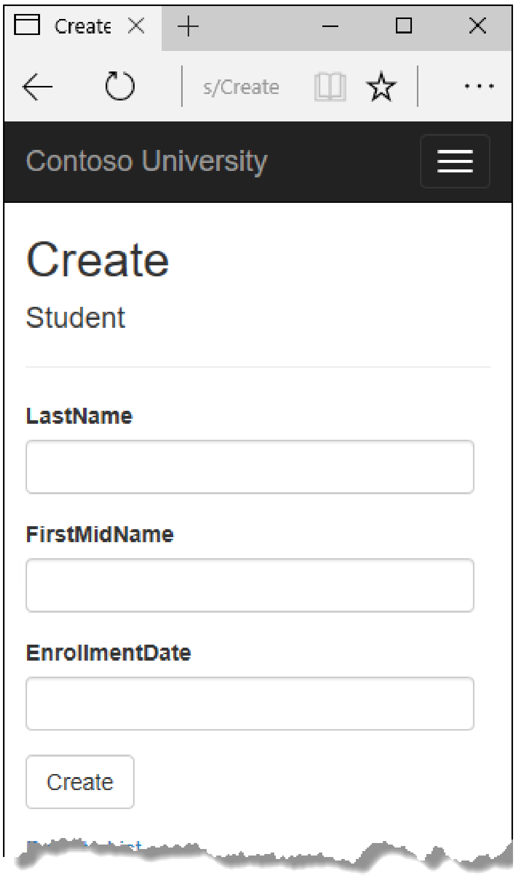
In the previous tutorial you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

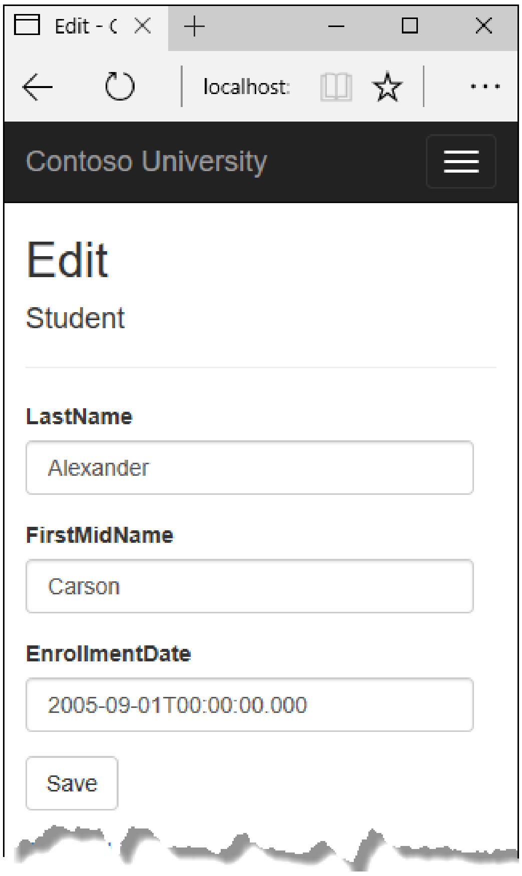
Note

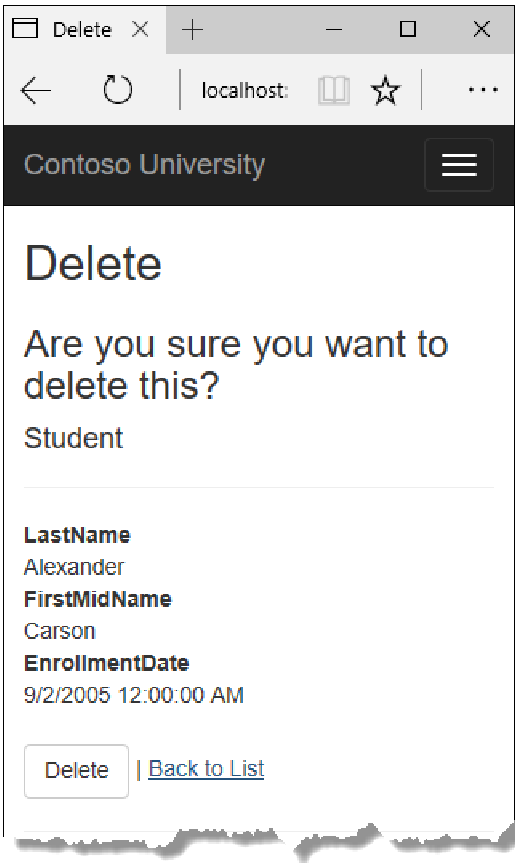
It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see the last tutorial in this series.

In this tutorial, you'll work with the following web pages:




Create
s/Create
Contoso University
Create
Student
LastName
FirstMidName
EnrollmentDate
Create


Edit - C
localhost:
Contoso University
Edit
Student
LastName
Alexander
FirstMidName
Carson
EnrollmentDate
2005-09-01T00:00:00.000
Save



Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page you'll display the contents of the collection in an HTML table.

In *Controllers/StudentsController.cs*, the action method for the Details view uses the `SingleOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls `Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

```
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);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the reading related data tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned will not be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that is passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

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

DbInitializer.Initialize(context);
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Details` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see Tag helpers in ASP.NET Core.

Add enrollments to the Details view

Open *Views/Students/Details.cshtml*. Each field is displayed using `DisplayNameFor` and `DisplayFor` helper, as shown in the following example:

```
<dt>
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd>
    @Html.DisplayFor(model => model.LastName)
</dd>
```

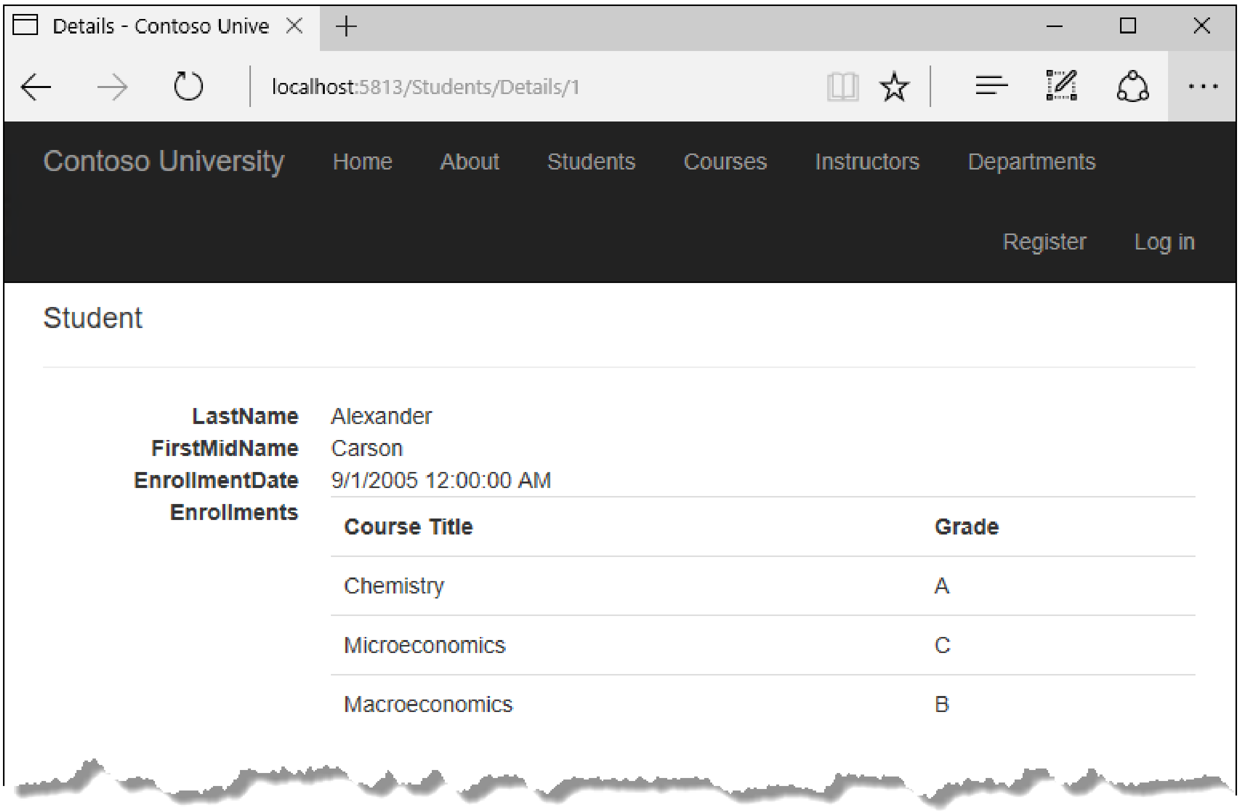
After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

```
<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>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the application, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades for the selected student:



Update the Create page

In *StudentsController.cs*, modify the HttpPost `Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

```
[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("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);
}
```

This code adds the Student entity created by the ASP.NET MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

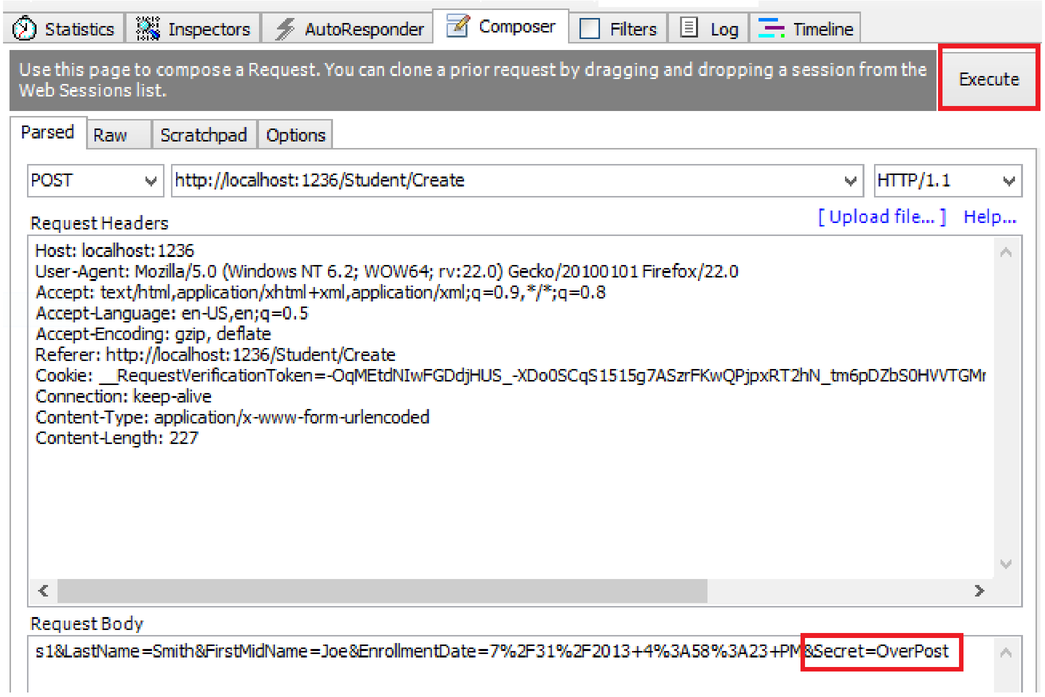
The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information about CSRF, see Anti-Request Forgery.

Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

```
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; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That is the method used in these tutorials.

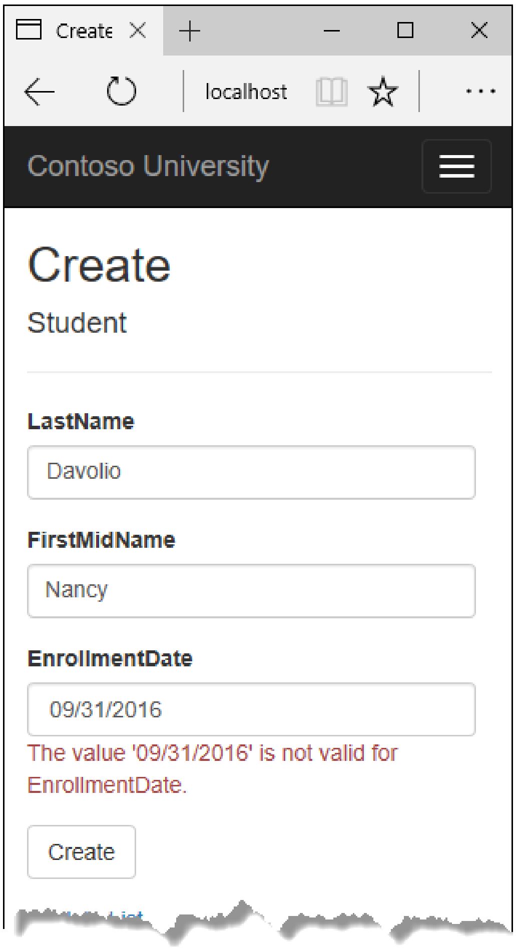
An alternative way to prevent overposting that is preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that is included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in *Views/Students/Create.cshtml* uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the page by selecting the **Students** tab and clicking **Create New**.

Enter names and an invalid date and click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

```
[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("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);
}
```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In *StudentController.cs*, the HttpGet `Edit` method (the one without the `HttpPost` attribute) uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

```
[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("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);
}
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code is not recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are whitelisted in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost `Edit` method is the same as the HttpGet `Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended HttpPost edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

```
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("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);
}
```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

`Added`. The entity does not yet exist in the database. The `SaveChanges` method issues an INSERT statement.

`Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.

`Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.

`Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.

`Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

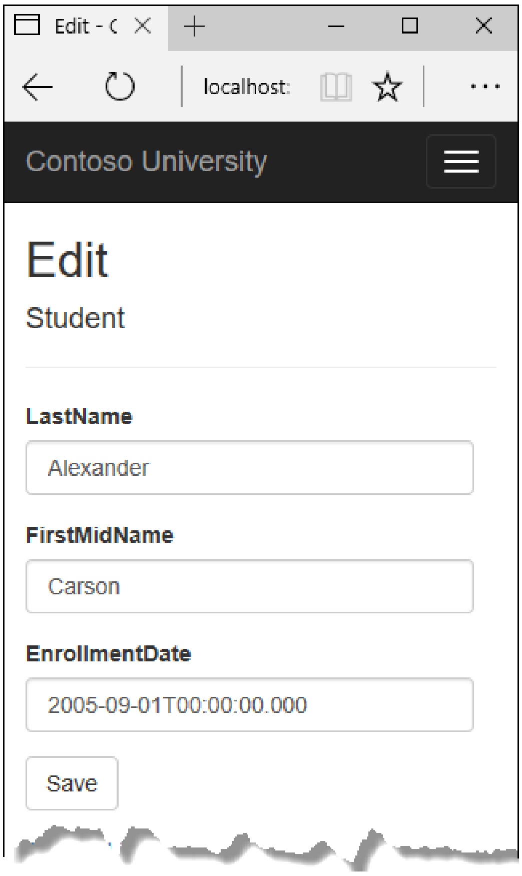
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative HttpPost Edit code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they are available when the HttpPost `Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the application and select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In *StudentController.cs*, the template code for the HttpGet `Delete` method uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the HttpPost `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete method calls the HttpGet Delete method, passing it a parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the HttpGet `Delete` action method with the following code, which manages error reporting.

```
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);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet `Delete` method is called without a previous failure. When it is called by the HttpPost `Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to HttpPost Delete

Replace the HttpPost `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[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("Index");
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("Delete", new { id = id, saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

```
[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("Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("Delete", new { id = id, saveChangesError = true });
    }
}
```

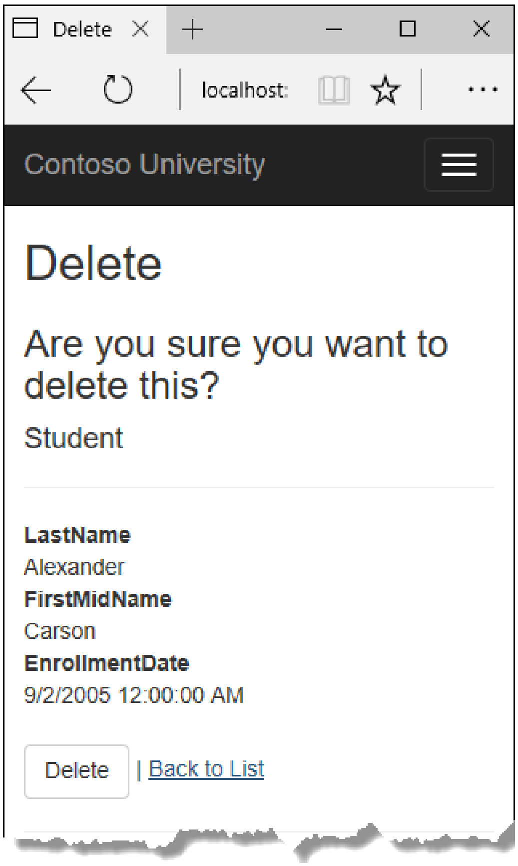
If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

Update the Delete view

In *Views/Student/Delete.cshtml*, add an error message between the h2 heading and the h3 heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the page by selecting the **Students** tab and clicking a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Closing database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in dependency injection takes care of that task for you.

In *Startup.cs* you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handling Transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.

You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.

You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Summary

You now have a complete set of pages that perform simple CRUD operations for Student entities. In the next tutorial you'll expand the functionality of the **Index** page by adding sorting, filtering, and paging.

PREVIOUS NEXT

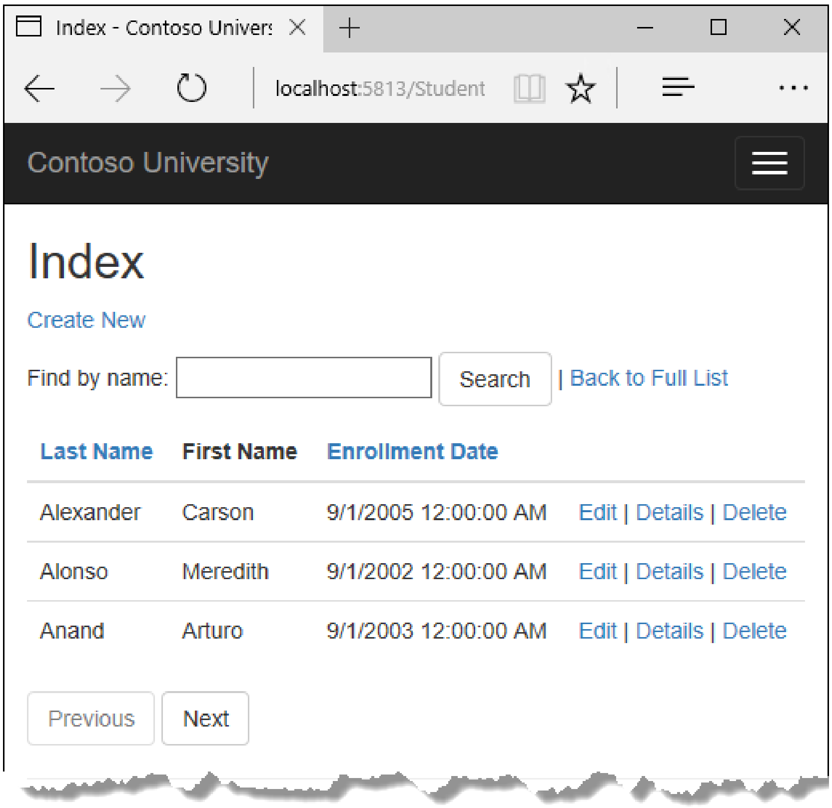
Sorting, filtering, paging, and grouping - EF Core with ASP.NET Core MVC tutorial (3 of 10)

By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.11 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



Add Column Sort Links to the Students Index Page

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code to the Student Index view.

Add sorting Functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code:

```
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());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

```
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());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that is not executed until the `return View` statement.

This code could get verbose with a large number of columns. The last tutorial in this series shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

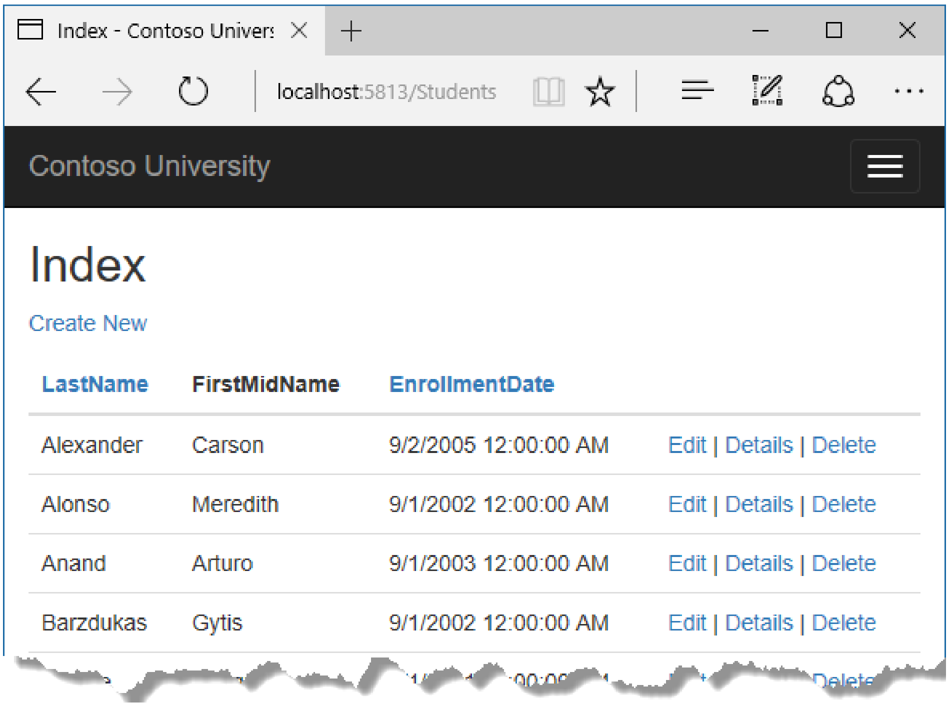
Add column heading hyperlinks to the Student Index view

Replace the code in *Views/Students/Index.cshtml*, with the following code to add column heading hyperlinks. The changed lines are highlighted.

```
@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>
```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the page and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search Box to the Students Index page

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code (the changes are highlighted).

```
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());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

▯ Note

Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there is a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

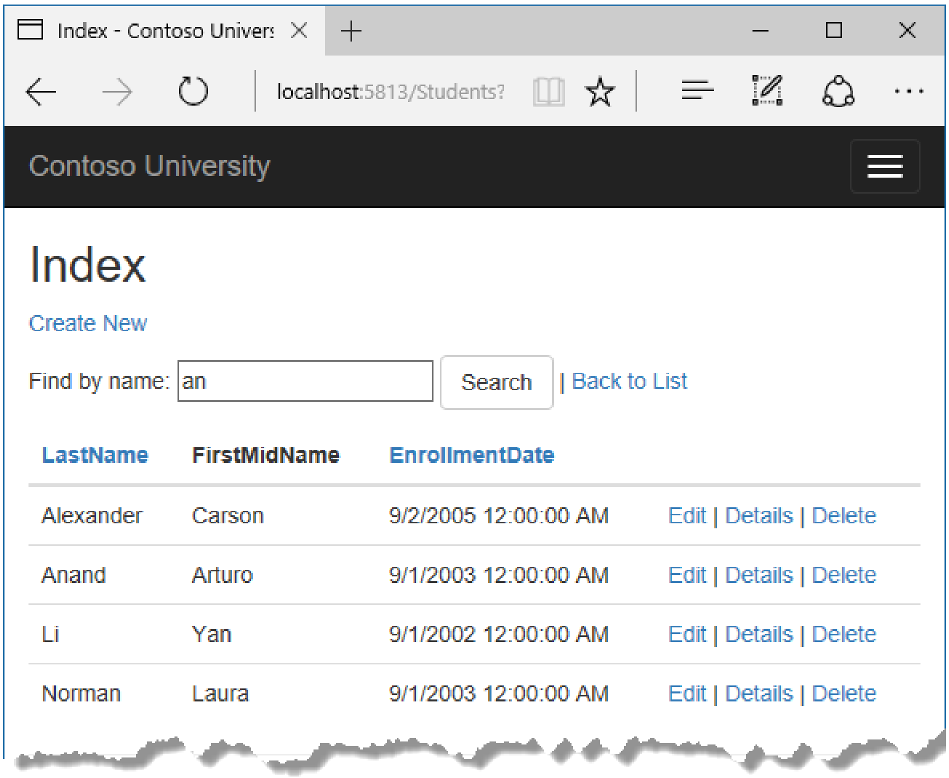
Add a Search Box to the Student Index View

In *Views/Student/Index.cshtml*, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

```
<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">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action does not result in an update.

Run the page, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

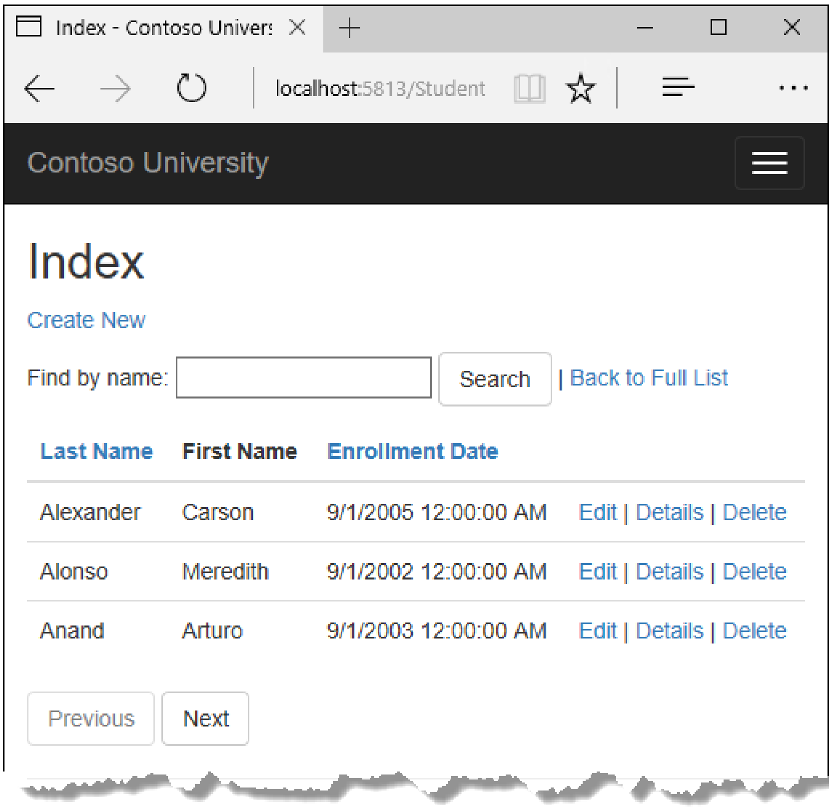
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging functionality to the Students Index page

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder create `PaginatedList.cs`, and then replace the template code with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

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);
    }
}
```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code.

```
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));
}
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter is not null.

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add paging links to the Student Index view

In *Views/Students/Index.cshtml*, replace the existing code with the following code. The changes are highlighted.

```
@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>
```

```
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</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>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex - 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="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

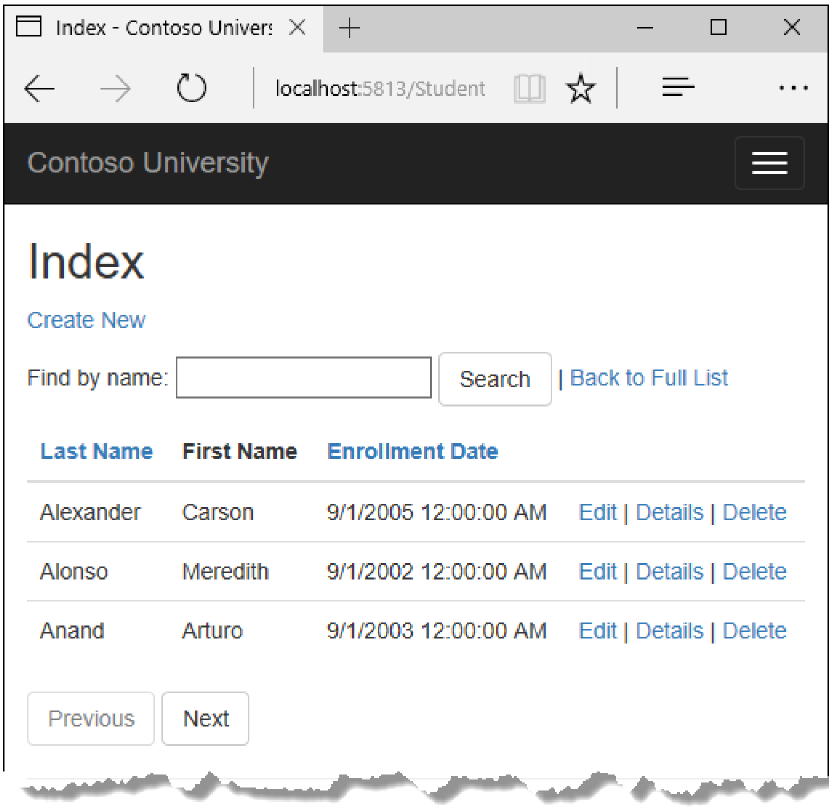
The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-page="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled btn">
   Previous
</a>
```

Run the page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page that shows Student statistics

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

Create a view model class for the data that you need to pass to the view.

Modify the About method in the Home controller.

Modify the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file EnrollmentDateGroup.cs and replace the template code with the following code:

```
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; }
    }
}
```

Modify the Home Controller

In *HomeController.cs*, add the following using statements at the top of the file:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
```

Replace the `About` method with the following code:

```
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());
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Note

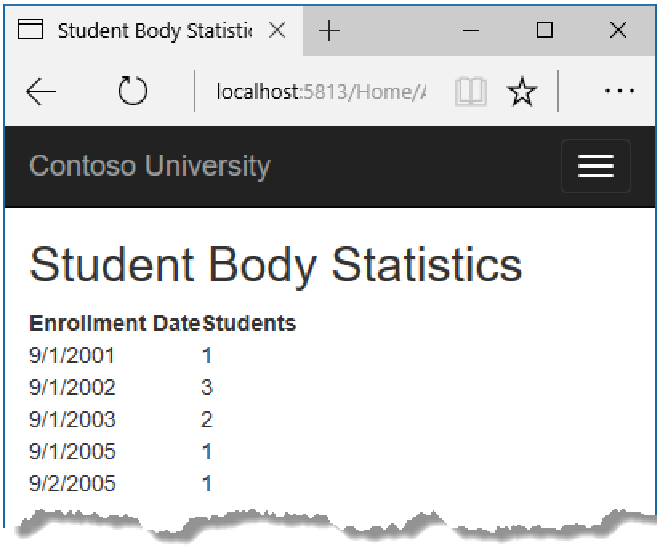
In the 1.0 version of Entity Framework Core, the entire result set is returned to the client, and grouping is done on the client. In some scenarios this could create performance problems. Be sure to test performance with production volumes of data, and if necessary use raw SQL to do the grouping on the server. For information about how to use raw SQL, see the last tutorial in this series.

Modify the About View

Replace the code in the *Views/Home/About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    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)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and click the **About** link. The count of students for each enrollment date is displayed in a table.



Summary

In this tutorial you've seen how to perform sorting, filtering, paging, and grouping. In the next tutorial you'll learn how to handle data model changes by using migrations.

PREVIOUS NEXT

Migrations - EF Core with ASP.NET Core MVC tutorial (4 of 10)

By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

Introduction to migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

Entity Framework Core NuGet packages for migrations

To work with migrations, you can use the **Package Manager Console** (PMC) or the command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at the end of this tutorial.

The EF tools for the command-line interface (CLI) are provided in Microsoft.EntityFrameworkCore.Tools.DotNet. To install this package, add it to the `DotNetCliToolReference` collection in the *.csproj* file, as shown. **Note:** You have to install this package by editing the *.csproj* file; you can't use the `install-package` command or the package manager GUI. You can edit the *.csproj* file by right-clicking the project name in **Solution Explorer** and selecting **Edit ContosoUniversity.csproj**.

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="1.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
</ItemGroup>
```

(The version numbers in this example were current when the tutorial was written.)

Change the connection string

In the *appsettings.json* file, change the name of the database in the connection string to ContosoUniversity2 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

This change sets up the project so that the first migration will create a new database. This isn't required for getting started with migrations, but you'll see later why it's a good idea.

 Note

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

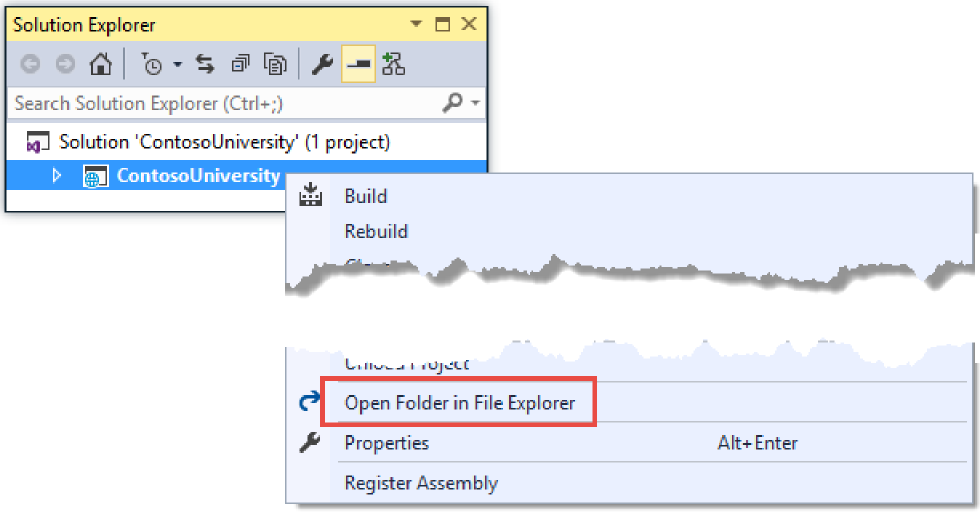
```
dotnet ef database drop
```

The following section explains how to run CLI commands.

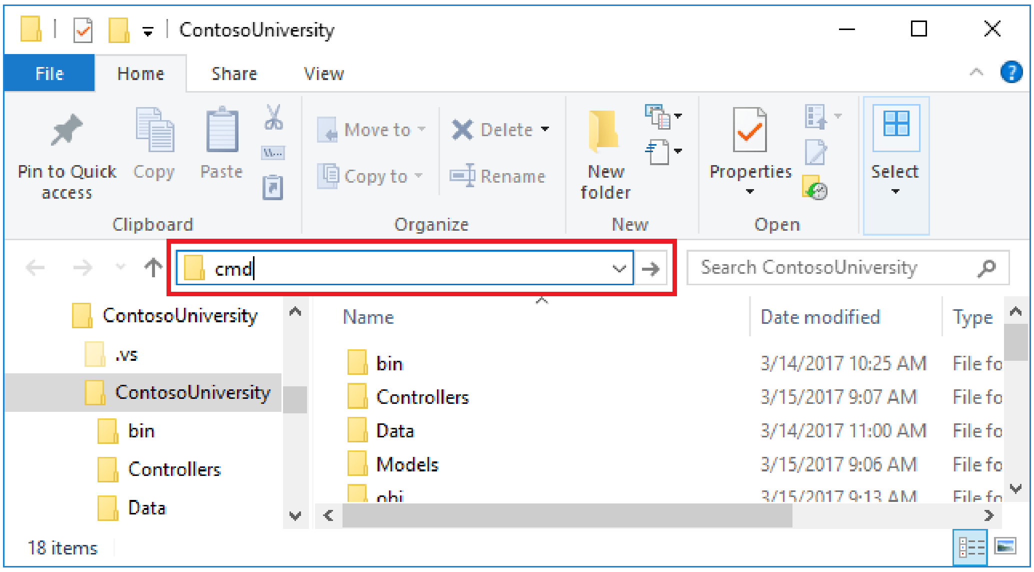
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

In **Solution Explorer**, right-click the project and choose **Open in File Explorer** from the context menu.



Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

```
dotnet ef migrations add InitialCreate
```

You see output like the following in the command window:

```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:15.63
Done. To undo this action, use 'ef migrations remove'
```

▯ Note

If you see an error message *No executable found matching command "dotnet-ef"*, see this blog post for help troubleshooting.

If you see an error message "*cannot access the file ... ContosoUniversity.dll because it is being used by another process.*", find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity > Stop Site**.

Examine the Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named *_InitialCreate.cs*. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Student",
            columns: table => new
            {
                ID = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                EnrollmentDate = table.Column<DateTime>(nullable: false),
                FirstMidName = table.Column<string>(nullable: true),
                LastName = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Student", x => x.ID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Course");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

Examine the data model snapshot

Migrations also creates a *snapshot* of the current database schema in *Migrations/SchoolContextModelSnapshot.cs*. Here's what that code looks like:

```
[DbContext(typeof(SchoolContext))]
partial class SchoolContextModelSnapshot : ModelSnapshot
{
    protected override void BuildModel(ModelBuilder modelBuilder)
    {
        modelBuilder
            .HasAnnotation("ProductVersion", "1.1.1")
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

        modelBuilder.Entity("ContosoUniversity.Models.Course", b =>
            {
                b.Property<int>("CourseID");

                b.Property<int>("Credits");

                b.Property<string>("Title");

                b.HasKey("CourseID");

                b.ToTable("Course");
            });

        // Additional code for Enrollment and Student tables not shown

        modelBuilder.Entity("ContosoUniversity.Models.Enrollment", b =>
            {
                b.HasOne("ContosoUniversity.Models.Course", "Course")
                    .WithMany("Enrollments")
                    .HasForeignKey("CourseID")
                    .OnDelete(DeleteBehavior.Cascade);

                b.HasOne("ContosoUniversity.Models.Student", "Student")
                    .WithMany("Enrollments")
                    .HasForeignKey("StudentID")
                    .OnDelete(DeleteBehavior.Cascade);
            });
    }
}
```

Because the current database schema is represented in code, EF Core doesn't have to interact with the database to create migrations. When you add a migration, EF determines what changed by comparing the data model to the snapshot file. EF interacts with the database only when it has to update the database.

The snapshot file has to be kept in sync with the migrations that create it, so you can't remove a migration just by deleting the file named *_.cs*. If you delete that file, the remaining migrations will be out of sync with the database snapshot file. To delete the last migration that you added, use the dotnet ef migrations remove command.

Apply the migration to the database

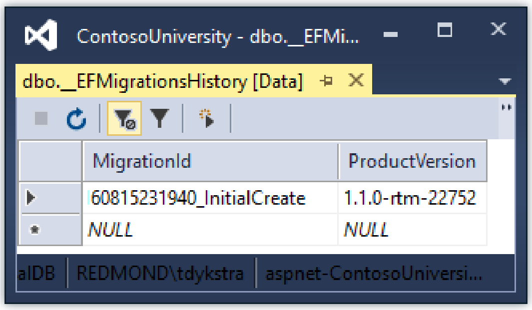
In the command window, enter the following command to create the database and tables in it.

```
dotnet ef database update
```

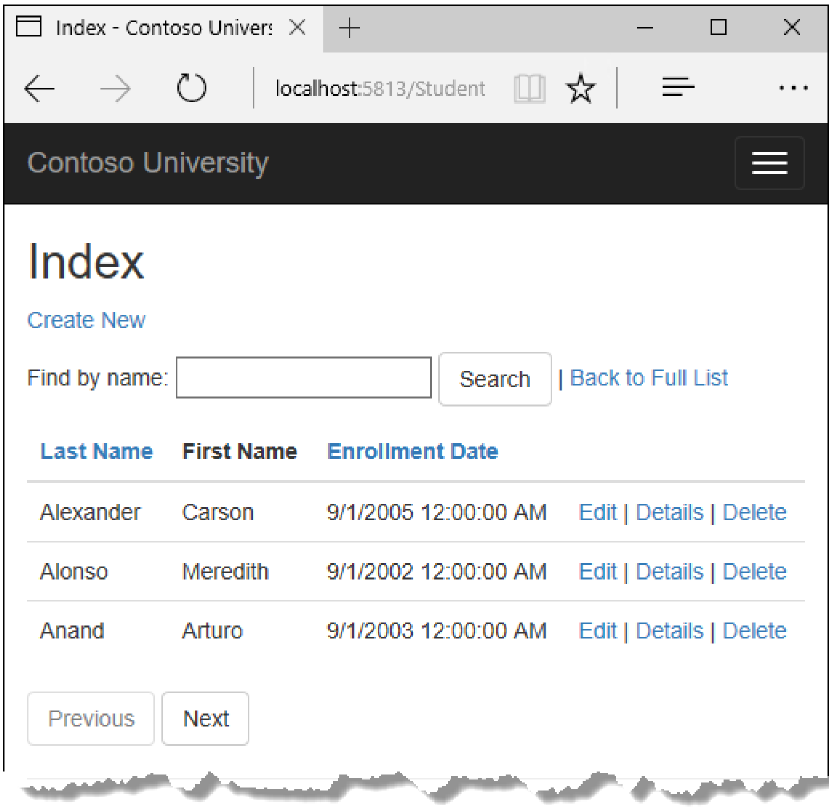
The output from the command is similar to the `migrations add` command.

```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:17.34
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one entry for the first migration.



Run the application to verify that everything still works the same as before.



Command-line interface (CLI) vs. Package Manager Console (PMC)

The EF tooling for managing migrations is available from .NET Core CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

If you want to use the PMC commands, install the Microsoft.EntityFrameworkCore.Tools package. Unlike the CLI tools, you don't have to edit the *.csproj* file; you can install it by using the **Package Manager Console** or the **NuGet Package Manager** GUI.

Note that this is not the same package as the one you install for the CLI: its name ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET Core CLI.

For more information about the PMC commands, see Package Manager Console (Visual Studio).

Summary

In this tutorial, you've seen how to create and apply your first migration. In the next tutorial, you'll begin looking at more advanced topics by expanding the data model. Along the way you'll create and apply additional migrations.

PREVIOUS NEXT

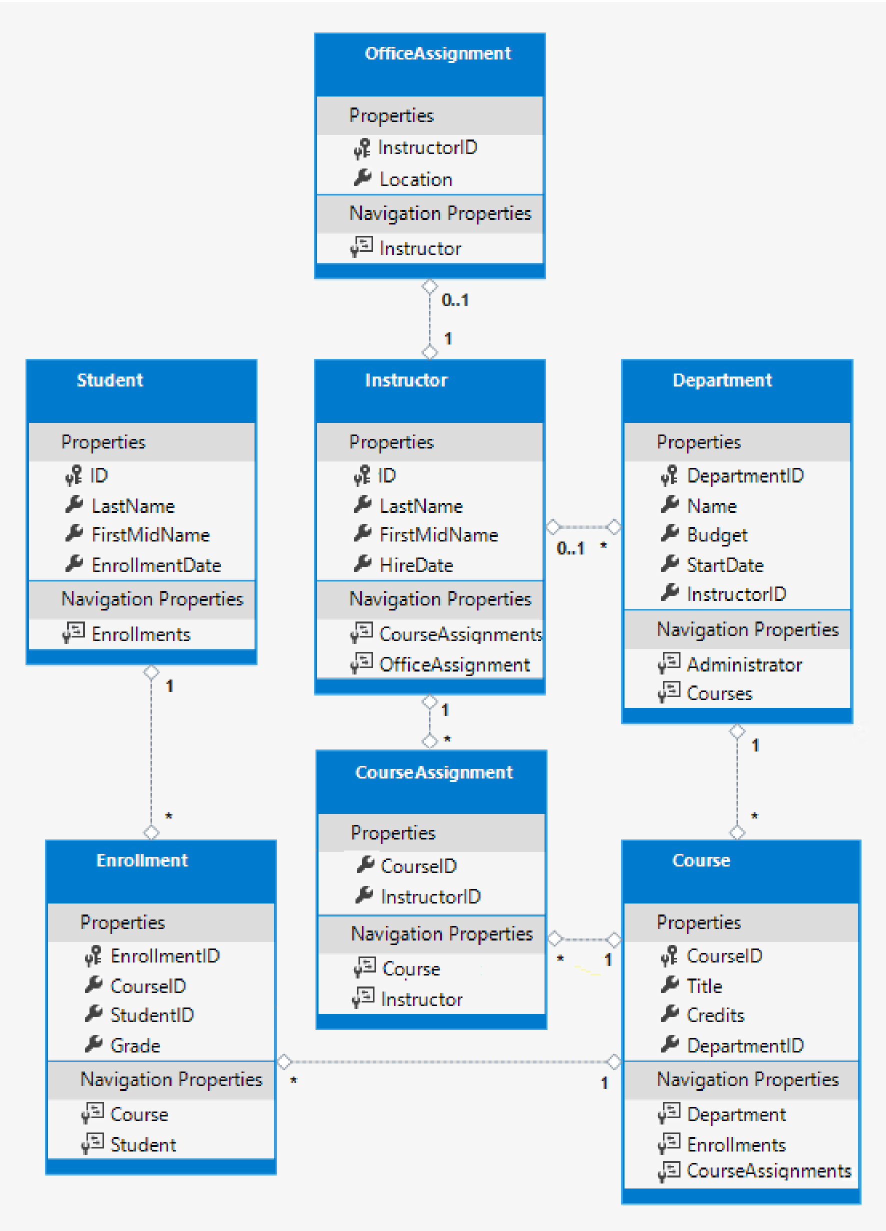
Creating a complex data model - EF Core with ASP.NET Core MVC tutorial (5 of 10)

By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorials you worked with a simple data model that was composed of three entities. In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



Customize the Data Model by Using Attributes

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models/Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
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; }
    }
}
```

The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

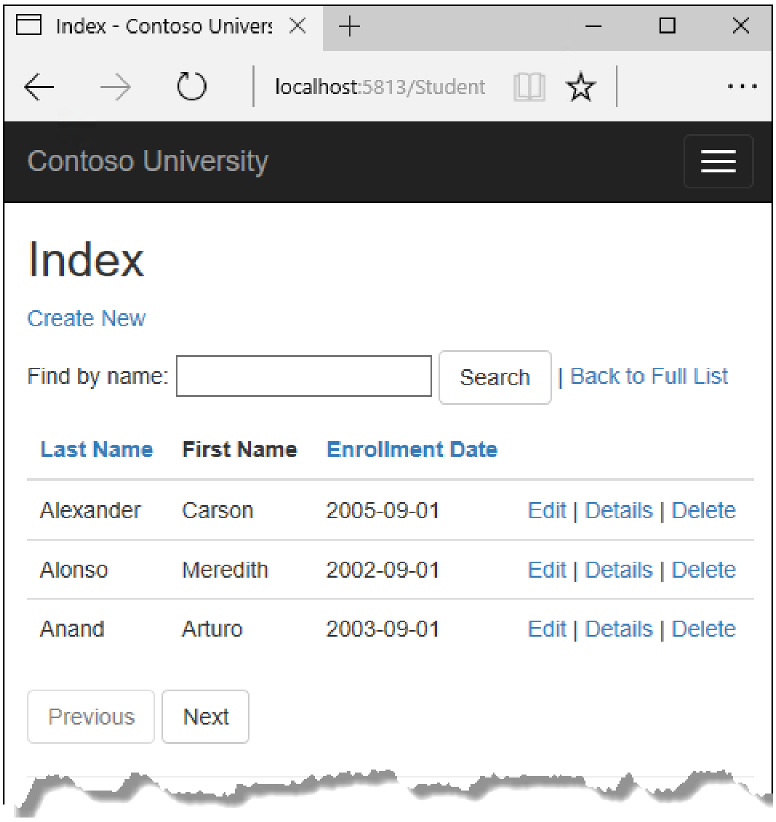
You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).

By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

Run the Students Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
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; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

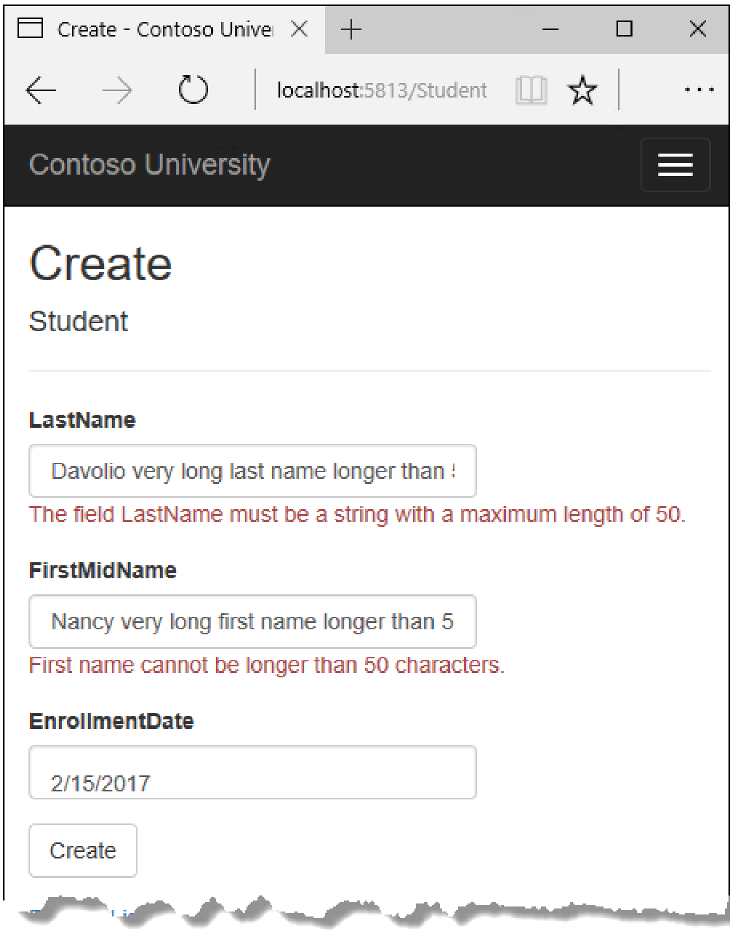
Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

```
dotnet ef migrations add MaxLengthOnNames
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named *_MaxLengthOnNames.cs*. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the Create page, and enter either name longer than 50 characters. When you click Create, client side validation shows an error message.



The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

In the *Student.cs* file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

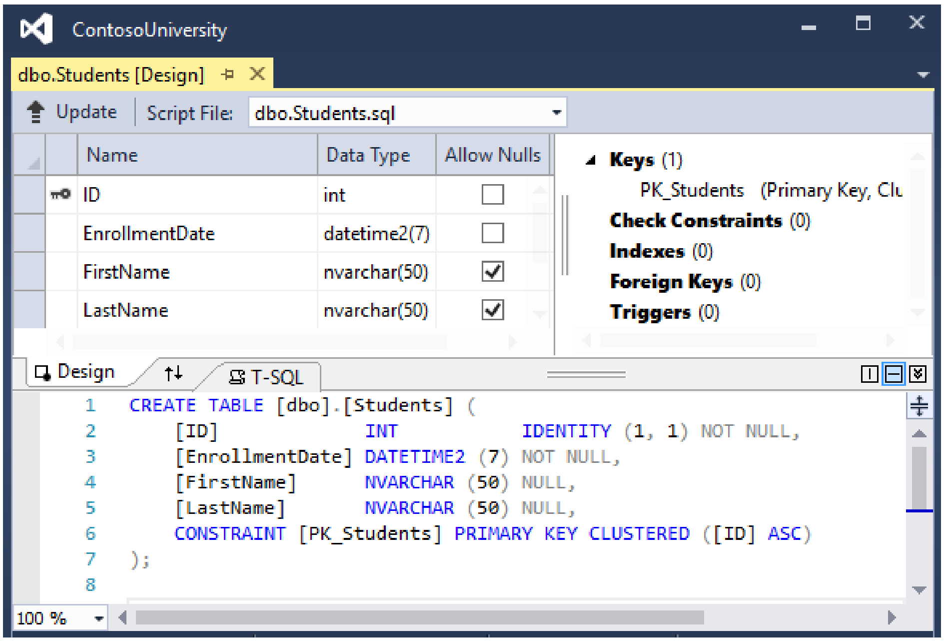
```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

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; }
    }
}
```

The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

```
dotnet ef migrations add ColumnFirstName
dotnet ef database update
```

In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.

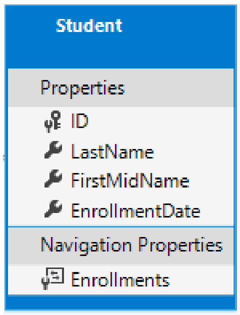


Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They are now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

Note

If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Final changes to the Student entity



In *Models/Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
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; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute is not needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

You could remove the `Required` attribute and replace it with a minimum length parameter for the `StringLength` attribute:

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

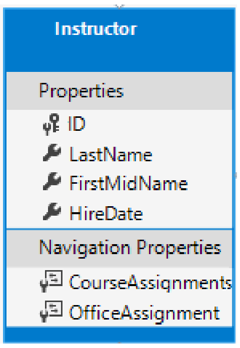
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create the Instructor Entity



Create *Models/Instructor.cs*, replacing the template code with the following code:

```
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; }
    }
}
```

Notice that several properties are the same in the Student and Instructor entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

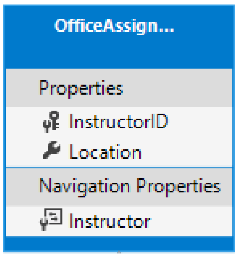
If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single OfficeAssignment entity (which may be null if no office is assigned).

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
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; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the Instructor and the OfficeAssignment entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the Instructor entity. But the Entity Framework can't automatically recognize InstructorID as the primary key of this entity because its name doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than classnameID or ID.

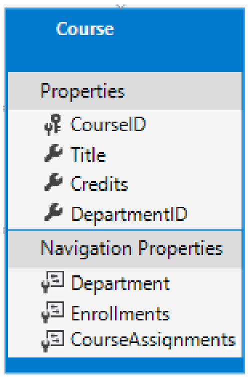
By default EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify the Course Entity



In *Models/Course.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
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; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed and creates shadow properties for them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the Department entity is null if you don't load it, so when you update the course entity, you would have to first fetch the Department entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the Department entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, the Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for Course entities, you'll use a user-specified course number such as a 1000 series for one department, a

2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key properties and navigation properties in the Course entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

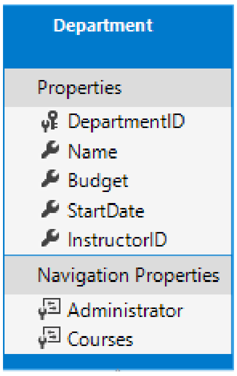
A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained later:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create the Department entity



Create *Models/Department.cs* with the following code:

```
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; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the Department entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

A department may have many courses, so there's a Courses navigation property:

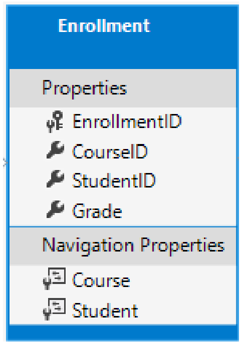
```
public ICollection<Course> Courses { get; set; }
```

Note

By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the Department.InstructorID property as nullable, EF would configure a cascade delete rule to delete the instructor when you delete the department, which is not what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

```
modelBuilder.Entity<Department>()
   .HasOne(d => d.Administrator)
   .WithMany()
   .OnDelete(DeleteBehavior.Restrict)
```

Modify the Enrollment entity



In *Models/Enrollment.cs*, replace the code you added earlier with the following code:

```
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; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

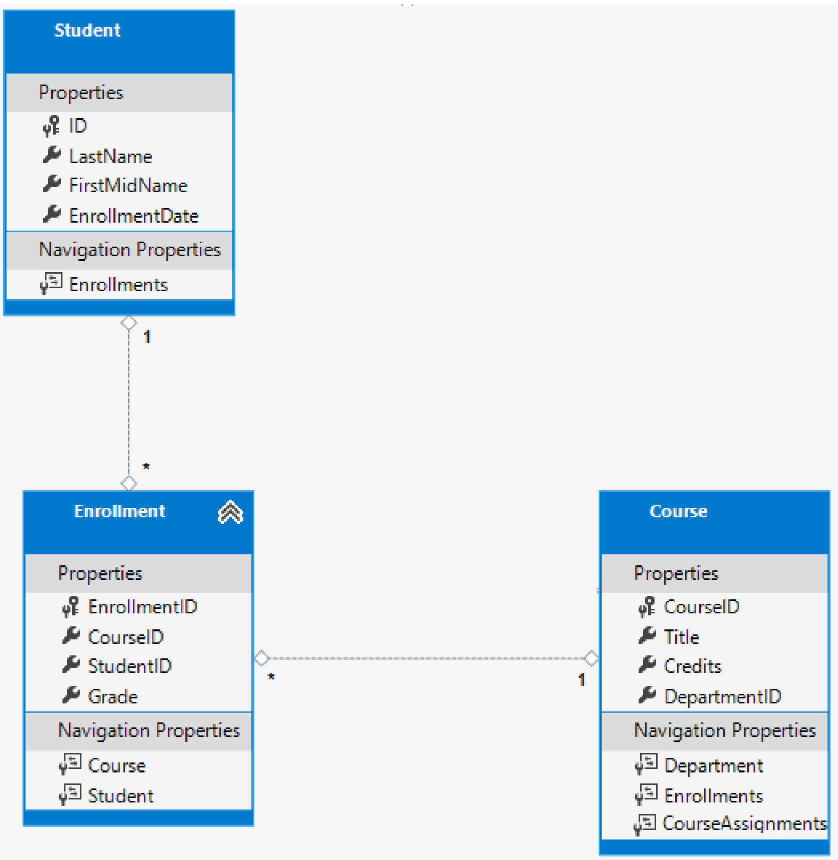
An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the Student and Course entities, and the Enrollment entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the Enrollment table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a Grade property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

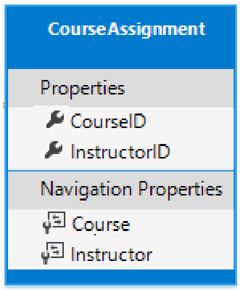


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the Enrollment table didn't include grade information, it would only need to contain the two foreign keys CourseID and StudentID. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core does not. For more information, see the discussion in the EF Core GitHub repository.)

The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

```
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; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there is no need for a separate primary key. The *InstructorID* and *CourseID* properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Add the following highlighted code to the *Data/SchoolContext.cs* file:

```
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 });
        }
    }
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

Fluent API alternative to attributes

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the EF Core documentation:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

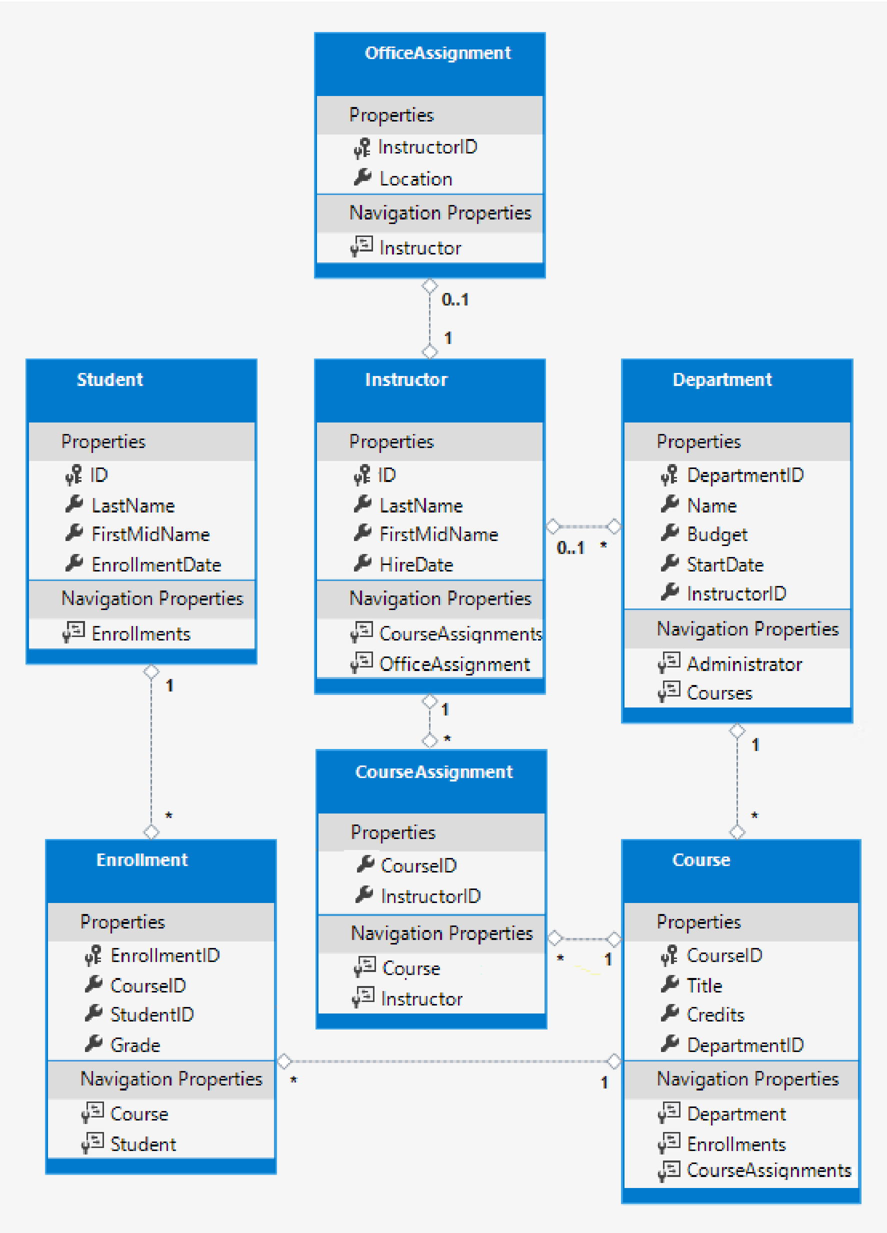
In this tutorial you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there is a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the Instructor and OfficeAssignment entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed the Database with Test Data

Replace the code in the *Data/DbInitializer.cs* file with the following code in order to provide seed data for the new entities you've created.

```
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"),
```

```
        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();
        }
    }
}
```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:11.58
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'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

> 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'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable DepartmentID foreign key to the Course table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

Open the *{timestamp}_ComplexDataModel.cs* file.

Comment out the line of code that adds the DepartmentID column to the Course table.

```
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);
```

Add the following highlighted code after the code that creates the Department table:

```
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);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string and update the database

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in *appsettings.json* to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

Save your change to *appsettings.json*.

 Note

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

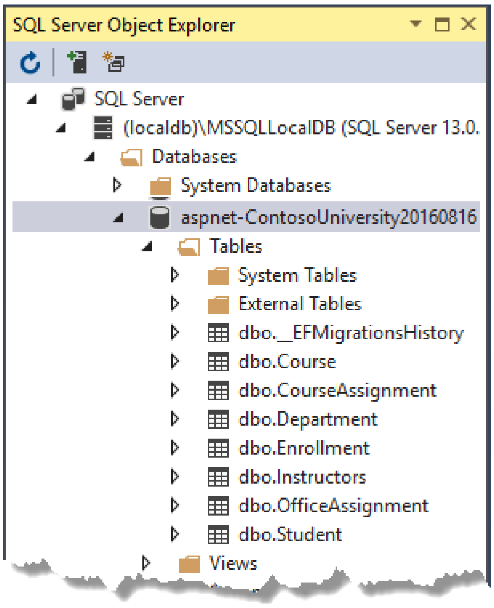
```
dotnet ef database drop
```

After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
dotnet ef database update
```

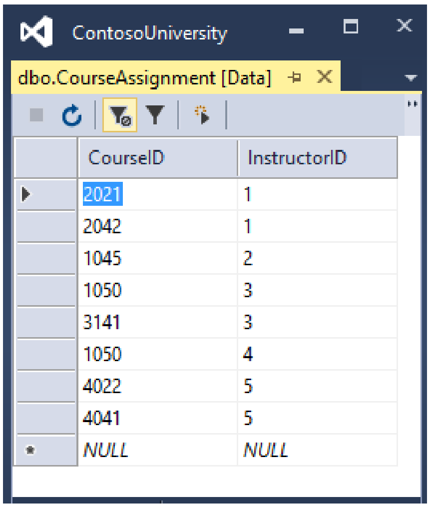
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the Refresh button.)



Run the application to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.



Summary

You now have a more complex data model and corresponding database. In the following tutorial, you'll learn more about how to access related data.

PREVIOUS NEXT

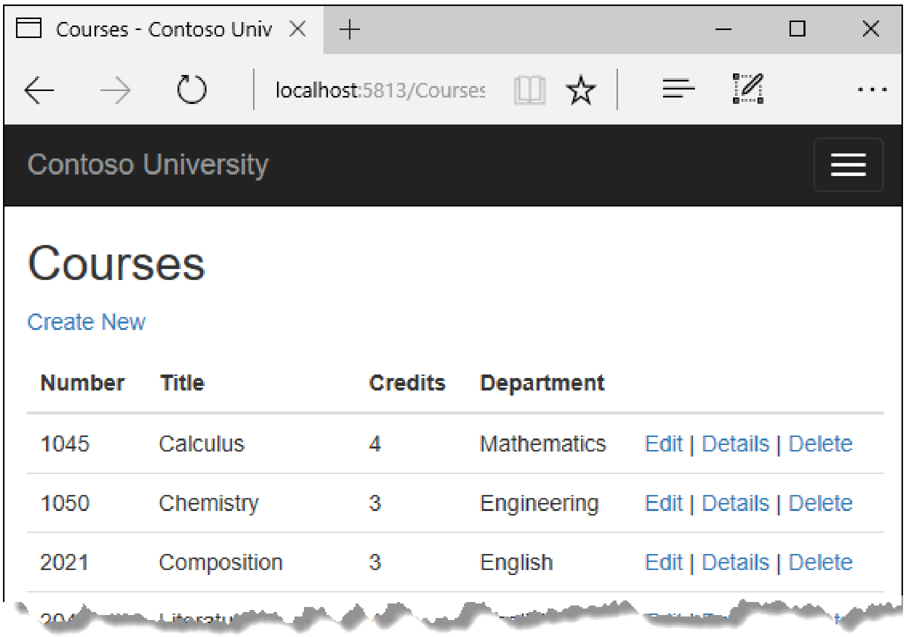
Reading related data - EF Core with ASP.NET Core MVC tutorial (6 of 10)

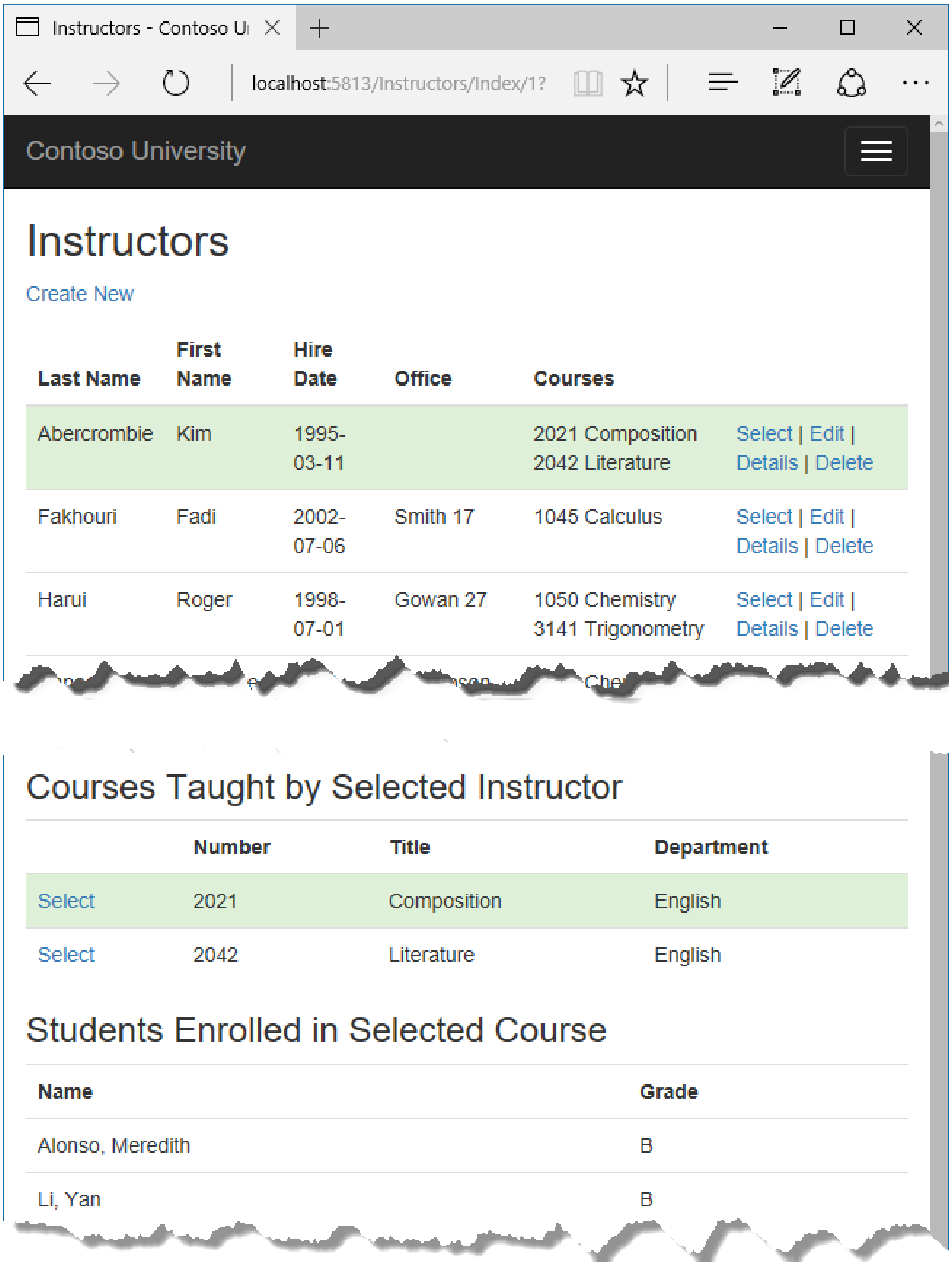
By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorial you completed the School data model. In this tutorial you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





Eager, explicit, and lazy Loading of related data

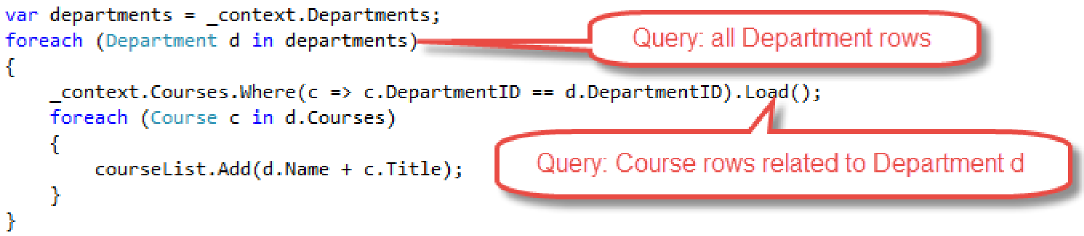
There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

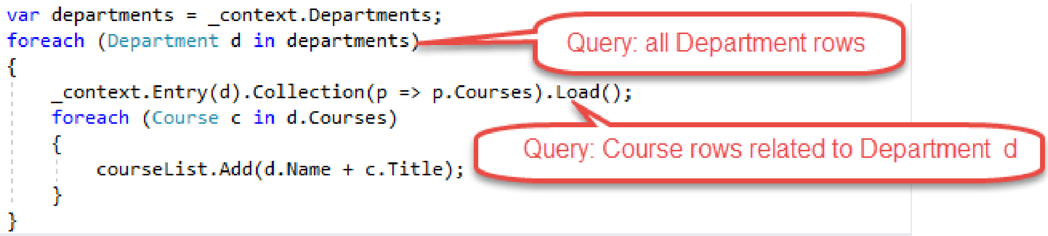
```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.



Explicit loading. When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:



Lazy loading. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 does not support lazy loading.

Performance considerations

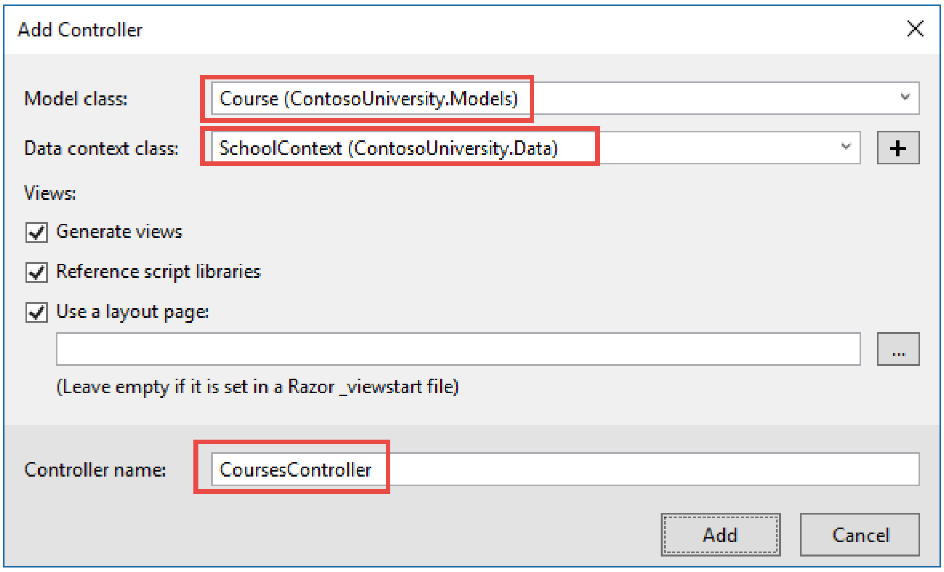
If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page that displays Department name

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that is in the `Course.Department` navigation property.

Create a controller named CoursesController for the Course entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you did earlier for the Students controller, as shown in the following illustration:



Open *CoursesController.cs* and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns Course entities (`courses` instead of `schoolContext`):

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open *Views/Courses/Index.cshtml* and replace the template code with the following code. The changes are highlighted:

```
@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>
```

You've made the following changes to the scaffolded code:

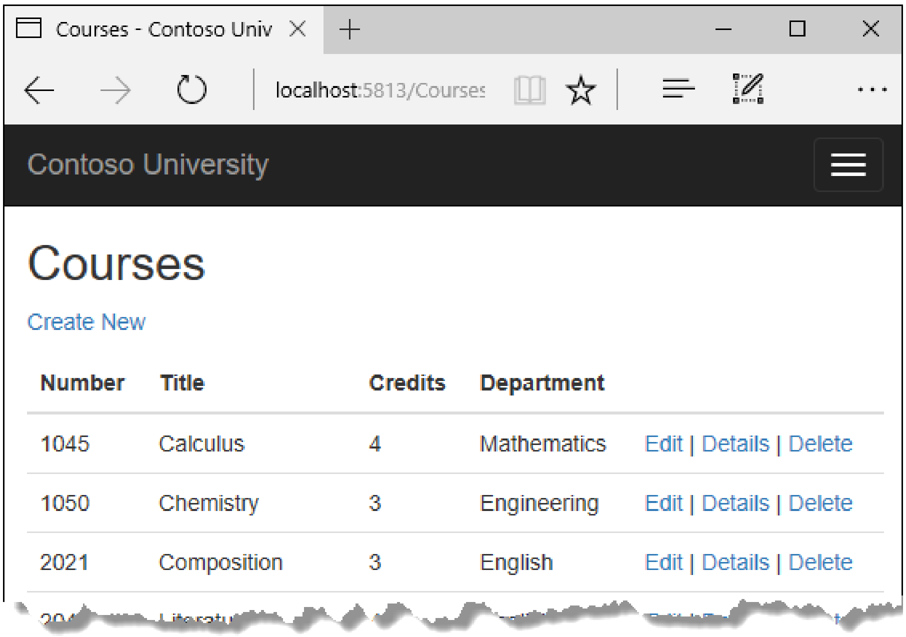
Changed the heading from Index to Courses.

Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.

Changed the **Department** column to display the department name. The code displays the `Name` property of the Department entity that's loaded into the `Department` navigation property:

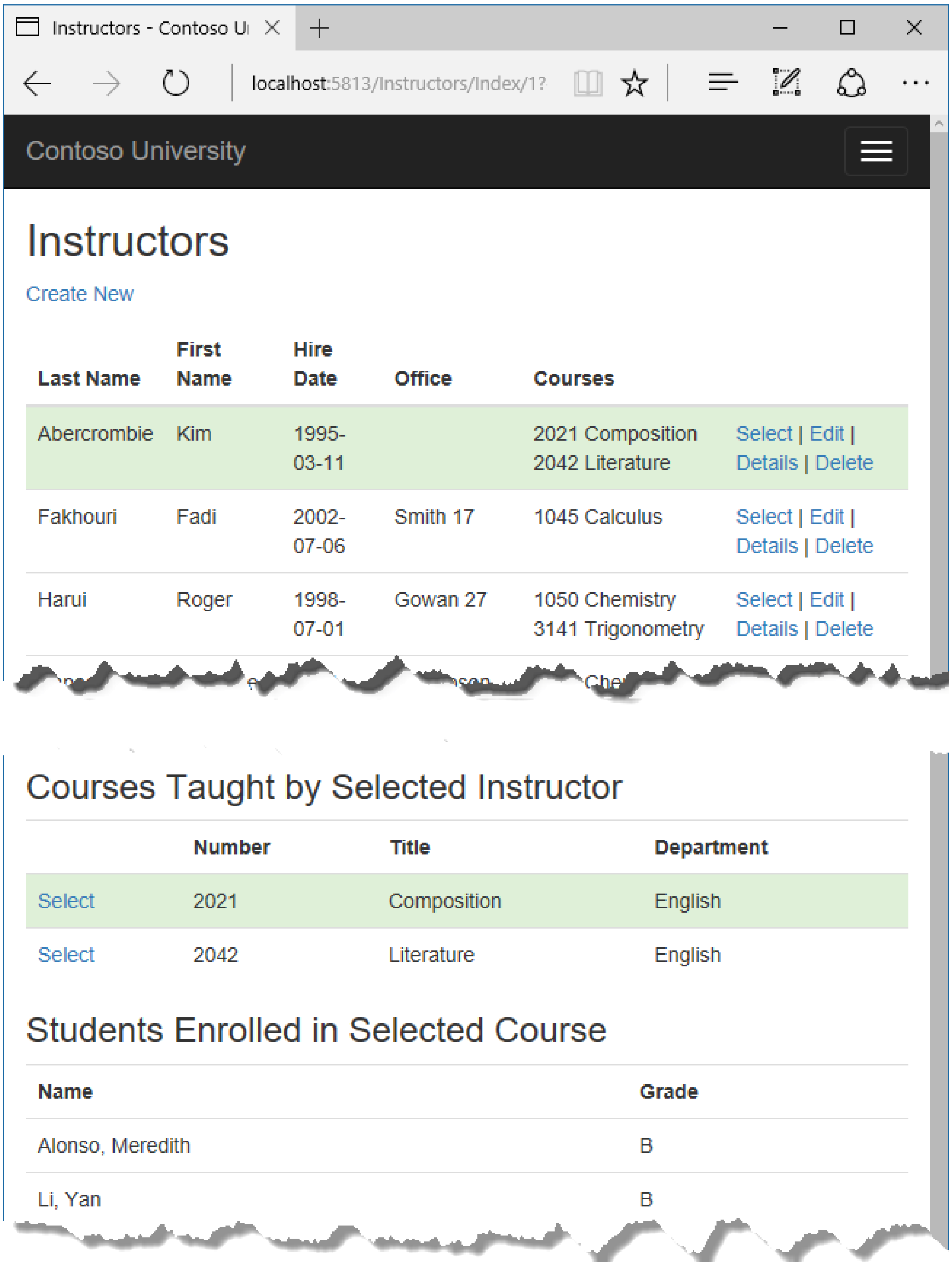
```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the page (select the Courses tab on the Contoso University home page) to see the list with department names.



Create an Instructors page that shows Courses and Enrollments

In this section you'll create a controller and view for the Instructor entity in order to display the Instructors page:



This page reads and displays related data in the following ways:

The list of instructors displays related data from the OfficeAssignment entity. The Instructor and OfficeAssignment entities are in a one-to-zero-or-one relationship. You'll use eager loading for the OfficeAssignment entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.

When the user selects an instructor, related Course entities are displayed. The Instructor and Course entities are in a many-to-many relationship. You'll use eager loading for the Course entities and their related Department entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.

When the user selects a course, related data from the Enrollments entity set is displayed. The Course and Enrollment entities are in a one-to-many relationship. You'll use separate queries for Enrollment entities and their related Student entities.

Create a view model for the Instructor Index view

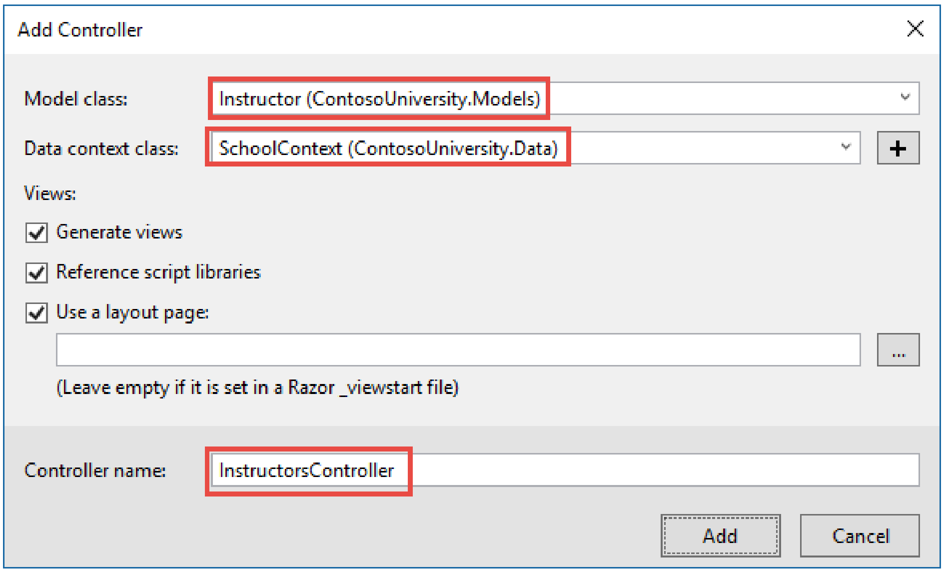
The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

In the *SchoolViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
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; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open *InstructorsController.cs* and add a using statement for the ViewModels namespace:

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

```
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);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

```
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();
```

Since the view always requires the OfficeAssignment entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `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();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `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();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the Course entities from that instructor's `CourseAssignments` navigation property.

```
#endregion

if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single Instructor entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The Single method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

```
.Single(i => i.ID == id.Value)
```

Instead of:

```
.Where(I => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

```
}

if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
```

Modify the Instructor Index view

In *Views/Instructors/Index.cshtml*, replace the template code with the following code. The changes are highlighted.

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

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

<h2>Instructors</h2>

<p>
    <a asp-action="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.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["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-action="Index" asp-route-id="@item.ID">Select</a> |
                    <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>
```

You've made the following changes to the existing code:

Changed the model class to `InstructorIndexData`.

Changed the page title from **Index** to **Instructors**.

Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

Added a **Courses** column that displays courses taught by each instructor.

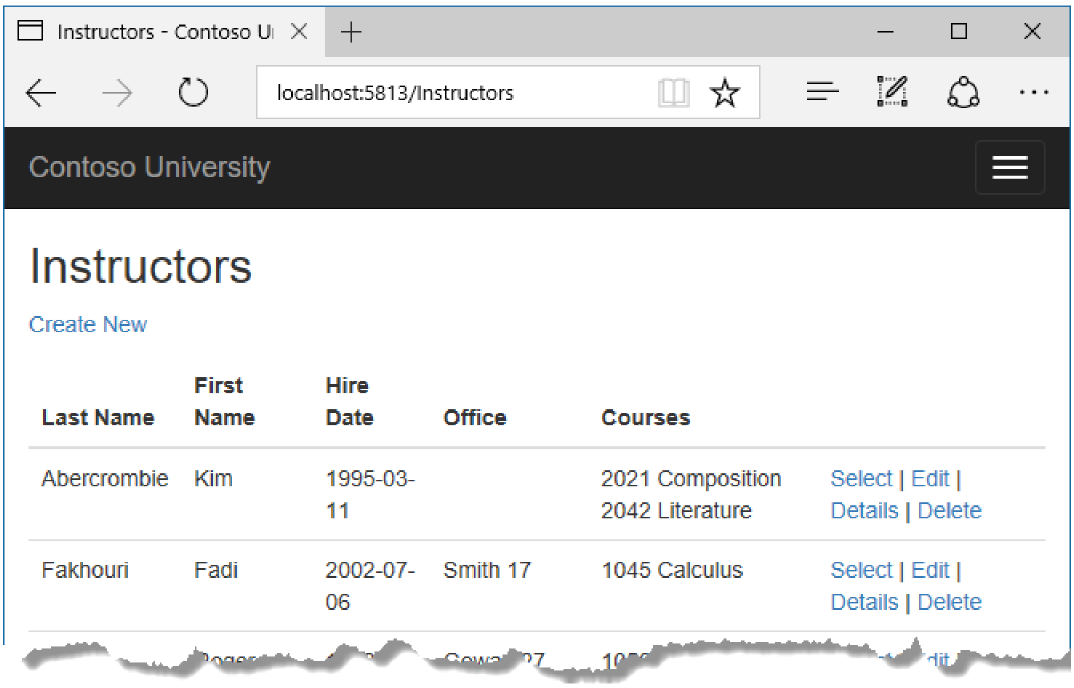
Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
```

Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the application and select the Instructors tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.

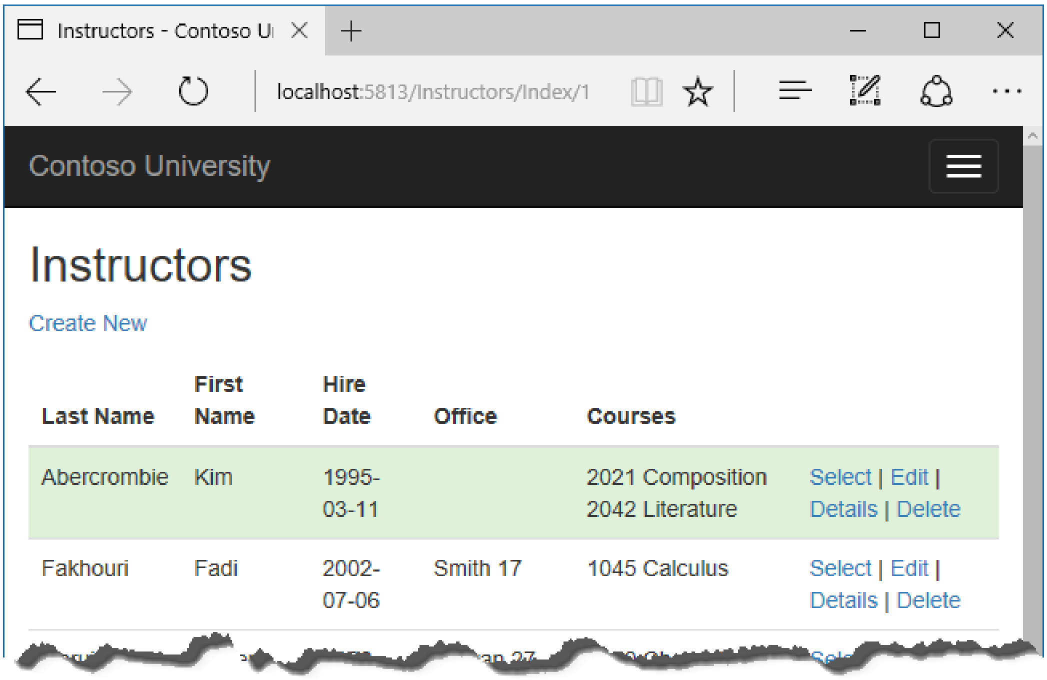


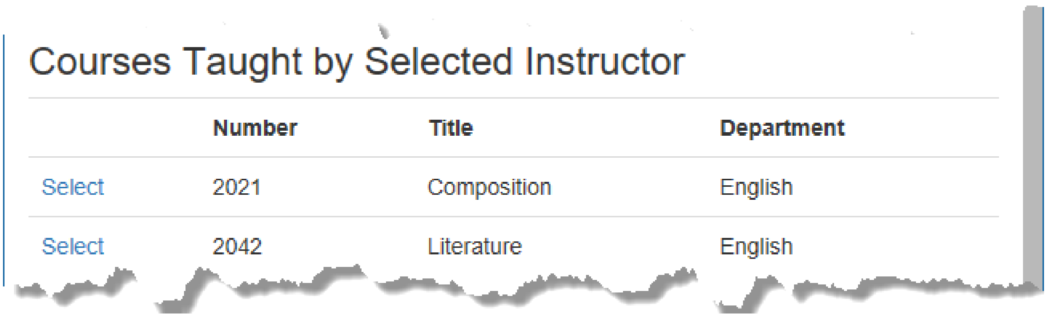
In the *Views/Instructors/Index.cshtml* file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

```
@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>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Run the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



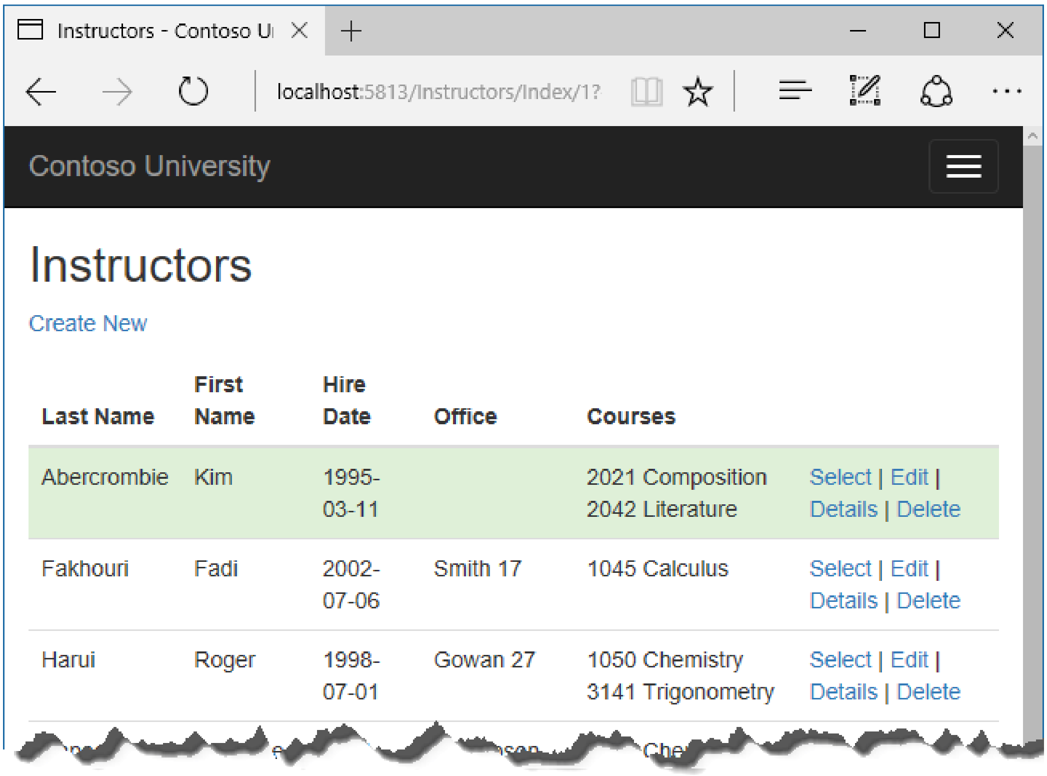


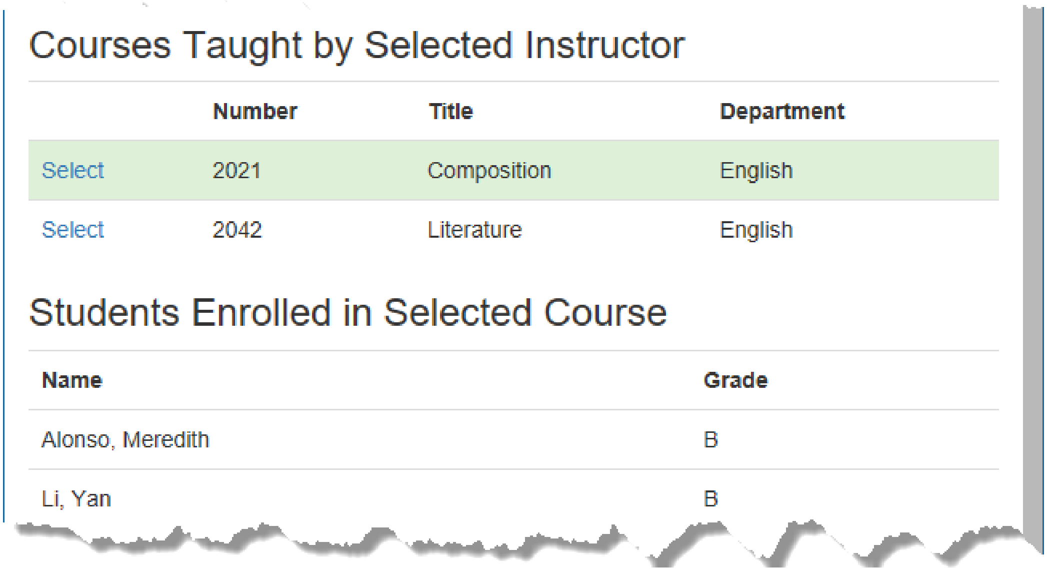
After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@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>
}
```

This code reads the Enrollments property of the view model in order to display a list of students enrolled in the course.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.





Explicit loading

When you retrieved the list of instructors in *InstructorsController.cs*, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for Enrollments and loads that property explicitly. The code changes are highlighted.

```
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);
}
```

The new code drops the *ThenInclude* method calls for enrollment data from the code that retrieves instructor entities. If an instructor and course are selected, the highlighted code retrieves Enrollment entities for the selected course, and Student entities for each Enrollment.

Run the Instructor Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Summary

You've now used eager loading with one query and with multiple queries to read related data into navigation properties. In the next tutorial you'll learn how to update related data.

PREVIOUS NEXT

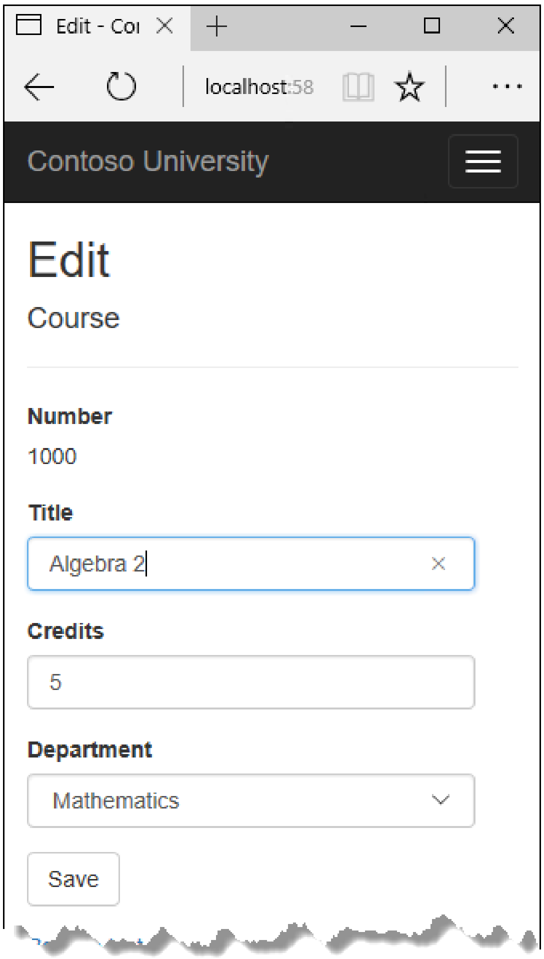
Updating related data - EF Core with ASP.NET Core MVC tutorial (7 of 10)

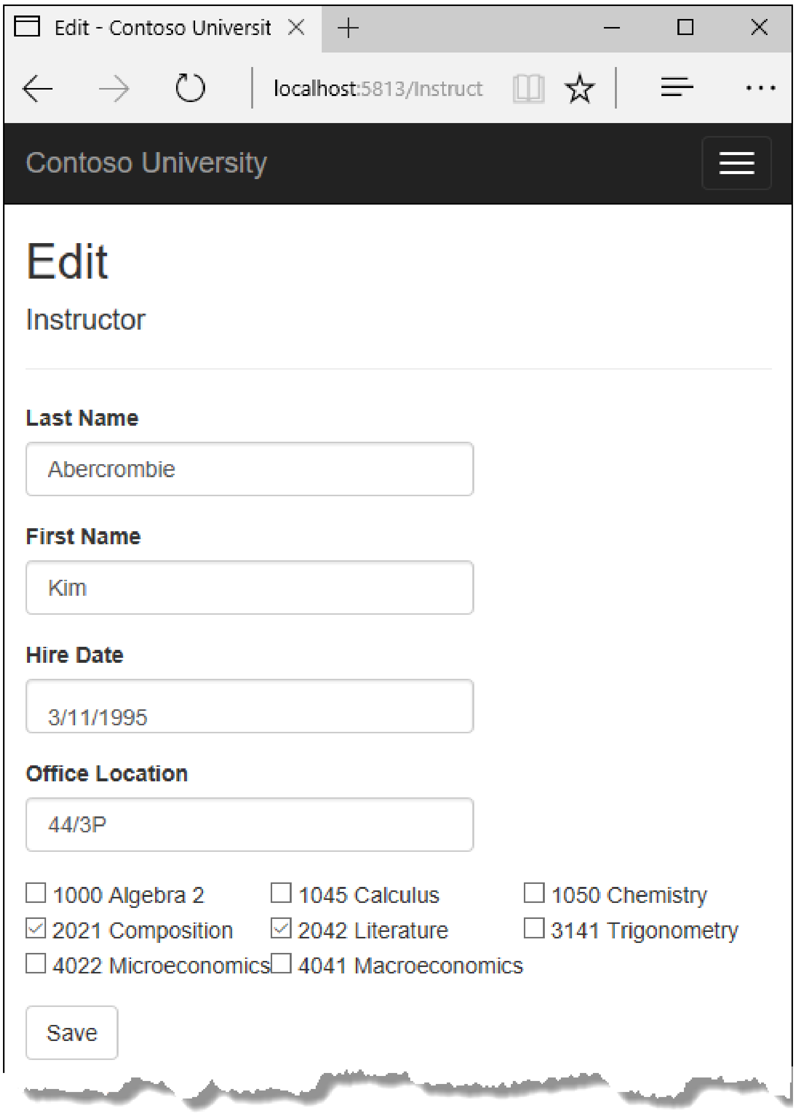
By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CoursesController.cs*, delete the four Create and Edit methods and replace them with the following code:

```
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("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("Index");
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

```
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);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

```
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);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

```
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);
}
```

Modify the Course views

In *Views/Courses/Create.cshtml*, add a "Select Department" option to the **Department** drop-down list, and change the caption for the field from **DepartmentID** to **Department**.

```
<div class="form-group">
    <label asp-for="Department" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <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" />
    </div>
</div>
```

In *Views/Courses/Edit.cshtml*, make the same change for the Department field that you just did in *Create.cshtml*.

Also in *Views/Courses/Edit.cshtml*, add a course number field before the Credits field. Because it's the primary key, it's displayed, but it can't be changed.

```
<div class="form-group">
    <label asp-for="CourseID" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        @Html.DisplayFor(model => model.CourseID)
    </div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

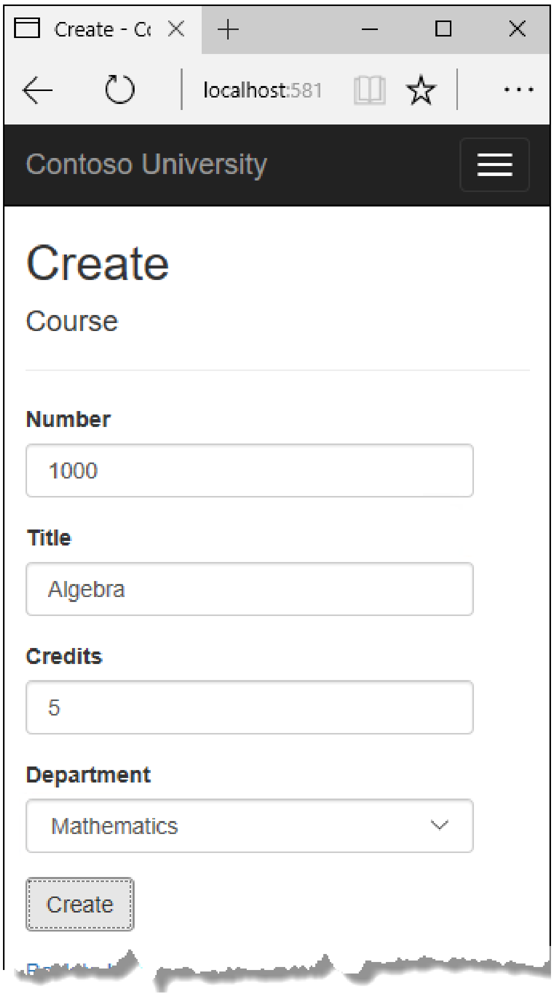
In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

```
@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>
```

In *Views/Course/Details.cshtml*, make the same change that you just did for *Delete.cshtml*.

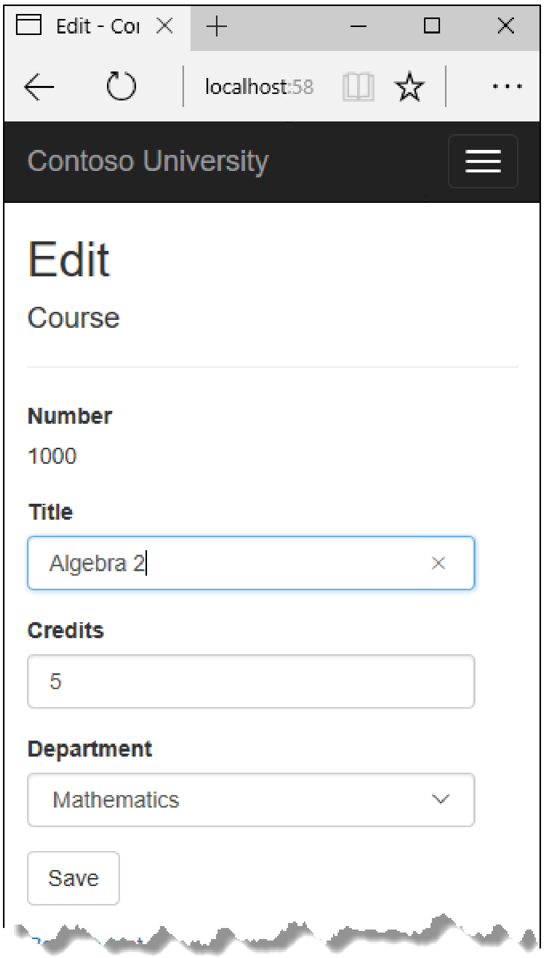
Test the Course pages

Run the **Create** page (display the Course Index page and click **Create New**) and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Run the **Edit** page (click **Edit** on a course in the Course Index page).



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The Instructor entity has a one-to-zero-or-one relationship with the OfficeAssignment entity, which means your code has to handle the following situations:

If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.

If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.

If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `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);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

```
[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("Index");
    }
    return View(instructorToUpdate);
}
```

The code does the following:

Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).

Gets the current Instructor entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the HttpGet `Edit` method.

Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload enables you to whitelist the properties you want to include. This prevents over-posting, as explained in the second tutorial.

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

If the office location is blank, sets the Instructor.OfficeAssignment property to null so that the related row in the OfficeAssignment table will be deleted.

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

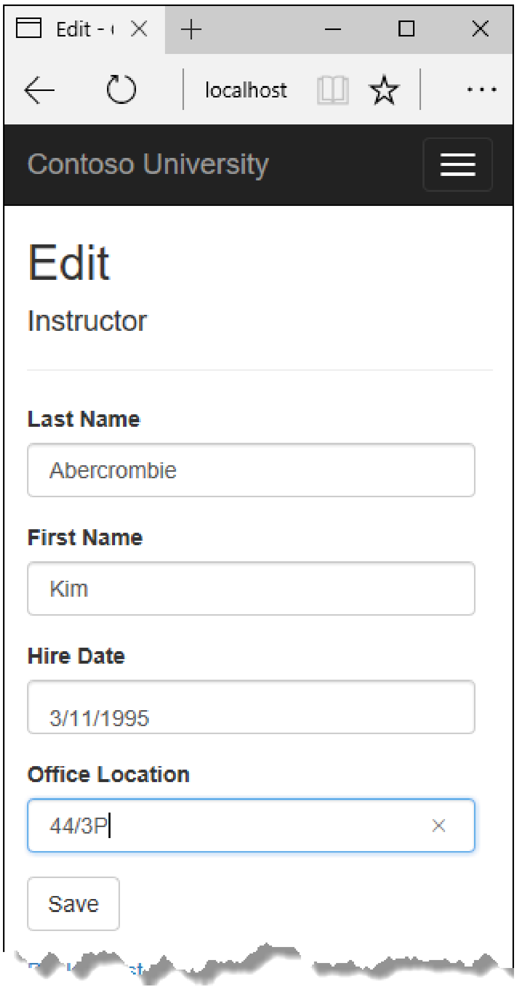
Saves the changes to the database.

Update the Instructor Edit view

In *Views/Instructors/Edit.cshtml*, add a new field for editing the office location, at the end before the **Save** button :

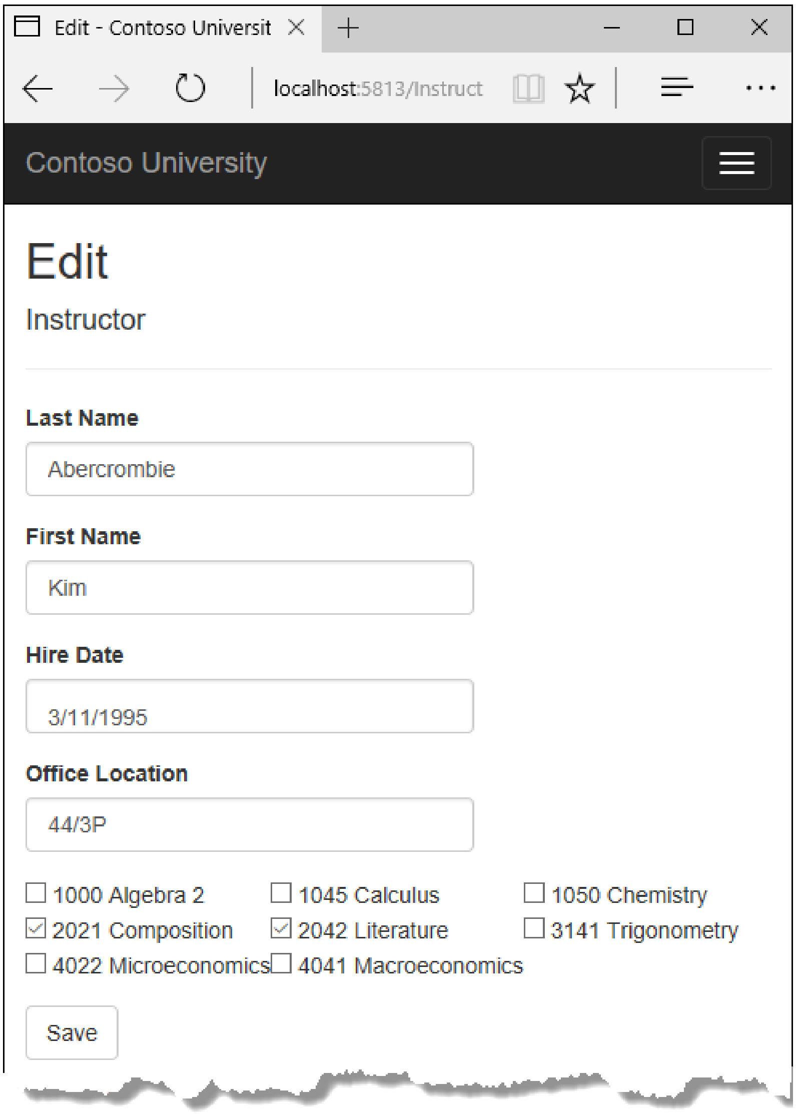
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="OfficeAssignment.Location" class="form-control" />
        <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
    </div>
</div>
```

Run the page (select the **Instructors** tab and then click **Edit** on an instructor). Change the **Office Location** and click **Save**.



Add Course assignments to the Instructor Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:



The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

```
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; }
    }
}
```

In *InstructorsController.cs*, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

```
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;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all Course entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

```
[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("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);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the whitelisting overload and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
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);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
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);
            }
        }
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
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);
            }
        }
    }
}
```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

Note

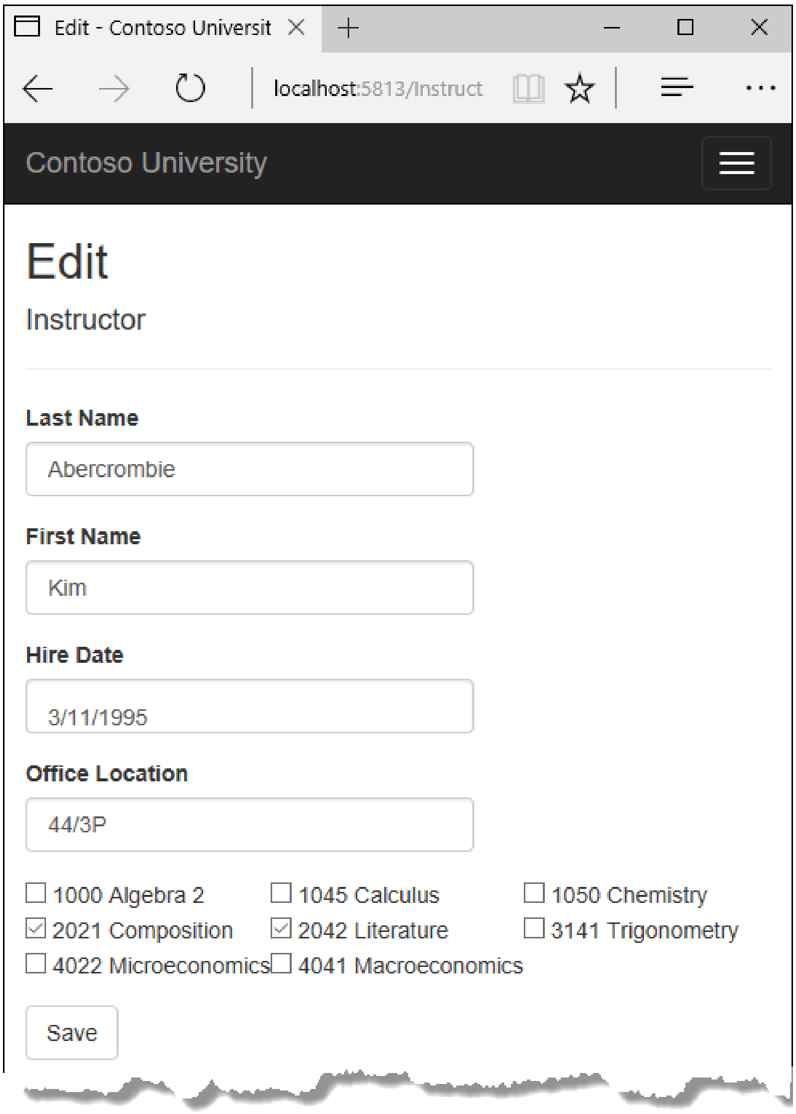
Open the file in a text editor such as Notepad to make this change. If you use Visual Studio, line breaks will be changed in a way that breaks the code. If that happens, fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. After editing the file in a text editor, you can open it in Visual Studio, highlight the block of new code, and press Tab twice to line up the new code with the existing code.

```
<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>
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the Instructor Index page, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

Note

The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[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("Index");
}
```

This code makes the following changes:

Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.

If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to the Create page

In *InstructorController.cs*, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
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("Index");
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In *Views/Instructor/Create.cshtml*, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, this will work better if you do it in a text editor such as Notepad.

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="OfficeAssignment.Location" class="form-control" />
        <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
    </div>
</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>
```

Test by running the **Create** page and adding an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Summary

You have now completed the introduction to working with related data. In the next tutorial you'll see how to handle concurrency conflicts.

PREVIOUS NEXT

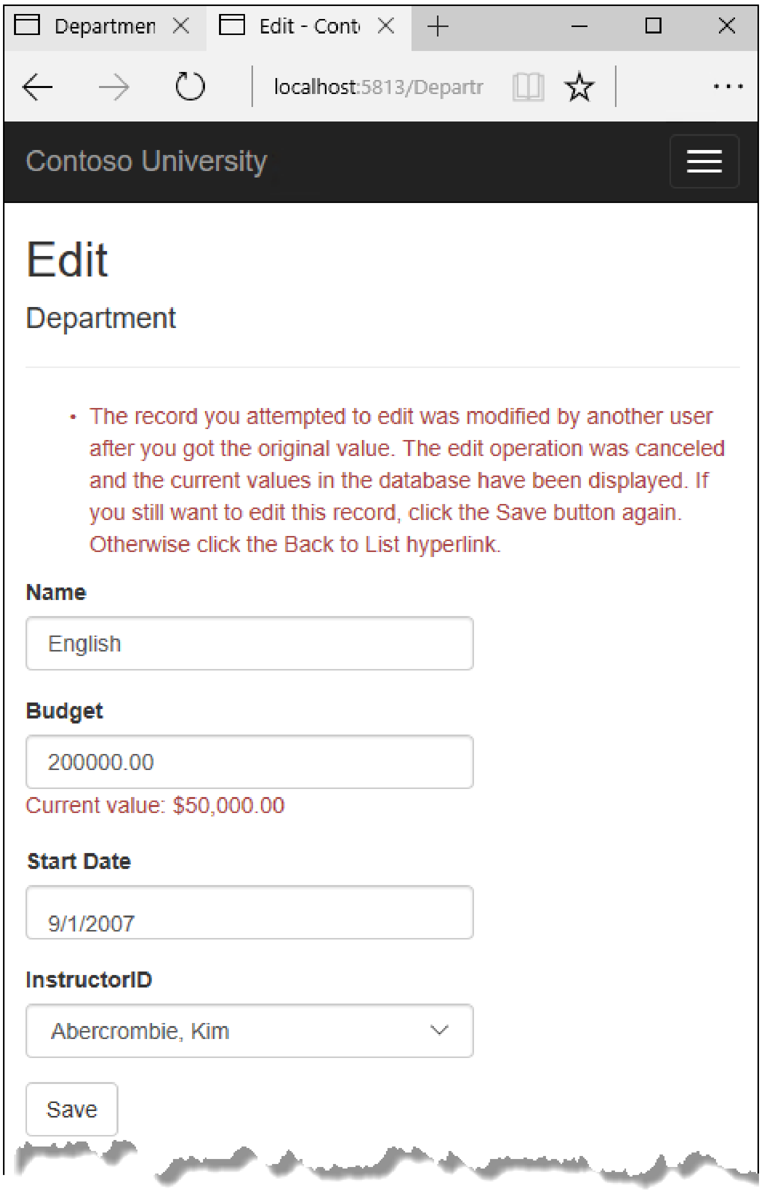
Handling concurrency conflicts - EF Core with ASP.NET Core MVC tutorial (8 of 10)

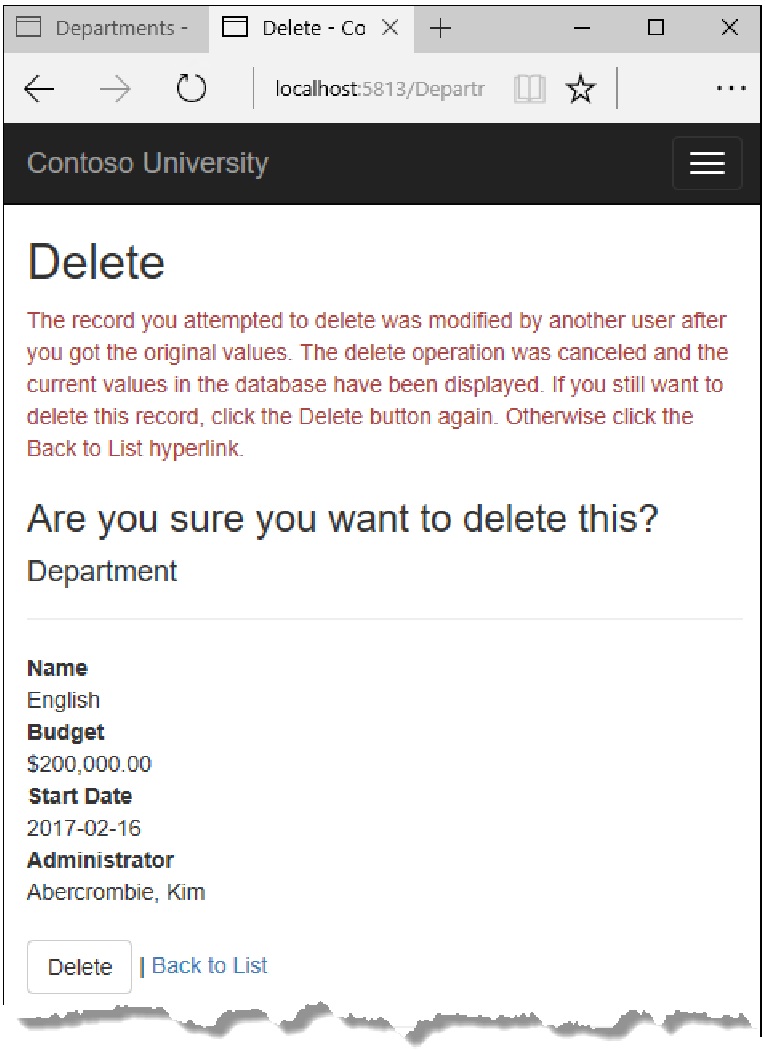
By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In earlier tutorials you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

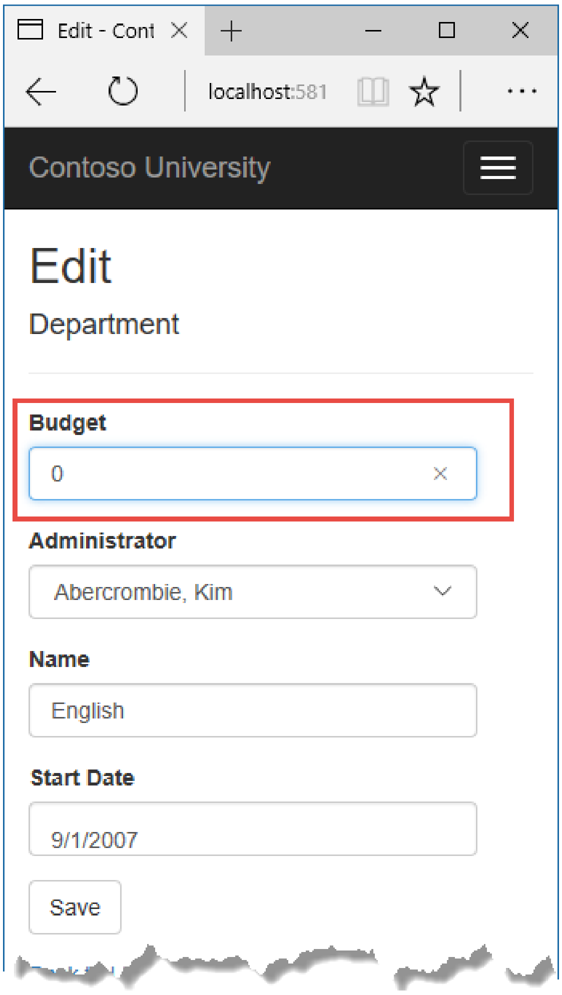
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

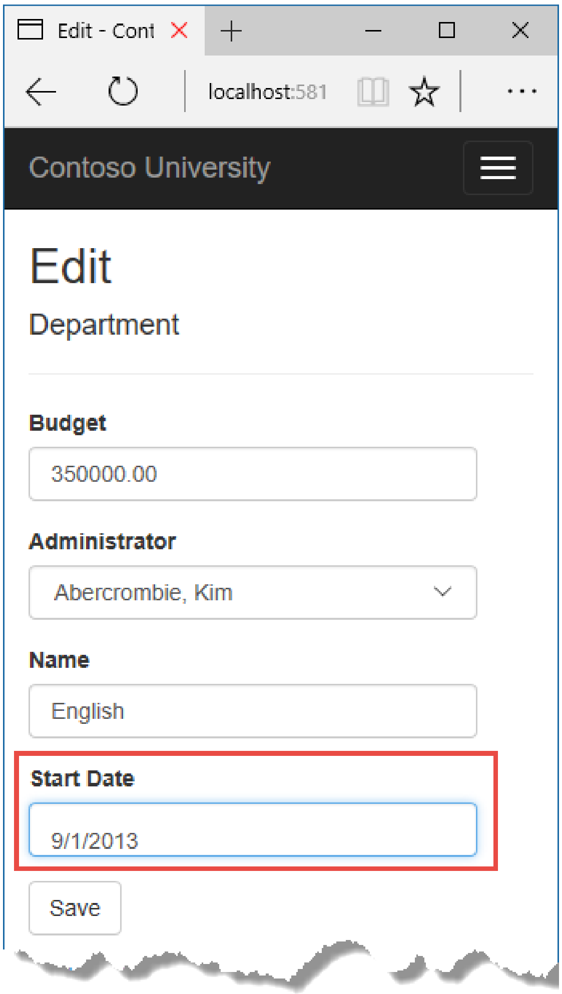
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

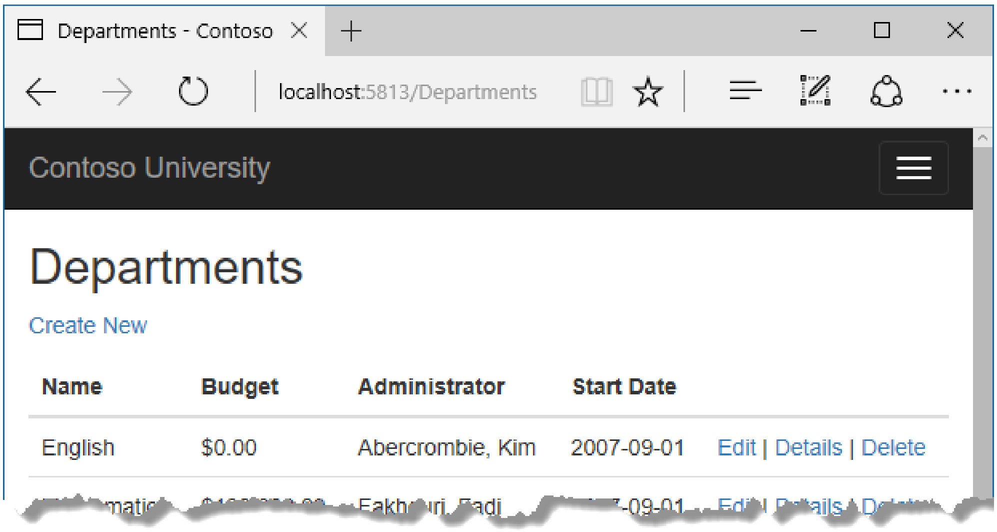
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

You can keep track of which property a user has modified and update only the corresponding columns in the database.

In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

You can let John's change overwrite Jane's change.

The next time someone browses the English department, they'll see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

You can prevent John's change from being updated in the database.

Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property to the Department entity

In *Models/Department.cs*, add a tracking property named RowVersion:

```
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; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsConcurrencyToken` method (in *Data/SchoolContext.cs*) to specify the tracking property, as shown in the following example:

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

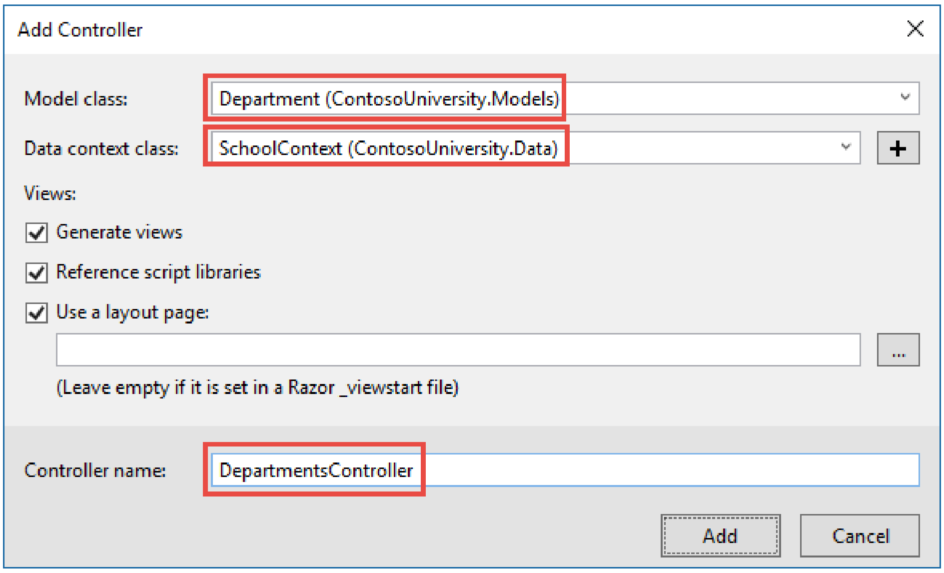
By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

Create a Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the *DepartmentsController.cs* file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

Update the Departments Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in *Views/Departments/Index.cshtml* with the following code.

```
@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>
```

This changes the heading to "Departments" deletes the RowVersion column, and shows full name instead of first name for the administrator.

Update the Edit methods in the Departments controller

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

```
[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("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)
```

```
                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);
}
```

The code begins by trying to read the department to be updated. If the `SingleOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Department Edit view

In *Views/Departments/Edit.cshtml*, make the following changes:

Remove the `<div>` element that was scaffolded for the `RowVersion` field.

Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.

Add a "Select Administrator" option to the drop-down list.

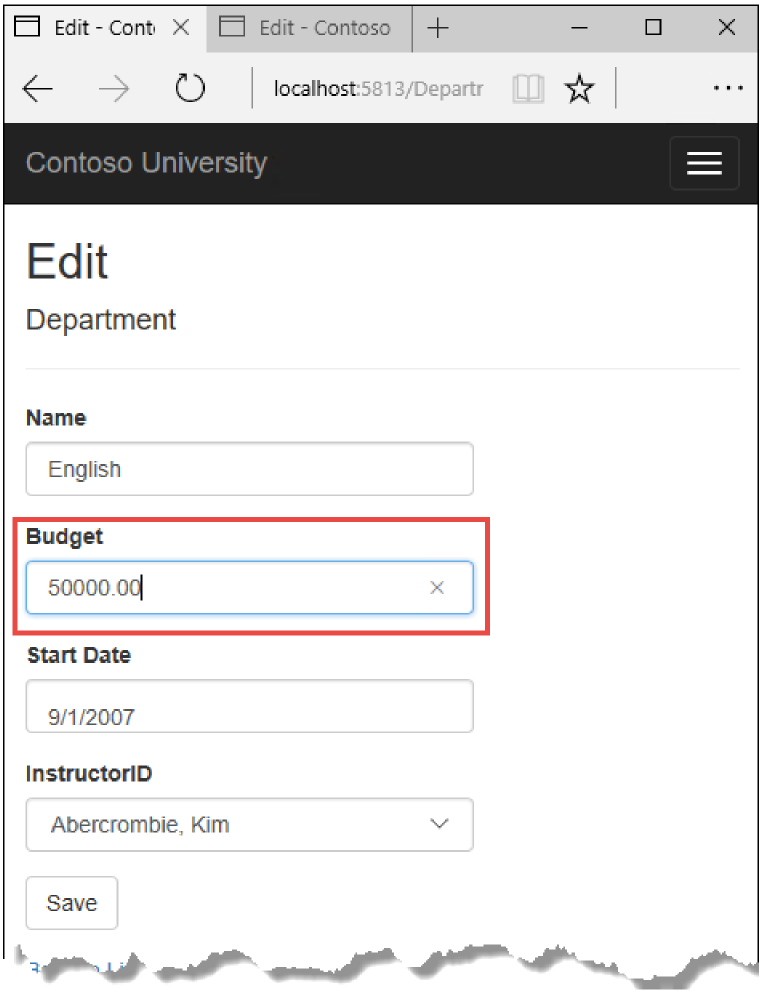
```
@model ContosoUniversity.Models.Department

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

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        <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="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Budget" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="StartDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="InstructorID" class="control-label col-md-2"></label>
            <div class="col-md-10">
                <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>
        <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");}
}
```

Test concurrency conflicts in the Edit page

Run the site and click Departments to go to the Departments Index page.

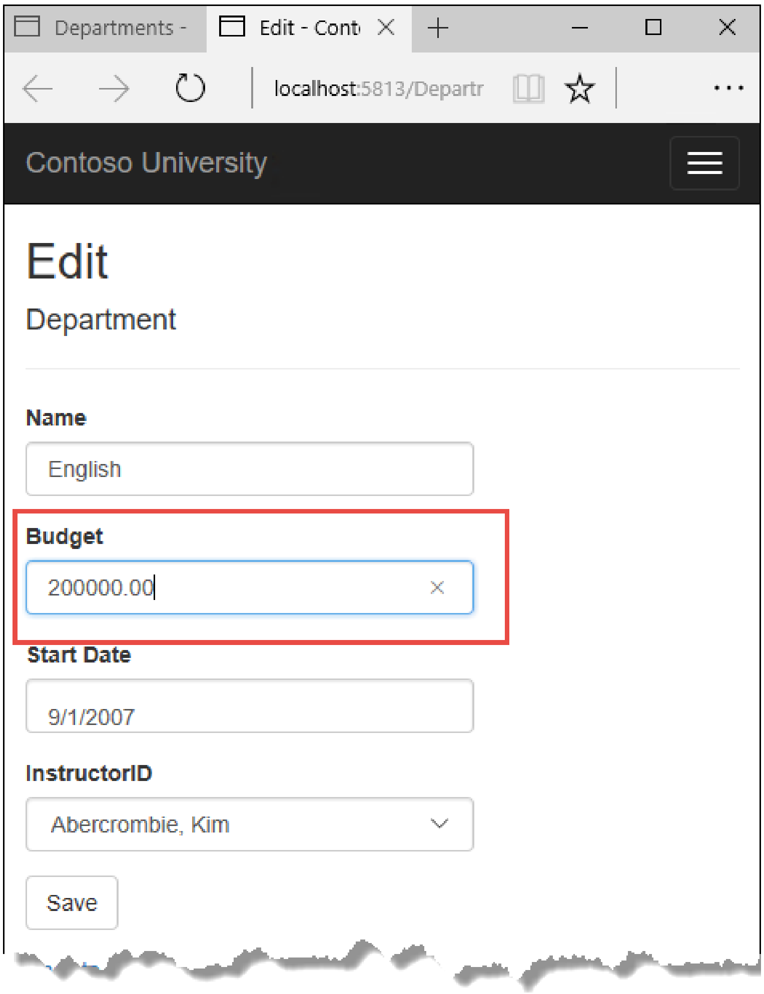
Right click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

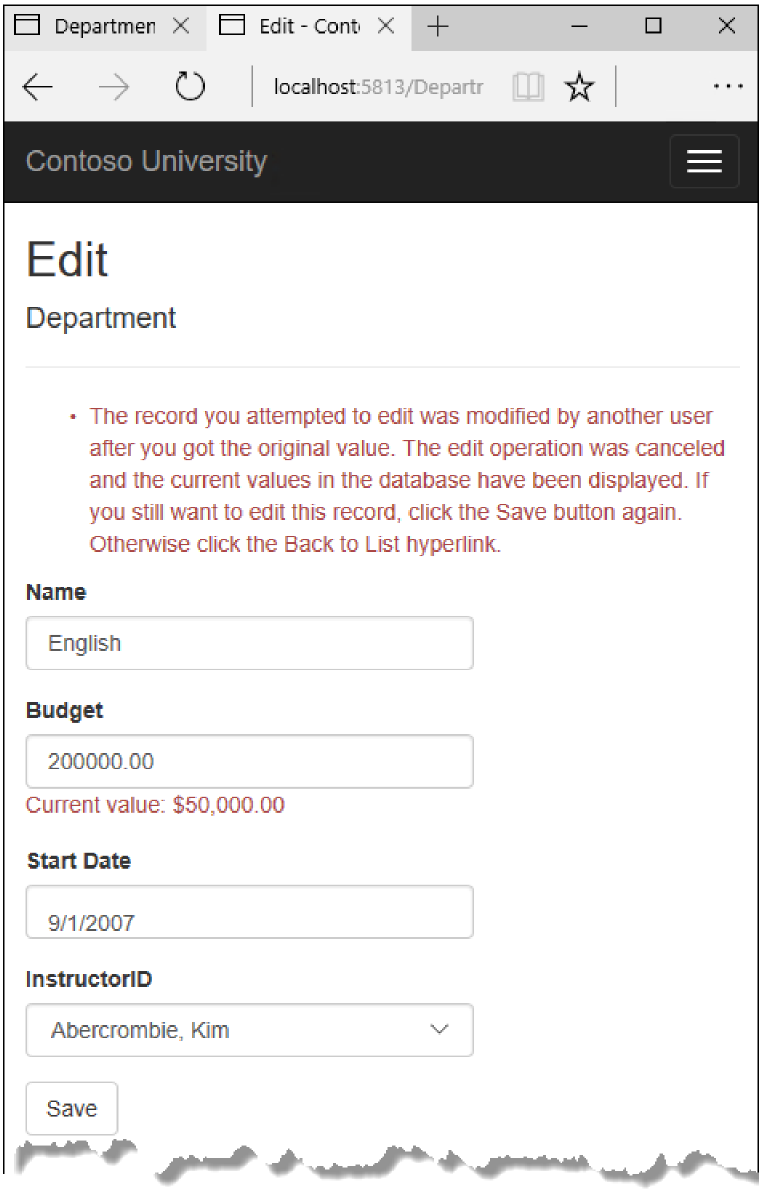


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In *DepartmentController.cs*, replace the HttpGet `Delete` method with the following code:

```
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("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);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

```
[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("Index");
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("Delete", new { concurrencyError = true, id = department.DepartmentID });
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In *Views/Department/Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

```
@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>
```

This makes the following changes:

Adds an error message between the `h2` and `h3` headings.

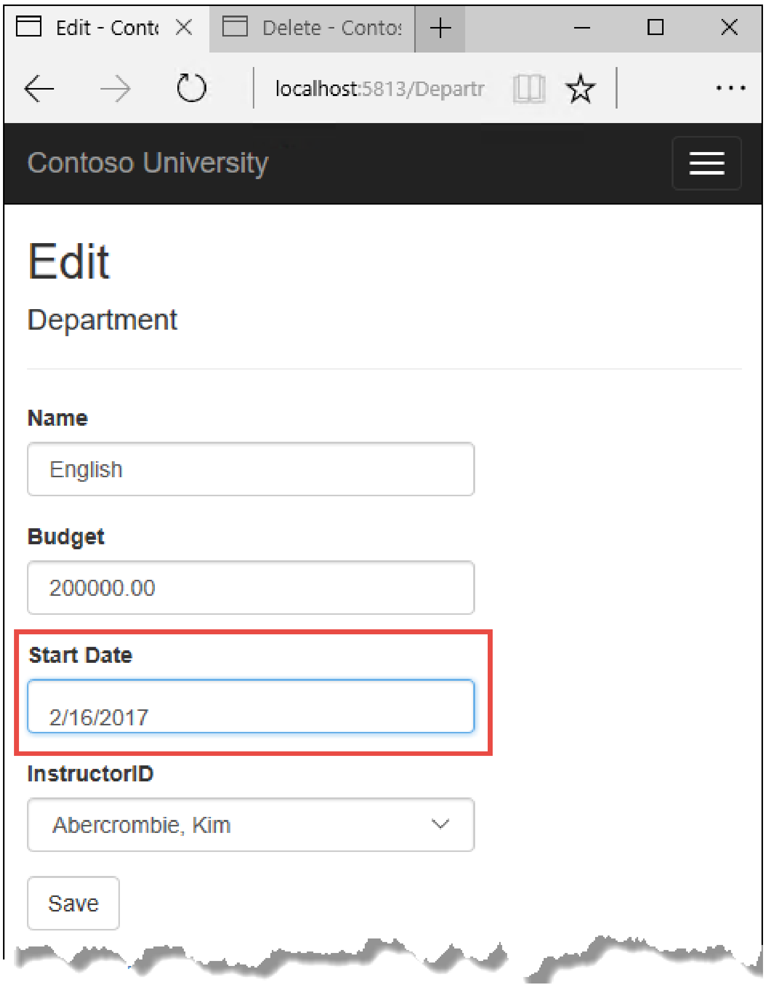
Replaces LastName with FullName in the **Administrator** field.

Removes the RowVersion field.

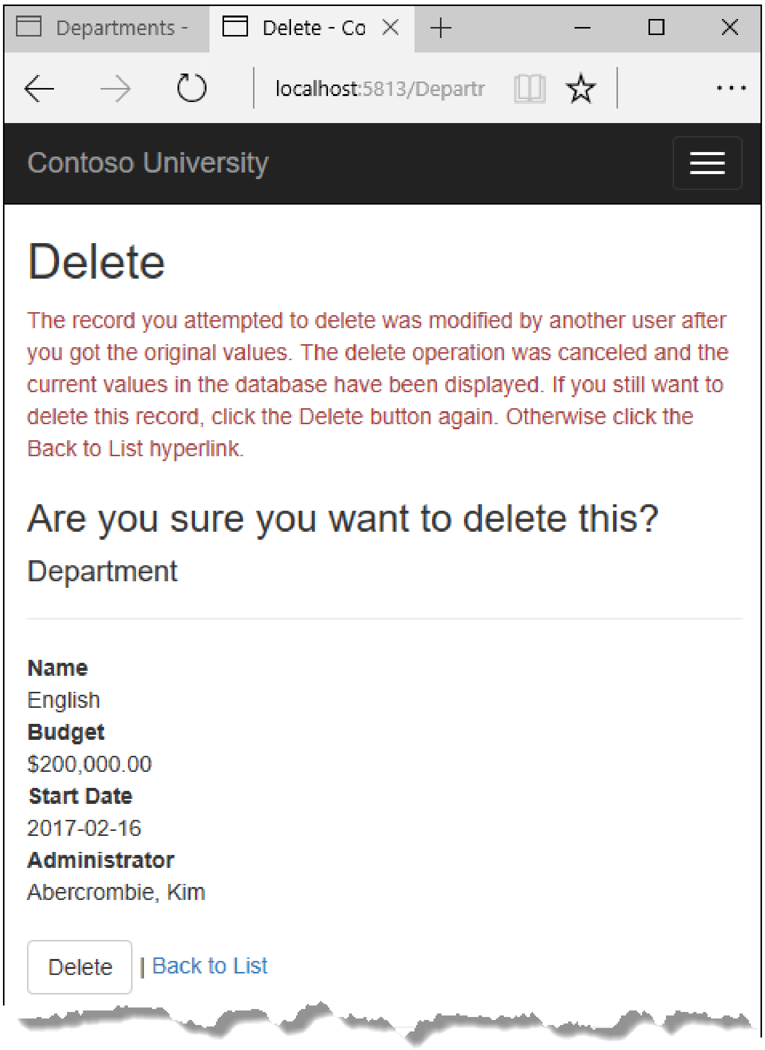
Adds hidden fields for the `DepartmentID` and `RowVersion` properties.

Run the Departments Index page. Right click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in *Views/Departments/Details.cshtml* to delete the RowVersion column and show the full name of the Administrator.

```
@model ContosoUniversity.Models.Department

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

<h2>Details</h2>

<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>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in *Views/Departments/Create.cshtml* to add a Select option to the drop-down list.

```
@model ContosoUniversity.Models.Department

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

<h2>Create</h2>

<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Name" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Budget" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="StartDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="InstructorID" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

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

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

Summary

This completes the introduction to handling concurrency conflicts. For more information about how to handle concurrency in EF Core, see Concurrency conflicts. The next tutorial shows how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Inheritance - EF Core with ASP.NET Core MVC tutorial (9 of 10)

By Tom Dykstra and Rick Anderson

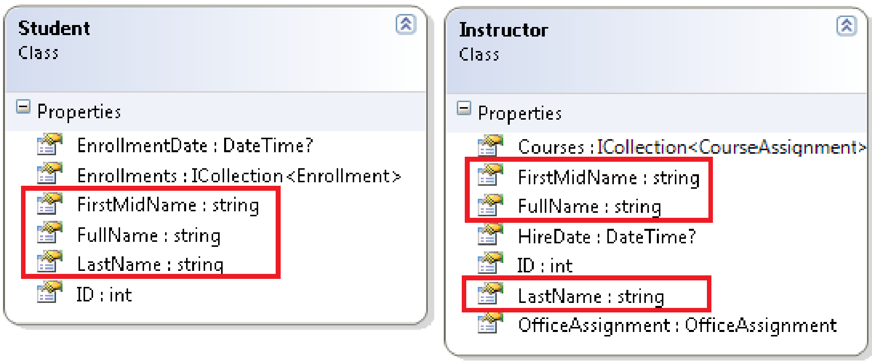
The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

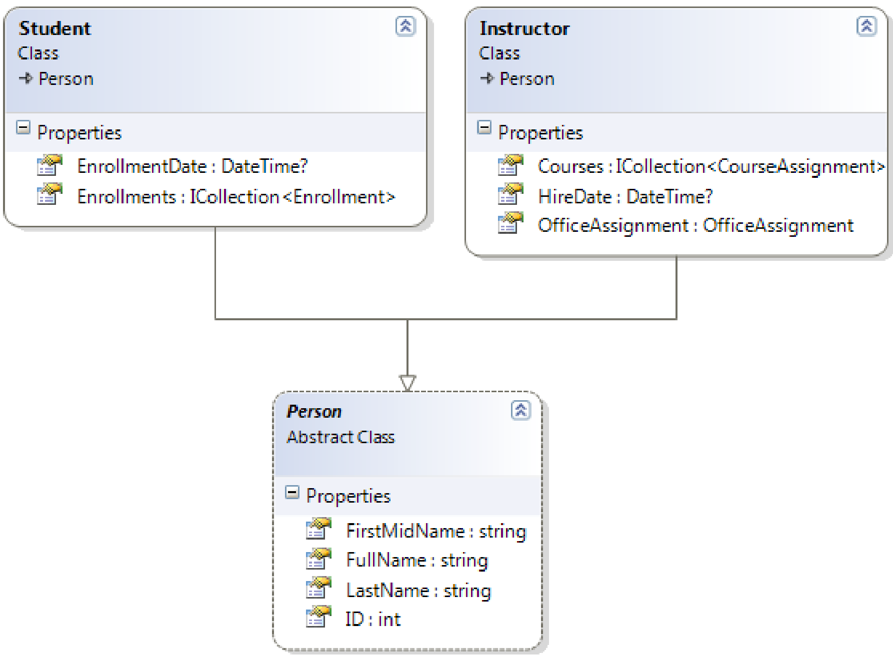
In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

Options for mapping inheritance to database tables

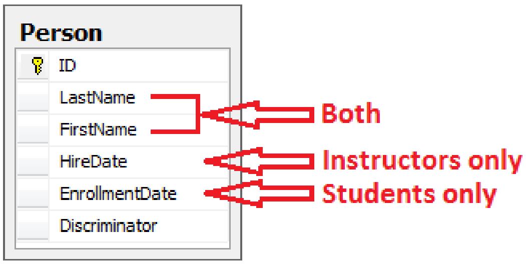
The `Instructor` and `Student` classes in the School data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:

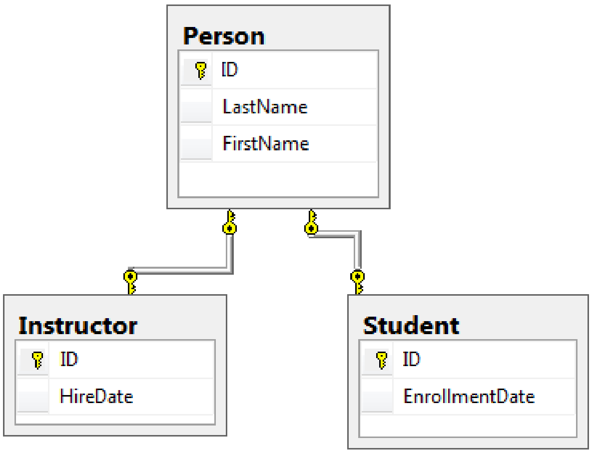


There are several ways this inheritance structure could be represented in the database. You could have a Person table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the Person table and have separate Instructor and Student tables with the date fields.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the Person, Student, and Instructor classes as shown earlier, the Student and Instructor tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

迭 Tip

Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

```
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;
            }
        }
    }
}
```

Make Student and Instructor classes inherit from Person

In *Instructor.cs*, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

```
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; }
    }
}
```

Make the same changes in *Student.cs*.

```
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; }
    }
}
```

Add the Person entity type to the data model

Add the Person entity type to *SchoolContext.cs*. The new lines are highlighted.

```
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 });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and customize migration code

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

```
dotnet ef migrations add Inheritance
```

Run the `database update` command:.

```
dotnet ef database update
```

The command will fail at this point because you have existing data that migrations doesn't know how to handle. You get an error message like the following one:

> The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_CourseAssignment_Person_InstructorID". The conflict occurred in database "ContosoUniversity09133", table "dbo.Person", column 'ID'.

Open *Migrations<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
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);
}
```

This code takes care of the following database update tasks:

Removes foreign key constraints and indexes that point to the Student table.

Renames the Instructor table as Person and makes changes needed for it to store Student data:

Adds nullable EnrollmentDate for students.

Adds Discriminator column to indicate whether a row is for a student or an instructor.

Makes HireDate nullable since student rows won't have hire dates.

Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they'll get new primary key values.

Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.

Fixes foreign key values that point to students.

Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command again:

```
dotnet ef database update
```

(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

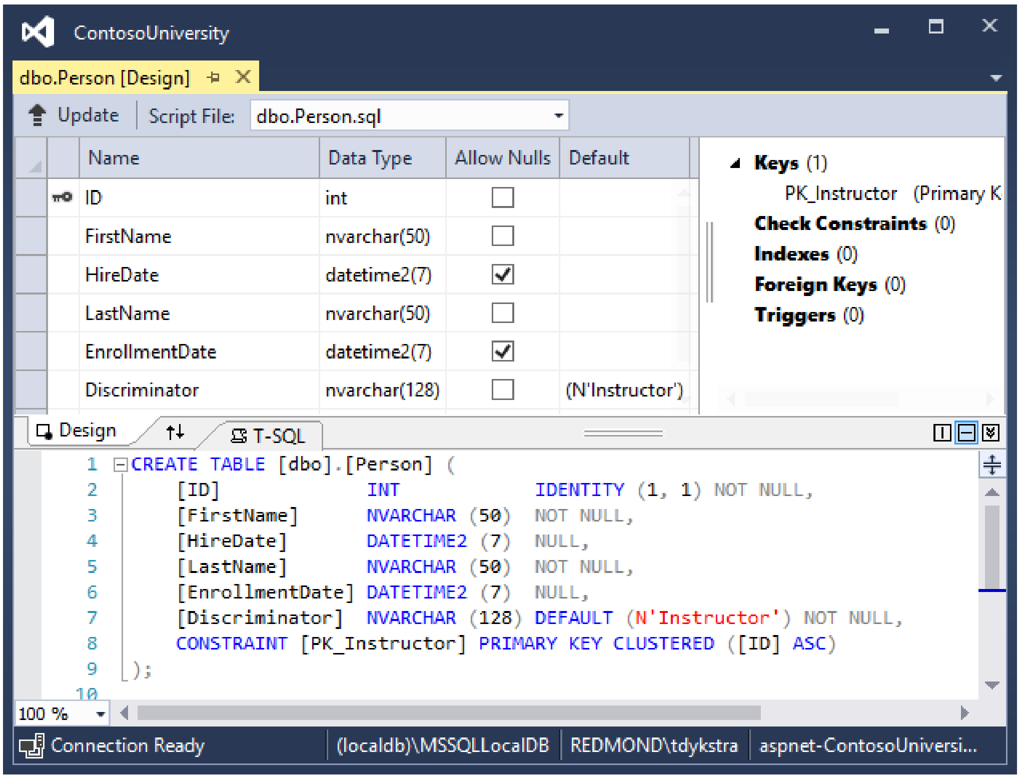
Note

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

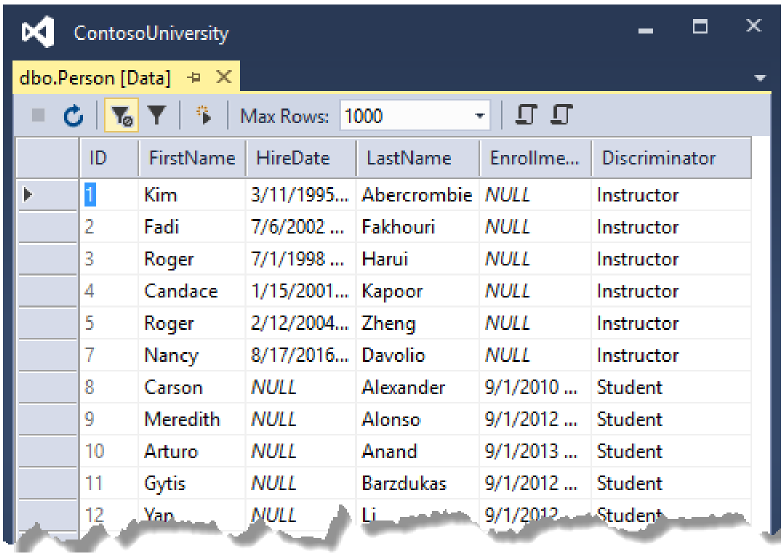
Test with inheritance implemented

Run the site and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Summary

You've implemented table-per-hierarchy inheritance for the `Person`, `Student`, and `Instructor` classes. For more information about inheritance in Entity Framework Core, see Inheritance. In the next tutorial you'll see how to handle a variety of relatively advanced Entity Framework scenarios.

PREVIOUS NEXT

Advanced topics - EF Core with ASP.NET Core MVC tutorial (10 of 10)

By Tom Dykstra and Rick Anderson

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.0 MVC web applications using Entity Framework Core 1.0 and Visual Studio 2015. For information about the tutorial series, see the first tutorial in the series.

In the previous tutorial you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Core.

Raw SQL Queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you turn tracking off.

Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

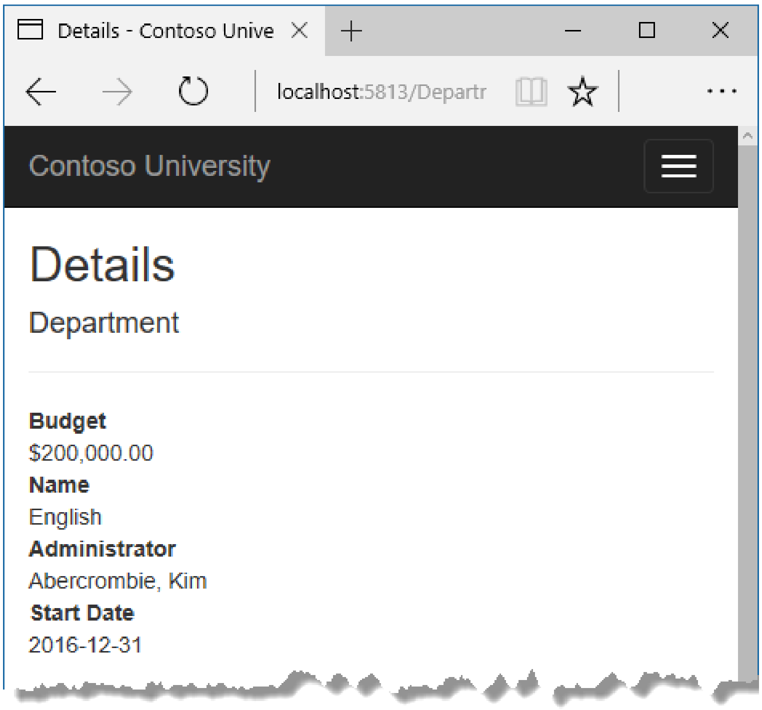
Call a query that returns entities

The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In *DepartmentsController.cs*, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

```
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);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query that returns other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

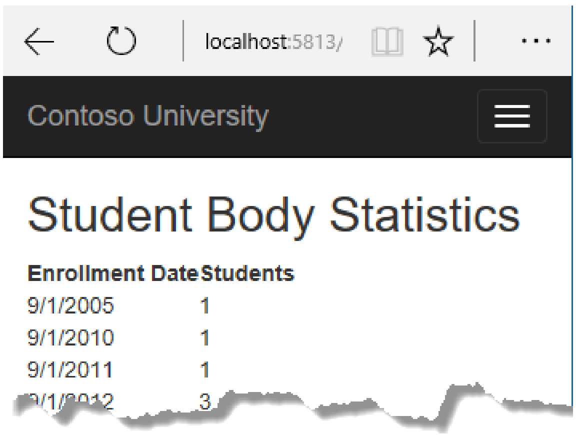
In *HomeController.cs*, replace the `About` method with the following code:

```
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);
}
```

Add a using statement:

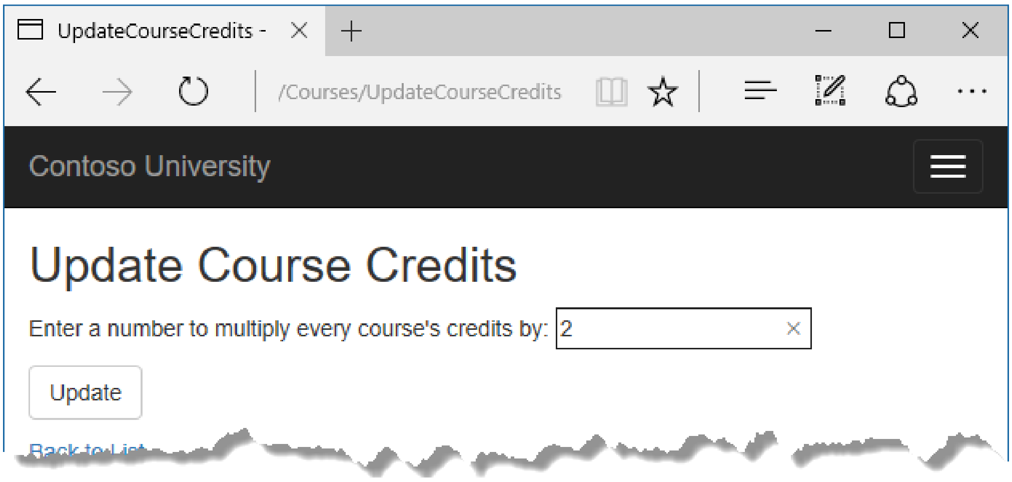
```
using System.Data.Common;
```

Run the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In *CoursesContoller.cs*, add UpdateCourseCredits methods for HttpGet and HttpPost:

```
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();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

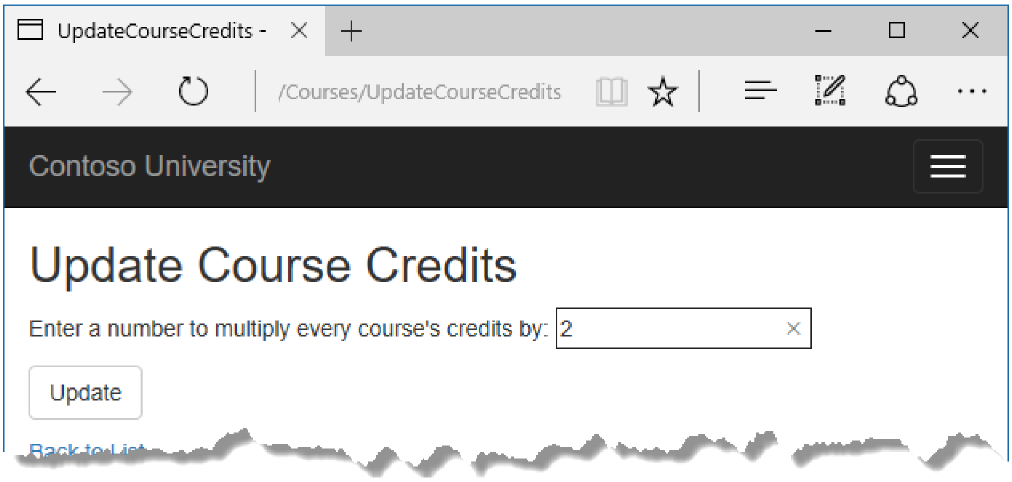
In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add > New Item**.

In the **Add New Item** dialog, click **ASP.NET** under **Installed** in the left pane, click **MVC View Page**, and name the new view *UpdateCourseCredits.cshtml*.

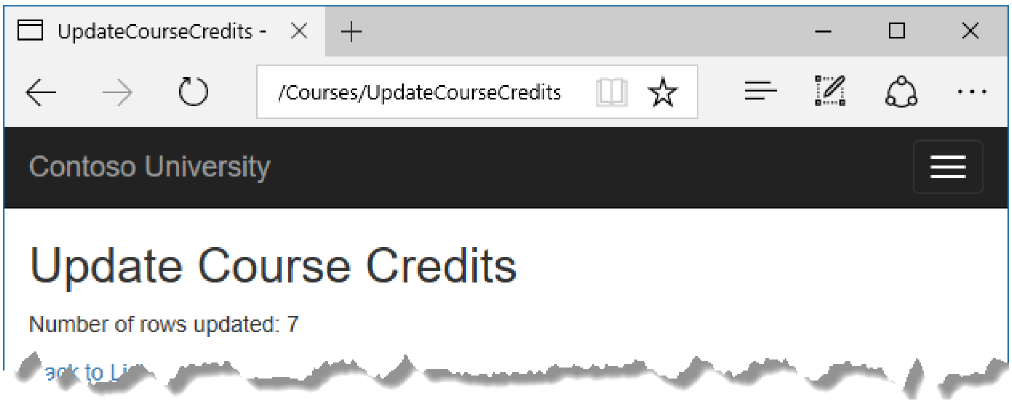
In *Views/Courses/UpdateCourseCredits.cshtml*, replace the template code with the following code:

```
@{
    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>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Course/UpdateCourseCredits)`. Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL sent to the database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open *StudentsController.cs* and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the application in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory:Information: Executed DbCommand
(225ms) [Parameters=[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT [e].[EnrollmentID], [e].[CourseID], [e].[Grade], [e].[StudentID], [c].[CourseID], [c].[Credits], [c].
[DepartmentID], [c].[Title]
FROM [Enrollment] AS [e]
INNER JOIN (
    SELECT DISTINCT TOP(2) [s].[ID]
    FROM [Person] AS [s]
    WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
    ORDER BY [s].[ID]
) AS [s0] ON [e].[StudentID] = [s0].[ID]
INNER JOIN [Course] AS [c] ON [e].[CourseID] = [c].[CourseID]
ORDER BY [s0].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`). The `SingleOrDefaultAsync` method doesn't resolve to one row on the server. If the Where clause matches multiple rows, the method must return null, so EF only has to select a maximum of 2 rows, because if 3 or more match the Where clause, the result from the `SingleOrDefault` method is the same as if 2 rows match.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Repository and unit of work patterns

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

The EF context class itself insulates your code from data-store-specific code.

The EF context class can act as a unit-of-work class for database updates that you do using EF.

EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Testing with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

DbContext.SaveChanges

DbContext.Entry

ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

Entity Framework Core source code and development plans

The source code for Entity Framework Core is available at https://github.com/aspnet/EntityFramework. Besides source code, you can get nightly builds, issue tracking, feature specs, design meeting notes, the roadmap for future development, and more. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify sort selection code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

```
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));
}
```

Next steps

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET MVC application.

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework

Core in Action.

For information about how to deploy your web application after you've built it, see Publishing and deployment.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see the ASP.NET Core documentation.

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials.

Common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in *appsettings.json*. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found

> or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

PREVIOUS

Creating Backend Services for Native Mobile Applications

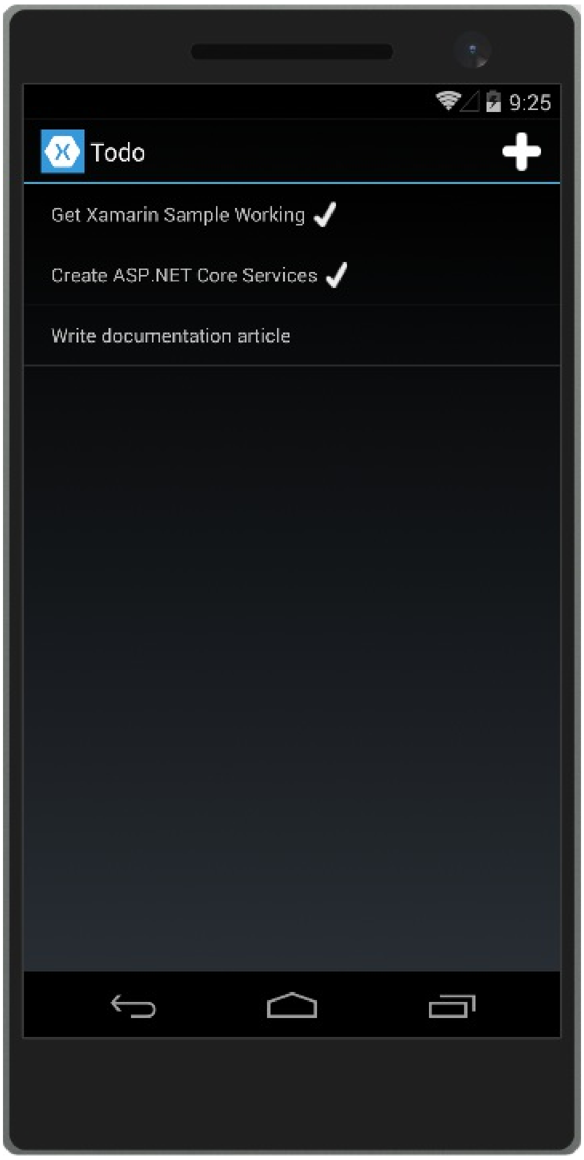
By Steve Smith

Mobile apps can easily communicate with ASP.NET Core backend services.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the Xamarin Forms ToDoRest app as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

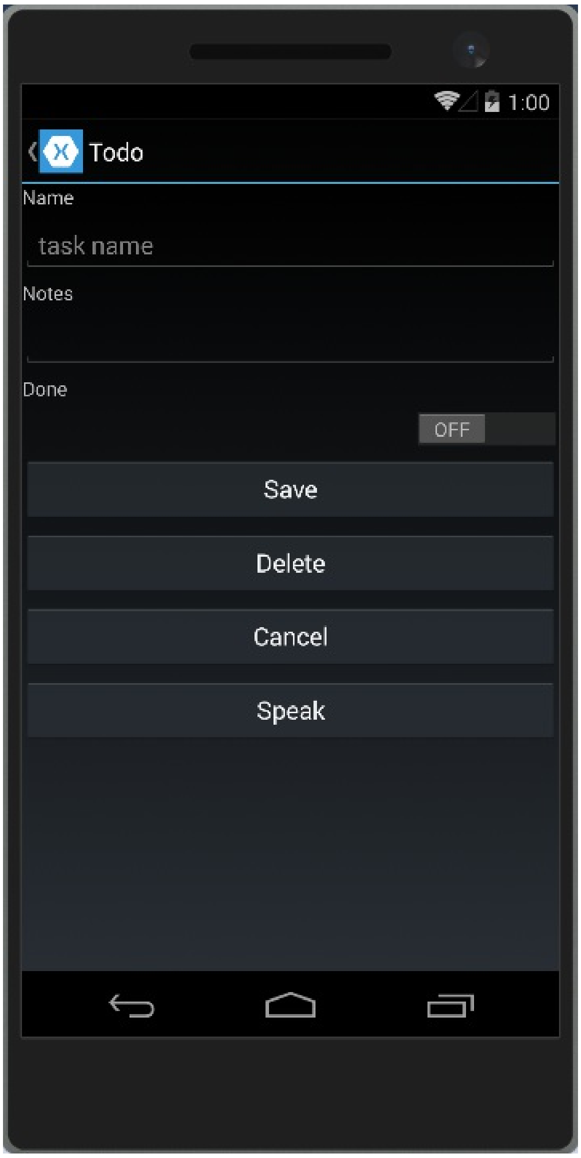


Features

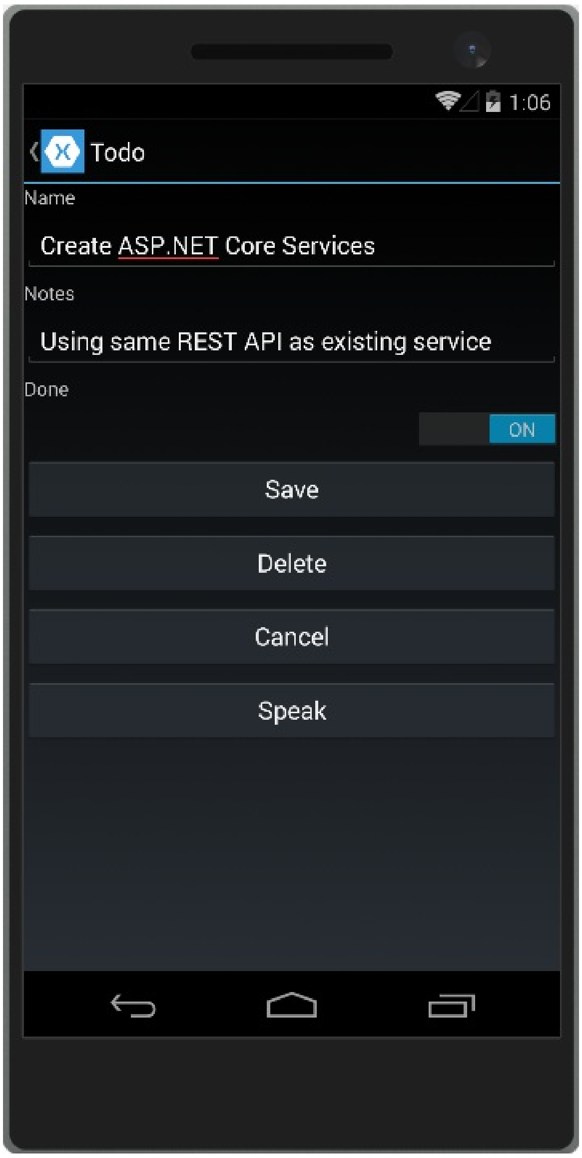
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the `+` icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:

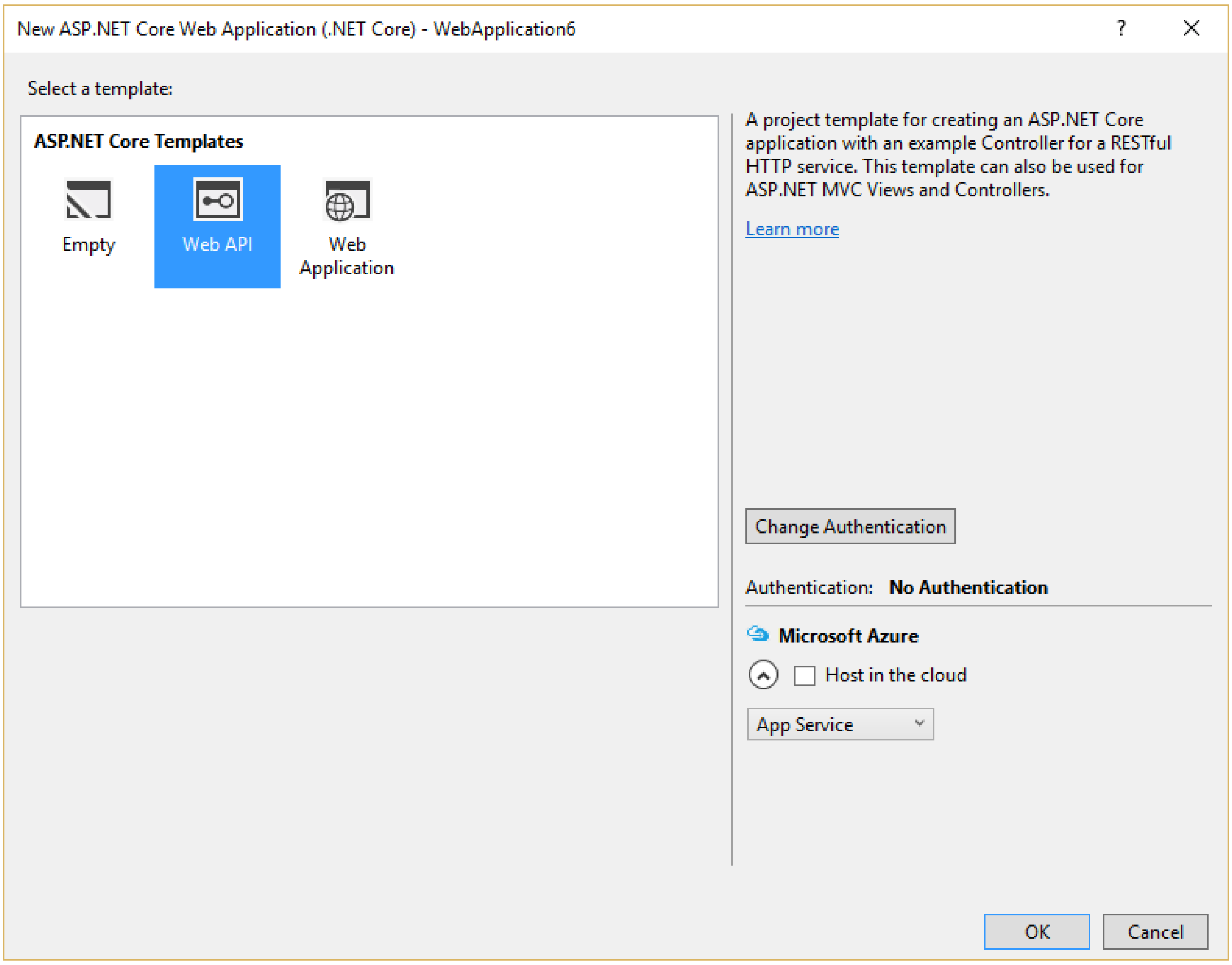


This sample is configured by default to use backend services hosted at developer.xamarin.com, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the *Constants.cs* file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// 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}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include `.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Note

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```
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; }
    }
}
```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```
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);
    }
}
```

For this sample, the implementation just uses a private collection of items:

```
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);
        }
    }
}
```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository,ToDoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

> 汖 Tip
>
> Learn more about creating web APIs in Building Your First Web API with ASP.NET Core MVC and Visual Studio.

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from Microsoft.AspNetCore.Mvc.Controller. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `IToDoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for dependency injection.

```
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;
        }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

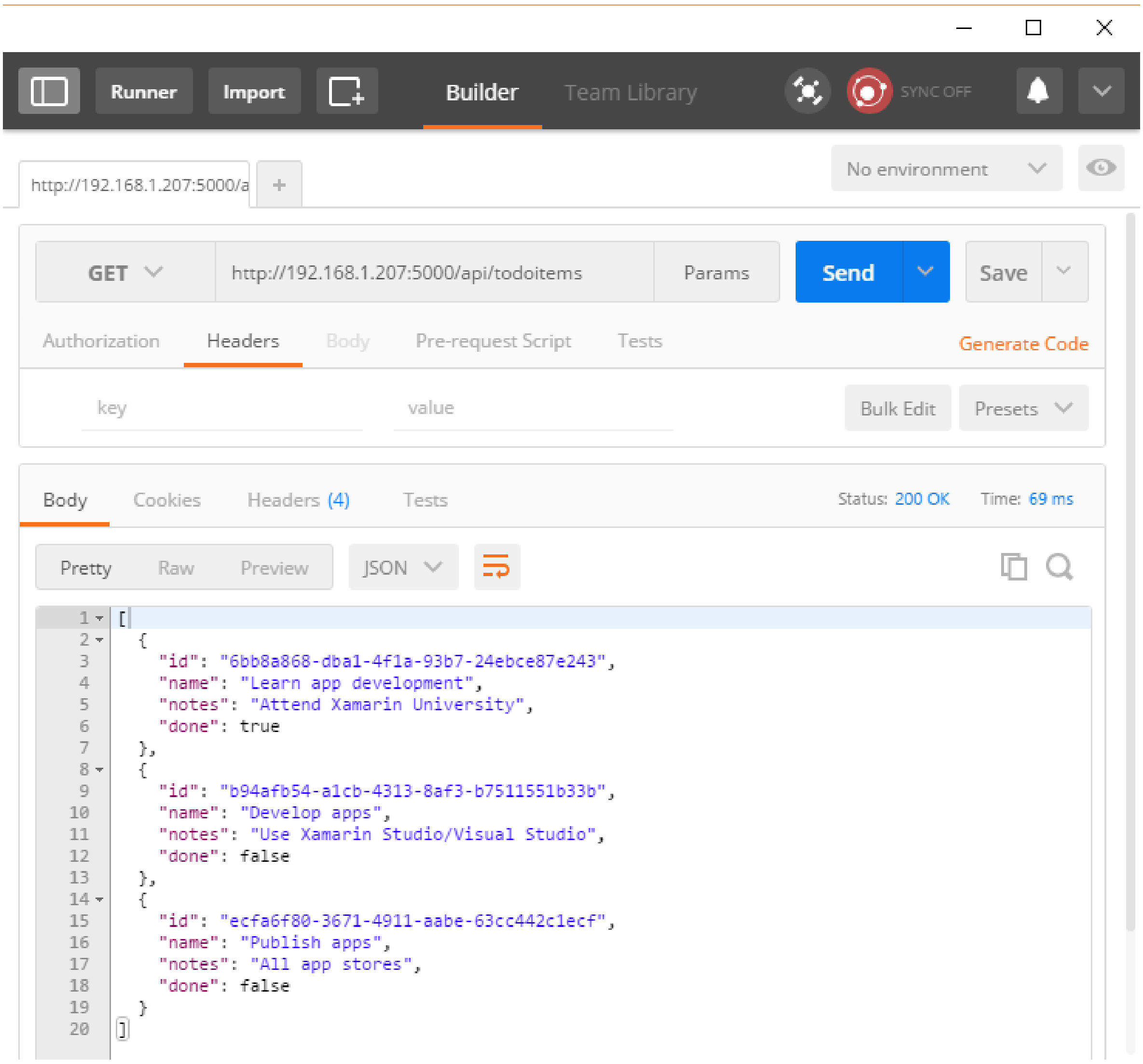
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_toDoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the ToDo items, serialized as JSON.

You can test your new API method using a variety of tools, such as Postman, shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts an ID parameter and a `ToDoItem` instance. The HTTP verb attributes, like `[HttpPost]`, optionally accept a route template string (`{id}` in this example). This has the same effect as adding a `[Route]` attribute to the action. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

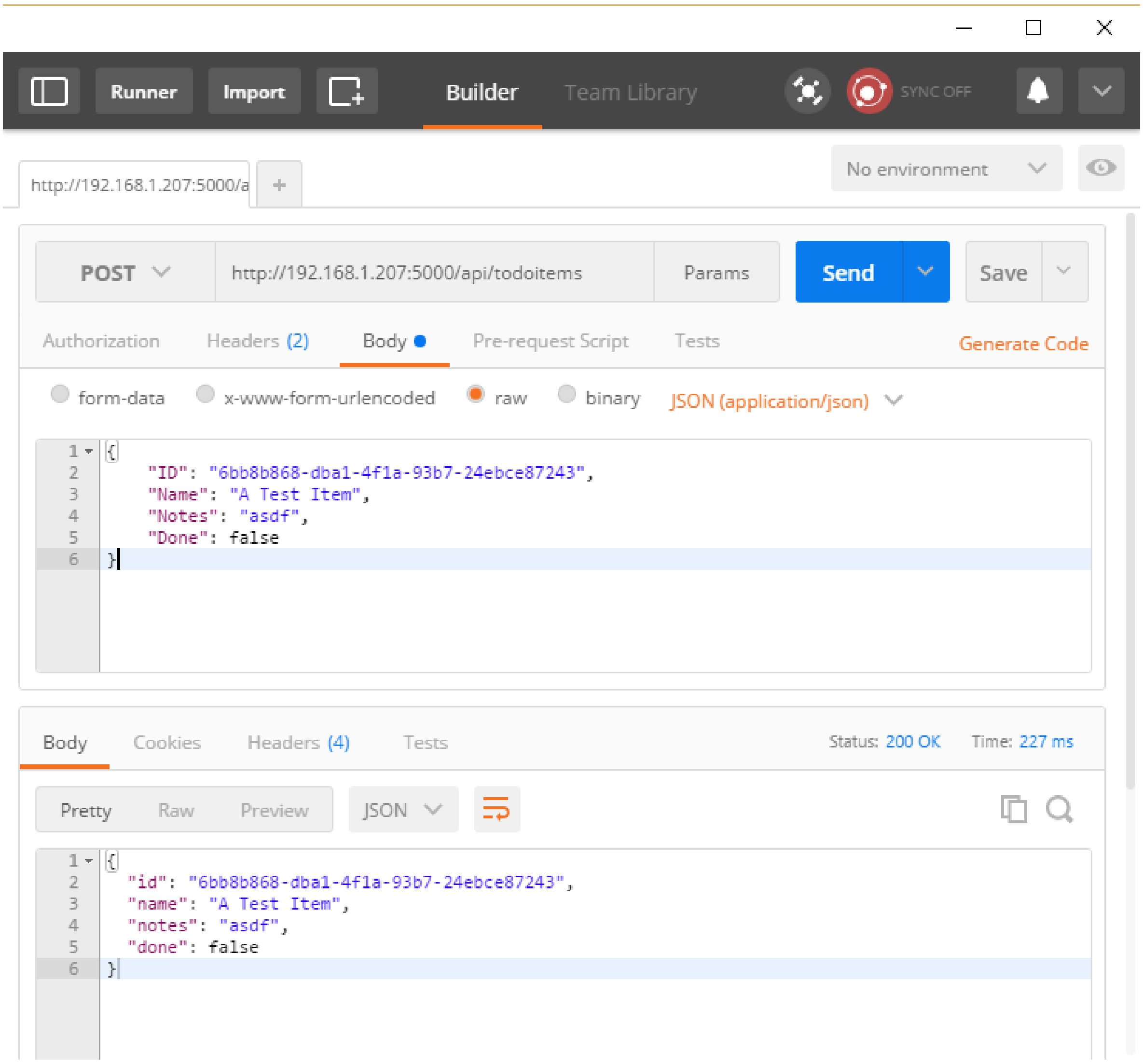
Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

```
[HttpPost("{id}")]
public IActionResult Create(string id, [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);
}
```

The sample uses an enum containing error codes that are passed to the mobile client:

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.



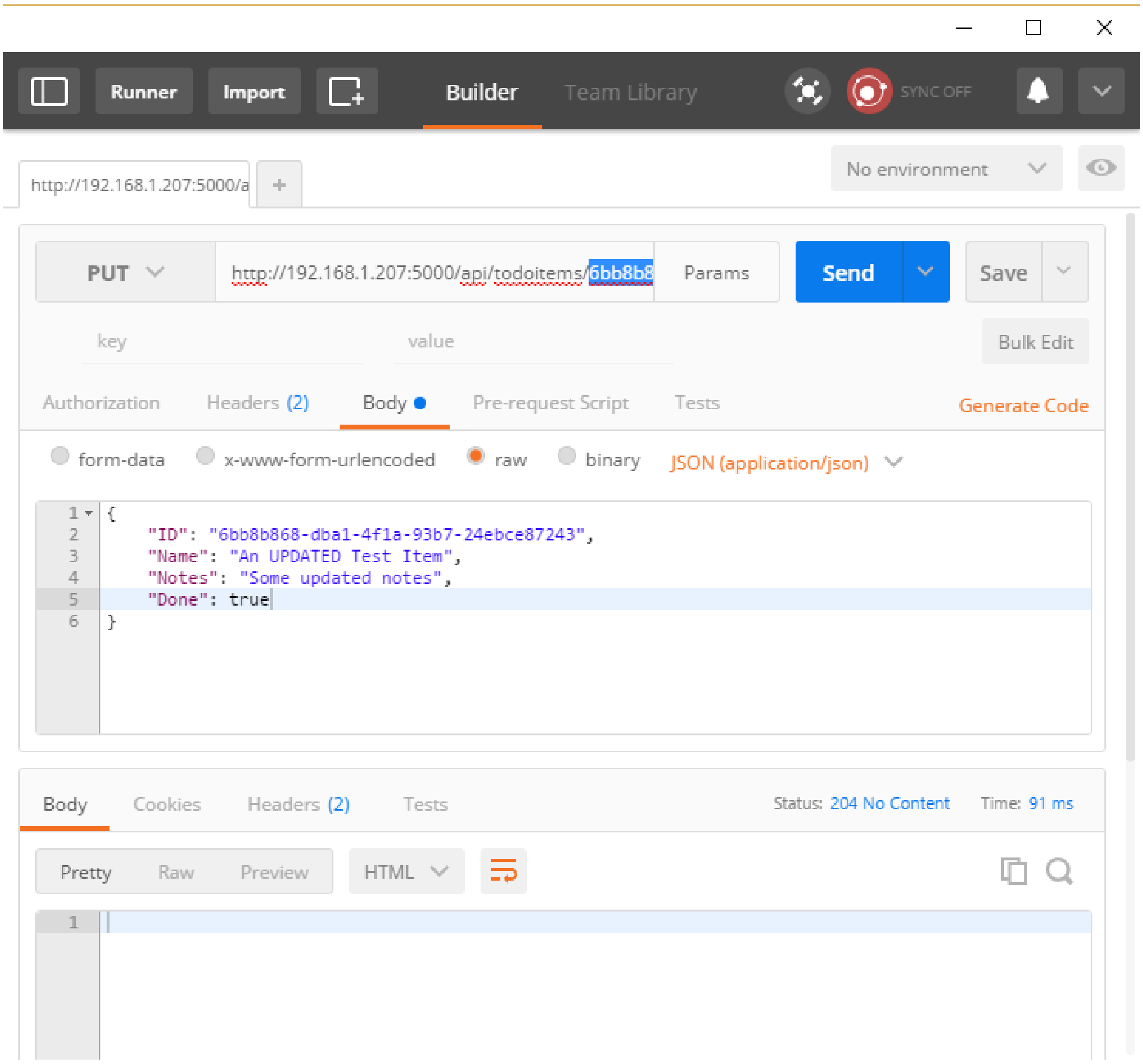
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut("{id}")]
public IActionResult Edit(string id, [FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _toDoRepository.Find(id);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT and add the ID of the record being updated to the URL. Specify the updated object data in the Body of the request.



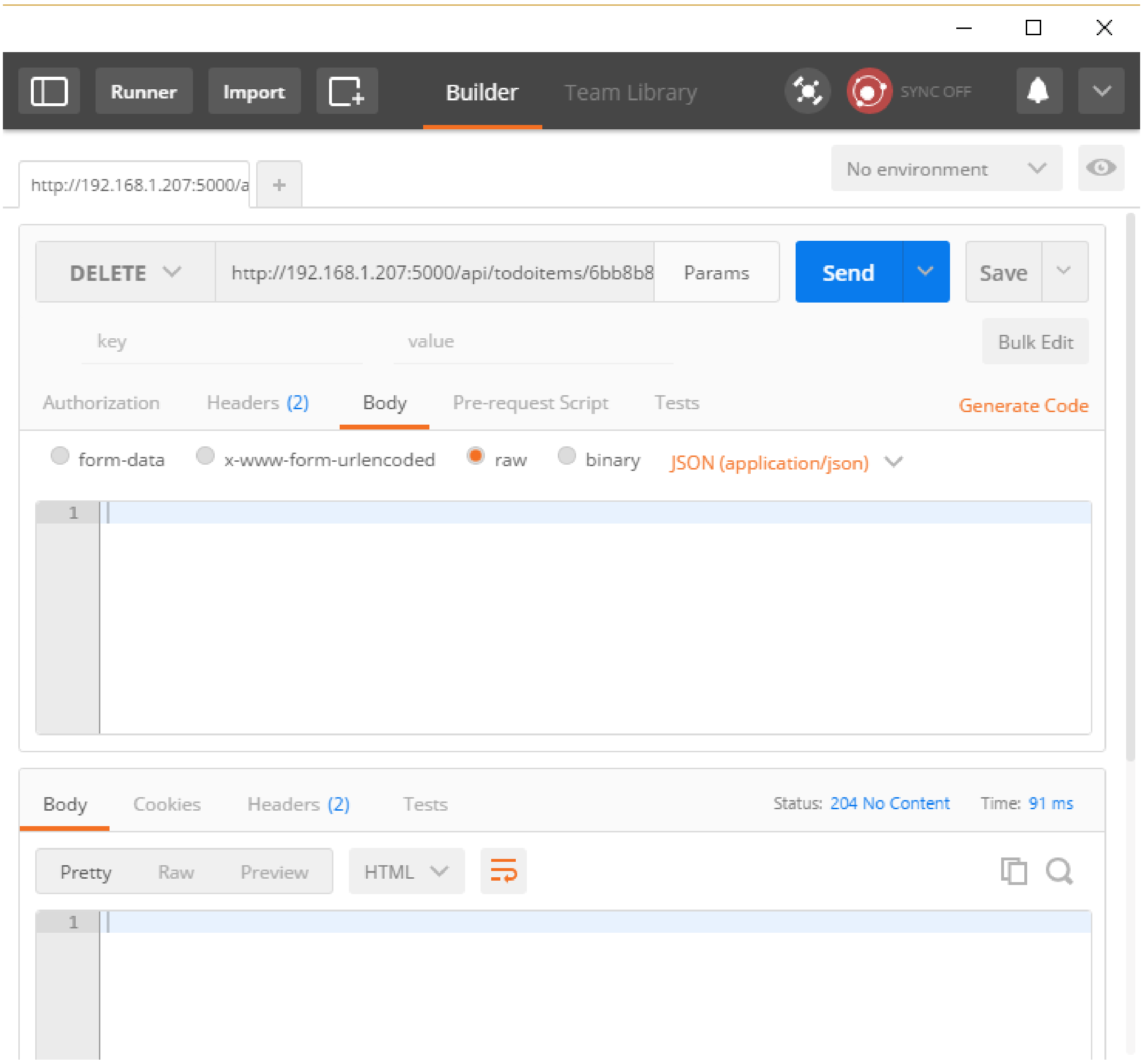
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[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();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Handling requests with controllers in ASP.NET MVC Core

By Steve Smith

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET MVC Core.

What is a Controller

In ASP.NET MVC, a *Controller* is used to define and group a set of actions. An *action* (or *action method*) is a method on a controller that handles incoming requests. Controllers provide a logical means of grouping similar actions together, allowing common sets of rules (e.g. routing, caching, authorization) to be applied collectively. Incoming requests are mapped to actions through routing.

In ASP.NET Core MVC, a controller can be any instantiable class that ends in "Controller" or inherits from a class that ends with "Controller". Controllers should follow the Explicit Dependencies Principle and request any dependencies their actions require through their constructor using dependency injection.

By convention, controller classes:

Are located in the root-level "Controllers" folder

Inherit from Microsoft.AspNetCore.Mvc.Controller

These two conventions are not required.

Within the Model-View-Controller pattern, a Controller is responsible for the initial processing of the request and instantiation of the Model. Generally, business decisions should be performed within the Model.

 Note

The Model should be a *Plain Old CLR Object (POCO)*, not a `DbContext` or database-related type.

The controller takes the result of the model's processing (if any), returns the proper view along with the associated view data. Learn more: Overview of ASP.NET Core MVC and Getting started with ASP.NET Core MVC and Visual Studio.

 Tip

The Controller is a *UI level* abstraction. Its responsibility is to ensure incoming request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps it will not directly include data access or business logic, but instead will delegate to services handling these responsibilities.

Defining Actions

Any public method on a controller type is an action. Parameters on actions are bound to request data and validated using model binding.

 Warning

Action methods that accept parameters should verify the `ModelState.IsValid` property is true.

Action methods should contain logic for mapping an incoming request to a business concern. Business concerns should typically be represented as services that your controller accesses through dependency injection. Actions then map the result of the business action to an application state. Actions can return anything, but frequently will return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*; the action result *does the responding*.

Controller Helper Methods

Although not required, most developers will want to have their controllers inherit from the base `Controller` class. Doing so provides controllers with access to many properties and helpful methods, including the following helper methods designed to assist in returning various responses:

View

Returns a view that uses a model to render HTML. Example: `return View(customer);`

HTTP Status Code

Return an HTTP status code. Example: `return BadRequest();`

Formatted Response

Return `Json` or similar to format an object in a specific manner. Example: `return Json(customer);`

Content negotiated response

Instead of returning an object directly, an action can return a content negotiated response (using `Ok`, `Created`, `CreatedAtRoute` or `CreatedAtAction`). Examples: `return Ok();` or `return CreatedAtRoute("routename",values,newobject);`

Redirect

Returns a redirect to another action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction` or `RedirectToRoute`). Example: `return RedirectToAction("Complete", new {id = 123});`

In addition to the methods above, an action can also simply return an object. In this case, the object will be formatted based on the client's request. Learn more about Formatting Response Data

Cross-Cutting Concerns

In most apps, many actions will share parts of their workflow. For instance, most of an app might be available only to authenticated users, or might benefit from caching. When you want to perform some logic before or after an action method runs, you can use a *filter*. You can help keep your actions from growing too large by using Filters to handle these cross-cutting concerns. This can help eliminate duplication within your actions, allowing them to follow the Don't Repeat Yourself (DRY) principle.

In the case of authorization and authentication, you can apply the `Authorize` attribute to any actions that require it. Adding it to a controller will apply it to all actions within that controller. Adding this attribute will ensure the appropriate filter is applied to any request for this action. Some attributes can be applied at both controller and action levels to provide granular control over filter behavior. Learn more: Filters.

Other examples of cross-cutting concerns in MVC apps may include:

Error handling

Response Caching

Note

Many cross-cutting concerns can be handled using filters in MVC apps. Another option to keep in mind that is available to any ASP.NET Core app is custom middleware.

Routing to Controller Actions

By Ryan Nowak and Rick Anderson

ASP.NET Core MVC uses the Routing middleware to match the URLs of incoming requests and map them to actions. Routes are defined in startup code or attributes. Routes describe how URL paths should be matched to actions. Routes are also used to generate URLs (for links) sent out in responses.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. See Mixed routing for more information.

This document will explain the interactions between MVC and routing, and how typical MVC apps make use of routing features. See Routing for details on advanced routing.

Setting up Routing Middleware

In your *Configure* method you may see code similar to:

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Inside the call to `UseMvc`, `MapRoute` is used to create a single route, which we'll refer to as the `default` route. Most MVC apps will use a route with a template similar to the `default` route.

The route template `"{controller=Home}/{action=Index}/{id?}"` can match a URL path like `/Products/Details/5` and will extract the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. MVC will attempt to locate a controller named `ProductsController` and run the action `Details`:

```
public class ProductsController : Controller
{
   public IActionResult Details(int id) { ... }
}
```

Note that in this example, model binding would use the value of `id = 5` to set the `id` parameter to `5` when invoking this action. See the Model Binding for more details.

Using the `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

The route template:

`{controller=Home}` defines `Home` as the default `controller`

`{action=Index}` defines `Index` as the default `action`

`{id?}` defines `id` as optional

Default and optional route parameters do not need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.

`"{controller=Home}/{action=Index}/{id?}"` can match the URL path `/` and will produce the route values `{ controller = Home, action = Index }`. The values for `controller` and `action` make use of the default values, `id` does not produce a value since there is no corresponding segment in the URL path. MVC would use these route values to select the `HomeController` and `Index` action:

```
public class HomeController : Controller
{
  public IActionResult Index() { ... }
}
```

Using this controller definition and route template, the `HomeController.Index` action would be executed for any of the following URL paths:

`/Home/Index/17`

`/Home/Index`

`/Home`

`/`

The convenience method `UseMvcWithDefaultRoute`:

```
app.UseMvcWithDefaultRoute();
```

Can be used to replace:

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

`UseMvc` and `UseMvcWithDefaultRoute` add an instance of `RouterMiddleware` to the middleware pipeline. MVC doesn't interact directly with middleware, and uses routing to handle requests. MVC is connected to the routes through an instance of `MvcRouteHandler`. The code inside of `UseMvc` is similar to the following:

```
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` does not directly define any routes, it adds a placeholder to the route collection for the `attribute` route. The overload `UseMvc(Action<IRouteBuilder>)` lets you add your own routes and also supports attribute routing. `UseMvc` and all of its variations adds a placeholder for the attribute route - attribute routing is always available regardless of how you configure `UseMvc`. `UseMvcWithDefaultRoute` defines a default route and supports attribute routing. The Attribute Routing section includes more details on attribute routing.

Conventional routing

The `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

is an example of a *conventional routing*. We call this style *conventional routing* because it establishes a *convention* for URL paths:

the first path segment maps to the controller name

the second maps to the action name.

the third segment is used for an optional `id` used to map to a model entity

Using this `default` route, the URL path `/Products/List` maps to the `ProductsController.List` action, and `/Blog/Article/17` maps to `BlogController.Article`. This mapping is based on the controller and action names **only** and is not based on namespaces, source file locations, or method parameters.

 Tip

Using conventional routing with the default route allows you to build the application quickly without having to come up with a new URL pattern for each action you define. For an application with CRUD style actions, having consistency for the URLs across your controllers can help simplify your code and make your UI more predictable.

 Warning

The `id` is defined as optional by the route template, meaning that your actions can execute without the ID provided as part of the URL. Usually what will happen if `id` is omitted from the URL is that it will be set to `0` by model binding, and as a result no entity will be found in the database matching `id == 0`. Attribute routing can give you fine-grained control to make the ID required for some actions and not for others. By convention the documentation will include optional parameters like `id` when they are likely to appear in correct usage.

Multiple routes

You can add multiple routes inside `UseMvc` by adding more calls to `MapRoute`. Doing so allows you to define multiple conventions, or to add conventional routes that are dedicated to a specific action, such as:

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
            defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
}
```

The `blog` route here is a *dedicated conventional route*, meaning that it uses the conventional routing system, but is dedicated to a specific action. Since `controller` and `action` don't appear in the route template as parameters, they can only have the default values, and thus this route will always map to the action `BlogController.Article`.

Routes in the route collection are ordered, and will be processed in the order they are added. So in this example, the `blog` route will be tried before the `default` route.

 Note

Dedicated conventional routes often use catch-all route parameters like `{*article}` to capture the remaining portion of the URL path. This can make a route 'too greedy' meaning that it matches URLs that you intended to be matched by other routes. Put the 'greedy' routes later in the route table to solve this.

Fallback

As part of request processing, MVC will verify that the route values can be used to find a controller and action in your application. If the route values don't match an action then the route is not considered a match, and the next route will be tried. This is called *fallback*, and it's intended to simplify cases where conventional routes overlap.

Disambiguating actions

When two actions match through routing, MVC must disambiguate to choose the 'best' candidate or else throw an exception. For example:

```
public class ProductsController : Controller
{
   public IActionResult Edit(int id) { ... }

   [HttpPost]
   public IActionResult Edit(int id, Product product) { ... }
}
```

This controller defines two actions that would match the URL path `/Products/Edit/17` and route data `{ controller = Products, action = Edit, id = 17 }`. This is a typical pattern for MVC controllers where `Edit(int)` shows a form to edit a product, and `Edit(int, Product)` processes the posted form. To make this possible MVC would need to choose `Edit(int, Product)` when the request is an HTTP `POST` and `Edit(int)` when the HTTP verb is anything else.

The `HttpPostAttribute` (`[HttpPost]`) is an implementation of `IActionConstraint` that will only allow the action to be selected when the HTTP verb is `POST`. The presence of an `IActionConstraint` makes the `Edit(int, Product)` a 'better' match than `Edit(int)`, so `Edit(int, Product)` will be tried first.

You will only need to write custom `IActionConstraint` implementations in specialized scenarios, but it's important to understand the role of attributes like `HttpPostAttribute` - similar attributes are defined for other HTTP verbs. In conventional routing it's common for actions to use the same action name when they are part of a `show form -> submit form` workflow. The convenience of this pattern will become more apparent after reviewing the Understanding IActionConstraint section.

If multiple routes match, and MVC can't find a 'best' route, it will throw an `AmbiguousActionException`.

Route names

The strings `"blog"` and `"default"` in the following examples are route names:

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
               defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The route names give the route a logical name so that the named route can be used for URL generation. This greatly simplifies URL creation when the ordering of routes could make URL generation complicated. Routes names must be unique application-wide.

Route names have no impact on URL matching or handling of requests; they are used only for URL generation. Routing has more detailed information on URL generation including URL generation in MVC-specific helpers.

Attribute routing

Attribute routing uses a set of attributes to map actions directly to route templates. In the following example, `app.UseMvc();` is used in the `Configure` method and no route is passed. The `HomeController` will match a set of URLs similar to what the default route `{controller=Home}/{action=Index}/{id?}` would match:

```
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();
   }
}
```

The `HomeController.Index()` action will be executed for any of the URL paths `/`, `/Home`, or `/Home/Index`.

Note

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route; the conventional default route handles routes more succinctly. However, attribute routing allows (and requires) precise control of which route templates apply to each action.

With attribute routing the controller name and action names play **no** role in which action is selected. This example will match the same URLs as the previous example.

```
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");
   }
}
```

Note

The route templates above don't define route parameters for `action`, `area`, and `controller`. In fact, these route parameters are not allowed in attribute routes. Since the route template is already associated with an action, it wouldn't make sense to parse the action name from the URL.

Attribute routing with Http[Verb] attributes

Attribute routing can also make use of the `Http[Verb]` attributes such as `HttpPostAttribute`. All of these attributes can accept a route template. This example shows two actions that match the same route template:

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
   // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
   // ...
}
```

For a URL path like `/products` the `ProductsApi.ListProducts` action will be executed when the HTTP verb is `GET` and `ProductsApi.CreateProduct` will be executed when the HTTP verb is `POST`. Attribute routing first matches the URL against the set of route templates defined by route attributes. Once a route template matches, `IActionConstraint` constraints are applied to determine which actions can be executed.

涉 Tip

When building a REST API, it's rare that you will want to use `[Route(...)]` on an action method. It's better to use the more specific `Http*Verb*Attributes` to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In this example, `id` is required as part of the URL path.

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

The `ProductsApi.GetProduct(int)` action will be executed for a URL path like `/products/3` but not for a URL path like `/products`. See Routing for a full description of route templates and related options.

Route Name

The following code defines a *route name* of `Products_List`:

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

Route names can be used to generate a URL based on a specific route. Route names have no impact on the URL matching behavior of routing and are only used for URL generation. Route names must be unique application-wide.

 Note

Contrast this with the conventional *default route*, which defines the `id` parameter as optional (`{id?}`). This ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

```
[Route("products")]
public class ProductsApiController : Controller
{
   [HttpGet]
   public IActionResult ListProducts() { ... }

   [HttpGet("{id}")]
   public ActionResult GetProduct(int id) { ... }
}
```

In this example the URL path `/products` can match `ProductsApi.ListProducts`, and the URL path `/products/5` can match `ProductsApi.GetProduct(int)`. Both of these actions only match HTTP `GET` because they are decorated with the `HttpGetAttribute`.

Route templates applied to an action that begin with a `/` do not get combined with route templates applied to the controller. This example matches a set of URL paths similar to the *default route*.

```
[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("/")]     // Does not 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();
    }
}
```

Ordering attribute routes

In contrast to conventional routes which execute in a defined order, attribute routing builds a tree and matches all routes simultaneously. This behaves as-if the route entries were placed in an ideal ordering; the most specific routes have a chance to execute before the more general routes.

For example, a route like `blog/search/{topic}` is more specific than a route like `blog/{*article}`. Logically speaking the `blog/search/{topic}` route 'runs' first, by default, because that's the only sensible ordering. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order, using the `Order` property of all of the framework provided route attributes. Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` will run before routes that don't set an order. Setting a route using `Order = 1` will run after default route ordering.

> 汯 Tip
>
> Avoid depending on `Order`. If your URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general attribute routing will select the correct route with URL matching. If the default order used for URL generation isn't working, using route name as an override is usually simpler than applying the `Order` property.

Token replacement in route templates ([controller], [action], [area])

For convenience, attribute routes support *token replacement* by enclosing a token in square-braces (`[`, `]`). The tokens `[action]`, `[area]`, and `[controller]` will be replaced with the values of the action name, area name, and controller name from the action where the route is defined. In this example the actions can match URL paths as described in the comments:

```
[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) {
        // ...
    }
}
```

Token replacement occurs as the last step of building the attribute routes. The above example will behave the same as the following code:

```

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) {
        // ...
    }
}
```

Attribute routes can also be combined with inheritance. This is particularly powerful combined with token replacement.

```
[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
   [HttpGet] // Matches '/api/Products'
   public IActionResult List() { ... }

   [HttpPost("{id}")] // Matches '/api/Products/{id}'
   public IActionResult Edit(int id) { ... }
}
```

Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` will generate a unique route name for each action.

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Multiple Routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the *default conventional route* as shown in the following example:

```
[Route("[controller]")]
public class ProductsController : Controller
{
   [Route("")]     // Matches 'Products'
   [Route("Index")] // Matches 'Products/Index'
   public IActionResult Index()
}
```

Putting multiple route attributes on the controller means that each one will combine with each of the route attributes on the action methods.

```
[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()
}
```

When multiple route attributes (that implement `IActionConstraint`) are placed on an action, then each action constraint combines with the route template from the attribute that defined it.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
   [HttpPut("Buy")]      // Matches PUT 'api/Products/Buy'
   [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
   public IActionResult Buy()
}
```

 Tip

While using multiple routes on actions can seem powerful, it's better to keep your application's URL space simple and well-defined. Use multiple routes on actions only where needed, for example to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
   // ...
}
```

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using `IRouteTemplateProvider`

All of the route attributes provided in the framework (`[Route(...)]`, `[HttpGet(...)]` , etc.) implement the `IRouteTemplateProvider` interface. MVC looks for attributes on controller classes and action methods when the app starts and uses the ones that implement `IRouteTemplateProvider` to build the initial set of routes.

You can implement `IRouteTemplateProvider` to define your own route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
   public string Template => "api/[controller]";

   public int? Order { get; set; }

   public string Name { get; set; }
}
```

The attribute from the above example automatically sets the `Template` to `"api/[controller]"` when `[MyApiController]` is applied.

Using Application Model to customize attribute routes

The *application model* is an object model created at startup with all of the metadata used by MVC to route and execute your actions. The *application model* includes all of the data gathered from route attributes (through `IRouteTemplateProvider`). You can write *conventions* to modify the application model at startup time to customize how routing behaves. This section shows a simple example of customizing routing using application model.

```
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;

        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()
            };
        }
    }
}
```

Mixed routing: Attribute routing vs conventional routing

MVC applications can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. **Any** route attribute on the controller makes all actions in the controller attribute routed.

Note

What distinguishes the two types of routing systems is the process applied after a URL matches a route template. In conventional routing, the route values from the match are used to choose the action and controller from a lookup table of all conventional routed actions. In attribute routing, each template is already associated with an action, and no further lookup is needed.

URL Generation

MVC applications can use routing's URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making your code more robust and maintainable. This section focuses on the URL generation features provided by MVC and will only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The `IUrlHelper` interface is the underlying piece of infrastructure between MVC and routing for URL generation. You'll find an instance of `IUrlHelper` available through the `Url` property in controllers, views, and view components.

In this example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

```
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();
    }
}
```

If the application is using the default conventional route, the value of the `url` variable will be the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining the route values from the current request (ambient values), with the values passed to `Url.Action` and substituting those values into the route template:

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that does not have a value can use a default value if it has one, or be skipped if it is optional (as in the case of `id` in this example). URL generation will fail if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The example of `Url.Action` above assumes conventional routing, but URL generation works similarly with attribute routing, though the concepts are different. With conventional routing, the route values are used to expand a template, and the route values for `controller` and `action` usually appear in that template - this works because the URLs matched by routing adhere to a *convention*. In attribute routing, the route values for `controller` and `action` are not allowed to appear in the template - they are instead used to look up which template to use.

This example uses attribute routing:

```
// 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 builds a lookup table of all attribute routed actions and will match the `controller` and `action` values to select the route template to use for URL generation. In the sample above, `custom/url/to/destination` is generated.

Generating URLs by action name

`Url.Action` (`IUrlHelper` . `Action`) and all related overloads all are based on that idea that you want to specify what you're linking to by specifying a controller name and action name.

Note

When using `Url.Action`, the current route values for `controller` and `action` are specified for you - the value of `controller` and `action` are part of both *ambient values* **and** *values*. The method `Url.Action`, always uses the current values of `action` and `controller` and will generate a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that you didn't provide when generating a URL. Using a route like `{a}/{b}/{c}/{d}` and ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`, routing has enough information to generate a URL without any additional values - since all route parameters have a value. If you added the value `{ d = Donovan }`, the value `{ d = David }` would be ignored, and the generated URL path would be `Alice/Bob/Carol/Donovan`.

Warning

URL paths are hierarchical. In the example above, if you added the value `{ c = Cheryl }`, both of the values `{ c = Carol, d = David }` would be ignored. In this case we no longer have a value for `d` and URL generation will fail. You would need to specify the desired value of `c` and `d`. You might expect to hit this problem with the default route (`{controller}/{action}/{id?}`) - but you will rarely encounter this behavior in practice as `Url.Action` will always explicitly specify a `controller` and `action` value.

Longer overloads of `Url.Action` also take an additional *route values* object to provide values for route parameters other than `controller` and `action`. You will most commonly see this used with `id` like `Url.Action("Buy", "Products", new { id = 17 })`. By convention the *route values* object is usually an object of anonymous type, but it can also be an `IDictionary<>` or a *plain old .NET object*. Any additional route values that don't match route parameters are put in the query string.

```
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);
    }
}
```

 Tip

To create an absolute URL, use an overload that accepts a `protocol`:
`Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme)`

Generating URLs by route

The code above demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the `Url.RouteUrl` family of methods. These methods are similar to `Url.Action`, but they do not copy the current values of `action` and `controller` to the route values. The most common usage is to specify a route name to use a specific route to generate the URL, generally *without* specifying a controller or action name.

```
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();
    }
}
```

Generating URLs in HTML

`IHtmlHelper` provides the `HtmlHelper` methods `Html.BeginForm` and `Html.ActionLink` to generate `<form>` and `<a>` elements respectively. These methods use the `Url.Action` method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Working with Forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

Generating URLS in Action Results

The examples above have shown using `IUrlHelper` in a controller, while the most common usage in a controller is to generate a URL as part of an action result.

The `ControllerBase` and `Controller` base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input.

```
public Task<IActionResult> Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        return RedirectToAction("Index");
    }
}
```

The action results factory methods follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a *dedicated conventional route*. In the example below, the route named `blog` is a dedicated conventional route.

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Using these route definitions, `Url.Action("Index", "Home")` will generate the URL path `/` with the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being "too greedy" with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` will fail because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate routing-namespace (for controller actions) and folder structure (for views). Using areas allows an application to have multiple controllers with the same name - as long as they have different *areas*. Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section will discuss how routing interacts with areas - see Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an *area route* for an area named `Blog`:

```
app.UseMvc(routes =>
{
    routes.MapAreaRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    routes.MapRoute("default_route", "{controller}/{action}/{id?}");
});
```

When matching a URL path like `/Manage/Users/AddUser`, the first route will produce the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`, in fact the route created by `MapAreaRoute` is equivalent to the following:

```
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` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog` . The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

> 䢧 Tip
>
> Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they are more specific than routes without an area.

Using the above example, the route values would match the following action:

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        public IActionResult AddUser()
        {
            return View();
        }
    }
}
```

The `AreaAttribute` is what denotes a controller as part of an area, we say that this controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and will **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

```
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();

        }
    }
}
```

Note

The namespace of each controller is shown here for completeness - otherwise the controllers would have a naming conflict and generate a compiler error. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller is not a member of any area, and can only match when no value for `area` is provided by routing.

Note

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` will be available as an *ambient value* for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

```
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);
        }
    }
}
```

Understanding IActionConstraint

Note

This section is a deep-dive on framework internals and how MVC chooses an action to execute. A typical application won't need a custom `IActionConstraint`

You have likely already used `IActionConstraint` even if you're not familiar with the interface. The `[HttpGet]` Attribute and similar `[Http-VERB]` attributes implement `IActionConstraint` in order to limit the execution of an action method.

```
public class ProductsController : Controller
{
    [HttpGet]
    public IActionResult Edit() { }

    public IActionResult Edit(...) { }
}
```

Assuming the default conventional route, the URL path `/Products/Edit` would produce the values `{ controller = Products, action = Edit }`, which would match **both** of the actions shown here. In `IActionConstraint` terminology we would say that both of these actions are considered candidates - as they both match the route data.

When the `HttpGetAttribute` executes, it will say that *Edit()* is a match for *GET* and is not a match for any other HTTP verb. The `Edit(...)` action doesn't have any constraints defined, and so will match any HTTP verb. So assuming a `POST` - only `Edit(...)` matches. But, for a `GET` both actions can still match - however, an action with an `IActionConstraint` is always considered *better* than an action without. So because `Edit()` has `[HttpGet]` it is considered more specific, and will be selected if both actions can match.

Conceptually, `IActionConstraint` is a form of *overloading*, but instead of overloading methods with the same name, it is overloading between actions that match the same URL. Attribute routing also uses `IActionConstraint` and can result in actions from different controllers both being considered candidates.

Implementing IActionConstraint

The simplest way to implement an `IActionConstraint` is to create a class derived from `System.Attribute` and place it on your actions and controllers. MVC will automatically discover any `IActionConstraint` that are applied as attributes. You can use the application model to apply constraints, and this is probably the most flexible approach as it allows you to metaprogram how they are applied.

In the following example a constraint chooses an action based on a *country code* from the route data. The full sample on GitHub.

```
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);
    }
}
```

You are responsible for implementing the `Accept` method and choosing an 'Order' for the constraint to execute. In this case, the `Accept` method returns `true` to denote the action is a match when the `country` route value matches. This is different from a `RouteValueAttribute` in that it allows fallback to a non-attributed action. The sample shows that if you define an `en-US` action then a country code like `fr-FR` will fall back to a more generic controller that does not have `[CountrySpecific(...)]` applied.

The `Order` property decides which *stage* the constraint is part of. Action constraints run in groups based on the `Order`. For example, all of the framework provided HTTP method attributes use the same `Order` value so that they run in the same stage. You can have as many stages as you need to implement your desired policies.

 Tip

To decide on a value for `Order` think about whether or not your constraint should be applied before HTTP methods. Lower numbers run first.

Model Binding

By Rachel Appel

Introduction to model binding

Model binding in ASP.NET Core MVC maps data from HTTP requests to action method parameters. The parameters may be simple types such as strings, integers, or floats, or they may be complex types. This is a great feature of MVC because mapping incoming data to a counterpart is an often repeated scenario, regardless of size or complexity of the data. MVC solves this problem by abstracting binding away so developers don't have to keep rewriting a slightly different version of that same code in every app. Writing your own text to type converter code is tedious, and error prone.

How model binding works

When MVC receives an HTTP request, it routes it to a specific action method of a controller. It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action method's parameters. For example, consider the following URL:

`http://contoso.com/movies/edit/2`

Since the route template looks like this, `{controller=Home}/{action=Index}/{id?}`, `movies/edit/2` routes to the `Movies` controller, and its `Edit` action method. It also accepts an optional parameter called `id`. The code for the action method should look something like this:

```
public IActionResult Edit(int? id)
```

Note: The strings in the URL route are not case sensitive.

MVC will try to bind request data to the action parameters by name. MVC will look for values for each parameter using the parameter name and the names of its public settable properties. In the above example, the only action parameter is named `id`, which MVC binds to the value with the same name in the route values. In addition to route values MVC will bind data from various parts of the request and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:

`Form values`: These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).

`Route values`: The set of route values provided by Routing

`Query strings`: The query string part of the URI.

Note: Form values, route data, and query strings are all stored as name-value pairs.

Since model binding asked for a key named `id` and there is nothing named `id` in the form values, it moved on to the route values looking for that key. In our example, it's a match. Binding happens, and the value is converted to the integer 2. The same request using Edit(string id) would convert to the string "2".

So far the example uses simple types. In MVC simple types are any .NET primitive type or type with a string type converter. If the action method's parameter were a class such as the `Movie` type, which contains both simple and complex types as properties, MVC's model binding will still handle it nicely. It uses reflection and recursion to traverse the properties of complex types looking for matches. Model binding looks for the pattern *parameter_name.property_name* to bind values to properties. If it doesn't find matching values of this form, it will attempt to bind using just the property name. For those types such as `Collection` types, model binding looks for matches to *parameter_name[index]* or just *[index]*. Model binding treats `Dictionary` types similarly, asking for *parameter_name[key]* or just *[key]*, as long as the keys are simple types. Keys that are supported match the field names HTML and tag helpers generated for the same model type. This enables round-tripping values so that the form fields remain filled with the user's input for their convenience, for example, when bound data from a create or edit did not pass validation.

In order for binding to happen the class must have a public default constructor and member to be bound must be public writable properties. When model binding happens the class will only be instantiated using the public default constructor, then the properties can be set.

When a parameter is bound, model binding stops looking for values with that name and it moves on to bind the next parameter. If binding fails, MVC does not throw an error. You can query for model state errors by checking the `ModelState.IsValid` property.

Note: Each entry in the controller's `ModelState` property is a `ModelStateEntry` containing an `Errors property`. It's rarely necessary to query this collection yourself. Use `ModelState.IsValid` instead.

Additionally, there are some special data types that MVC must consider when performing model binding:

`IFormFile`, `IEnumerable<IFormFile>`: One or more uploaded files that are part of the HTTP request.

`CancelationToken`: Used to cancel activity in asynchronous controllers.

These types can be bound to action parameters or to properties on a class type.

Once model binding is complete, Validation occurs. Default model binding works great for the vast majority of development scenarios. It is also extensible so if you have unique needs you can customize the built-in behavior.

Customize model binding behavior with attributes

MVC contains several attributes that you can use to direct its default model binding behavior to a different source. For example, you can specify whether binding is required for a property, or if it should never happen at all by using the `[BindRequired]` or `[BindNever]` attributes. Alternatively, you can override the default data source, and specify the model binder's data source. Below is a list of model binding attributes:

`[BindRequired]`: This attribute adds a model state error if binding cannot occur.

`[BindNever]`: Tells the model binder to never bind to this parameter.

`[FromHeader]`, `[FromQuery]`, `[FromRoute]`, `[FromForm]`: Use these to specify the exact binding source you want to apply.

`[FromServices]`: This attribute uses dependency injection to bind parameters from services.

`[FromBody]`: Use the configured formatters to bind data from the request body. The formatter is selected based on content type of the request.

`[ModelBinder]`: Used to override the default model binder, binding source and name.

Attributes are very helpful tools when you need to override the default behavior of model binding.

Binding formatted data from the request body

Request data can come in a variety of formats including JSON, XML and many others. When you use the [FromBody] attribute to indicate that you want to bind a parameter to data in the request body, MVC uses a configured set of formatters to handle the request data based on its content type. By default MVC includes a `JsonInputFormatter` class for handling JSON data, but you can add additional formatters for handling XML and other custom formats.

 Note

There can be at most one parameter per action decorated with `[FromBody]`. The ASP.NET Core MVC run-time delegates the responsibility of reading the request stream to the formatter. Once the request stream is read for a parameter, it's generally not possible to read the request stream again for binding other `[FromBody]` parameters.

 Note

The `JsonInputFormatter` is the default formatter and is based on Json.NET.

ASP.NET selects input formatters based on the Content-Type header and the type of the parameter, unless there is an attribute applied to it specifying otherwise. If you'd like to use XML or another format you must configure it in the *Startup.cs* file, but you may first have to obtain a reference to `Microsoft.AspNetCore.Mvc.Formatters.Xml` using NuGet. Your startup code should look something like this:

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

Code in the *Startup.cs* file contains a `ConfigureServices` method with a `services` argument you can use to build up services for your ASP.NET app. In the sample, we are adding an XML formatter as a service that MVC will provide for this app. The `options` argument passed into the `AddMvc` method allows you to add and manage filters, formatters, and other system options from MVC upon app startup. Then apply the `Consumes` attribute to controller classes or action methods to work with the format you want.

Custom Model Binding

You can extend model binding by writing your own custom model binders. Learn more about custom model binding.

Introduction to model validation in ASP.NET Core MVC

By Rachel Appel

Introduction to model validation

Before an app stores data in a database, the app must validate the data. Data must be checked for potential security threats, verified that it is appropriately formatted by type and size, and it must conform to your rules. Validation is necessary although it can be redundant and tedious to implement. In MVC, validation happens on both the client and server.

Fortunately, .NET has abstracted validation into validation attributes. These attributes contain validation code, thereby reducing the amount of code you must write.

Validation Attributes

Validation attributes are a way to configure model validation so it's similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card, phone number, or email address. Validation attributes make enforcing these requirements much simpler and easier to use.

Below is an annotated `Movie` model from an app that stores information about movies and TV shows. Most of the properties are required and several string properties have length requirements. Additionally, there is a numeric range restriction in place for the `Price` property from 0 to $999.99, along with a custom validation attribute.

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

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [Required]
    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Required]
    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Simply reading through the model reveals the rules about data for this app, making it easier to maintain the code. Below are several popular built-in validation attributes:

`[CreditCard]`: Validates the property has a credit card format.

`[Compare]`: Validates two properties in a model match.

`[EmailAddress]`: Validates the property has an email format.

`[Phone]`: Validates the property has a telephone format.

`[Range]`: Validates the property value falls within the given range.

`[RegularExpression]`: Validates that the data matches the specified regular expression.

`[Required]`: Makes a property required.

`[StringLength]`: Validates that a string property has at most the given maximum length.

`[Url]`: Validates the property has a URL format.

MVC supports any attribute that derives from `ValidationAttribute` for validation purposes. Many useful validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

There may be instances where you need more features than built-in attributes provide. For those times, you can create custom validation attributes by deriving from `ValidationAttribute` or changing your model to implement `IValidatableObject`.

Model State

Model state represents validation errors in submitted HTML form values.

MVC will continue validating fields until reaches the maximum number of errors (200 by default). You can configure this number by inserting the following code into the `ConfigureServices` method in the `Startup.cs` file:

```
services.AddMvc(options => options.MaxModelValidationErrors = 50);
```

Handling Model State Errors

Model validation occurs prior to each controller action being invoked, and it is the action method's responsibility to inspect `ModelState.IsValid` and react appropriately. In many cases, the appropriate reaction is to return some kind of error response, ideally detailing the reason why model validation failed.

Some apps will choose to follow a standard convention for dealing with model validation errors, in which case a filter may be an appropriate place to implement such a policy. You should test how your actions behave with valid and invalid model states.

Manual validation

After model binding and validation are complete, you may want to repeat parts of it. For example, a user may have entered text in a field expecting an integer, or you may need to compute a value for a model's property.

You may need to run validation manually. To do so, call the `TryValidateModel` method, as shown here:

```
TryValidateModel(movie);
```

Custom validation

Validation attributes work for most validation needs. However, some validation rules are specific to your business, as they're not just generic data validation such as ensuring a field is required or that it conforms to a range of values. For these scenarios, custom validation attributes are a great solution. Creating your own custom validation attributes in MVC is easy. Just inherit from the `ValidationAttribute`, and override the `IsValid` method. The `IsValid` method accepts two parameters, the first is an object named *value* and the second is a `ValidationContext` object named *validationContext*. *Value* refers to the actual value from the field that your custom validator is validating.

In the following sample, a business rule states that users may not set the genre to *Classic* for a movie released after 1960. The `[ClassicMovie]` attribute checks the genre first, and if it is a classic, then it checks the release date to see that it is later than

1960. If it is released after 1960, validation fails. The attribute accepts an integer parameter representing the year that you can use to validate data. You can capture the value of the parameter in the attribute's constructor, as shown here:

```
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;
    }
```

The `movie` variable above represents a `Movie` object that contains the data from the form submission to validate. In this case, the validation code checks the date and genre in the `IsValid` method of the `ClassicMovieAttribute` class as per the rules. Upon successful validation `IsValid` returns a `ValidationResult.Success` code, and when validation fails, a `ValidationResult` with an error message. When a user modifies the `Genre` field and submits the form, the `IsValid` method of the `ClassicMovieAttribute` will verify whether the movie is a classic. Like any built-in attribute, apply the `ClassicMovieAttribute` to a property such as `ReleaseDate` to ensure validation happens, as shown in the previous code sample. Since the example works only with `Movie` types, a better option is to use `IValidatableObject` as shown in the following paragraph.

Alternatively, this same code could be placed in the model by implementing the `Validate` method on the `IValidatableObject` interface. While custom validation attributes work well for validating individual properties, implementing `IValidatableObject` can be used to implement class-level validation as seen here.

```
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" });
    }
}
```

Client side validation

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

You must have a view with the proper JavaScript script references in place for client side validation to work as you see here.

```
<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.3.min.js"></script>
```

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.min.js"></script>
<script
src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/3.2.6/jquery.validate.unobtrusive.min.js">
</script>
```

MVC uses validation attributes in addition to type metadata from model properties to validate data and display any error messages using JavaScript. When you use MVC to render form elements from a model using Tag Helpers or HTML helpers it will add HTML 5 data- attributes in the form elements that need validation, as shown below. MVC generates the `data-` attributes for both built-in and custom attributes. You can display validation errors on the client using the relevant tag helpers as shown here:

```
<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>
```

The tag helpers above render the HTML below. Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `ReleaseDate` property. The `data-val-required` attribute below contains an error message to display if the user doesn't fill in the release date field, and that message displays in the accompanying `<span>` element.

```
<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>
```

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

MVC determines type attribute values based on the .NET data type of a property, possibly overridden using `[DataType]` attributes. The base `[DataType]` attribute does no real server-side validation. Browsers choose their own error messages and display those errors however they wish, however the jQuery Validation Unobtrusive package can override the messages and display them consistently with others. This happens most obviously when users apply `[DataType]` subclasses such as `[EmailAddress]`.

IClientModelValidator

You may create client side logic for your custom attribute, and unobtrusive validation will execute it on the client for you automatically as part of validation. The first step is to control what data- attributes are added by implementing the `IClientModelValidator` interface as shown here:

```
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);
}
```

Attributes that implement this interface can add HTML attributes to generated fields. Examining the output for the `ReleaseDate` element reveals HTML that is similar to the previous example, except now there is a `data-val-classicmovie` attribute that was defined in the `AddValidation` method of `IClientModelValidator`.

```
<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="" />
```

Unobtrusive validation uses the data in the `data-` attributes to display error messages. However, jQuery doesn't know about rules or messages until you add them to jQuery's `validator` object. This is shown in the example below that adds a method named `classicmovie` containing custom client validation code to the jQuery `validator` object.

Now jQuery has the information to execute the custom JavaScript validation as well as the error message to display if that validation code returns false.

Remote validation

Remote validation is a great feature to use when you need to validate data on the client against data on the server. For example, your app may need to verify whether an email or user name is already in use, and it must query a large amount of data to do so. Downloading large sets of data for validating one or a few fields consumes too many resources. It may also expose sensitive information. An alternative is to make a round-trip request to validate a field.

You can implement remote validation in a two step process. First, you must annotate your model with the `[Remote]` attribute. The `[Remote]` attribute accepts multiple overloads you can use to direct client side JavaScript to the appropriate code to call. The example points to the `VerifyEmail` action method of the `Users` controller.

```
public class User
{
    [Remote(action: "VerifyEmail", controller: "Users")]
    public string Email { get; set; }
}
```

The second step is putting the validation code in the corresponding action method as defined in the `[Remote]` attribute. It returns a `JsonResult` that the client side can use to proceed or pause and display an error if needed.

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json(data: $"Email {email} is already in use.");
    }

    return Json(data: true);
}
```

Now when users enter an email, JavaScript in the view makes a remote call to see if that email has been taken, and if so, then displays the error message. Otherwise, the user can submit the form as usual.

File uploads

By Steve Smith

ASP.NET MVC actions support uploading of one or more files using simple model binding for smaller files or streaming for larger files.

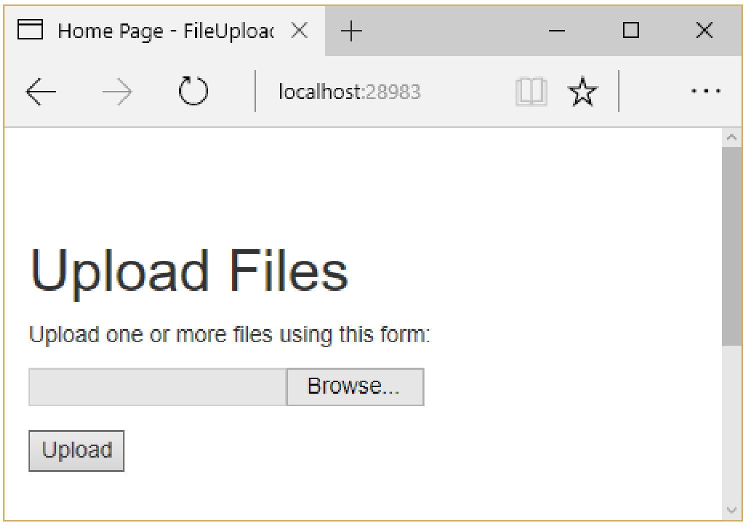
View or download sample from GitHub

Uploading small files with model binding

To upload small files, you can use a multi-part HTML form or construct a POST request using JavaScript. An example form using Razor, which supports multiple uploaded files, is shown below:

```
<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>
```

In order to support file uploads, HTML forms must specify an `enctype` of `multipart/form-data`. The `files` input element shown above supports uploading multiple files. Omit the `multiple` attribute on this input element to allow just a single file to be uploaded. The above markup renders in a browser as:



The individual files uploaded to the server can be accessed through Model Binding using the IFormFile interface. `IFormFile` has the following structure:

```
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);
}
```

Warning

Don't rely on or trust the `FileName` property without validation. The `FileName` property should only be used for display purposes.

When uploading files using model binding and the `IFormFile` interface, the action method can accept either a single `IFormFile` or an `IEnumerable<IFormFile>` (or `List<IFormFile>`) representing several files. The following example loops through one or more uploaded files, saves them to the local file system, and returns the total number and size of files uploaded.

```
[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});
}
```

Files uploaded using the `IFormFile` technique are buffered in memory or on disk on the web server before being processed. Inside the action method, the `IFormFile` contents are accessible as a stream. In addition to the local file system, files can be streamed to Azure Blob storage or Entity Framework.

To store binary file data in a database using Entity Framework, define a property of type `byte[]` on the entity:

```
public class ApplicationUser : IdentityUser
{
    public byte[] AvatarImage { get; set; }
}
```

Specify a viewmodel property of type `IFormFile`:

```
public class RegisterViewModel
{
    // other properties omitted

    public IFormFile AvatarImage { get; set; }
}
```

▯ Note

`IFormFile` can be used directly as an action method parameter or as a viewmodel property, as shown above.

Copy the `IFormFile` to a stream and save it to the byte array:

```
// 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.
}
```

▯ Note

Use caution when storing binary data in relational databases, as it can adversely impact performance.

Uploading large files with streaming

If the size or frequency of file uploads is causing resource problems for the app, consider streaming the file upload rather than buffering it in its entirety, as the model binding approach shown above does. While using `IFormFile` and model binding is a much simpler solution, streaming requires a number of steps to implement properly.

▯ Note

Any single buffered file exceeding 64KB will be moved from RAM to a temp file on disk on the server. The resources (disk, RAM) used by file uploads depend on the number and size of concurrent file uploads. Streaming is not so much about perf, it's about scale. If you try to buffer too many uploads, your site will crash when it runs out of memory or disk space.

The following example demonstrates using JavaScript/Angular to stream to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed in HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, model binding is disabled by another filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. Once all sections have been read, the action performs its own model binding.

The initial action loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieForAjax` attribute):

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

The attribute uses ASP.NET Core's built-in Antiforgery support to set a cookie with a request token:

```
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 automatically passes an antiforgery token in a request header named `X-XSRF-TOKEN`. The ASP.NET Core MVC app is configured to refer to this header in its configuration in *Startup.cs*:

```
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();
}
```

The `DisableFormValueModelBinding` attribute, shown below, is used to disable model binding for the `Upload` action method.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var formValueProviderFactory = context.ValueProviderFactories
            .OfType<FormValueProviderFactory>()
            .FirstOrDefault();
        if (formValueProviderFactory != null)
        {
            context.ValueProviderFactories.Remove(formValueProviderFactory);
        }

        var jqueryFormValueProviderFactory = context.ValueProviderFactories
            .OfType<JQueryFormValueProviderFactory>()
            .FirstOrDefault();
        if (jqueryFormValueProviderFactory != null)
        {
            context.ValueProviderFactories.Remove(jqueryFormValueProviderFactory);
        }
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

Since model binding is disabled, the `Upload` action method doesn't accept parameters. It works directly with the `Request` property of `ControllerBase`. A `MultipartReader` is used to read each section. The file is saved with a GUID filename and the the key/value data is stored in a `KeyValueAccumulator`. Once all sections have been read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `Upload` method is shown below:

```
// 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;
```

```
        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,
                    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);
}
```

Troubleshooting

Below are some common problems encountered when working with uploading files and their possible solutions.

Unexpected Not Found error with IIS

The following error indicates your file upload exceeds the server's configured `maxAllowedContentLength`:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

The default setting is `30000000`, which is approximately 28.6MB. The value can be customized by editing *web.config*:

```
<system.webServer>
  <security>
    <requestFiltering>
      <!-- This will handle requests up to 50MB -->
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

This setting only applies to IIS. The behavior doesn't occur by default when hosting on Kestrel. For more information, see Request Limits <requestLimits>.

Null Reference Exception with IFormFile

If your controller is accepting uploaded files using `IFormFile` but you find that the value is always null, confirm that your HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute is not set on the `<form>` element, the file upload will not occur and any bound `IFormFile` arguments will be null.

Dependency injection into controllers

By Steve Smith

ASP.NET Core MVC controllers should request their dependencies explicitly via their constructors. In some instances, individual controller actions may require a service, and it may not make sense to request at the controller level. In this case, you can also choose to inject a service as a parameter on the action method.

View or download sample code

Dependency Injection

Dependency injection is a technique that follows the Dependency Inversion Principle, allowing for applications to be composed of loosely coupled modules. ASP.NET Core has built-in support for dependency injection, which makes applications easier to test and maintain.

Constructor Injection

ASP.NET Core's built-in support for constructor-based dependency injection extends to MVC controllers. By simply adding a service type to your controller as a constructor parameter, ASP.NET Core will attempt to resolve that type using its built in service container. Services are typically, but not always, defined using interfaces. For example, if your application has business logic that depends on the current time, you can inject a service that retrieves the time (rather than hard-coding it), which would allow your tests to pass in implementations that use a set time.

```
using System;

namespace ControllerDI.Interfaces
{
    public interface IDateTime
    {
        DateTime Now { get; }
    }
}
```

Implementing an interface like this one so that it uses the system clock at runtime is trivial:

```
using System;
using ControllerDI.Interfaces;

namespace ControllerDI.Services
{
    public class SystemDateTime : IDateTime
    {
        public DateTime Now
        {
            get { return DateTime.Now; }
        }
    }
}
```

With this in place, we can use the service in our controller. In this case, we have added some logic to the `HomeController` `Index` method to display a greeting to the user based on the time of day.

```
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();
        }
    }
}
```

If we run the application now, we will most likely encounter an error:

```
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)
```

This error occurs when we have not configured a service in the `ConfigureServices` method in our `Startup` class. To specify that requests for `IDateTime` should be resolved using an instance of `SystemDateTime`, add the highlighted line in the listing below to your `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IDateTime, SystemDateTime>();
}
```

Note

This particular service could be implemented using any of several different lifetime options (`Transient`, `Scoped`, or `Singleton`). See Dependency Injection to understand how each of these scope options will affect the behavior of your service.

Once the service has been configured, running the application and navigating to the home page should display the time-based message as expected:

溌 Tip

See Testing Controller Logic to learn how to explicitly request dependencies http://deviq.com/explicit-dependencies-principle in controllers makes code easier to test.

ASP.NET Core's built-in dependency injection supports having only a single constructor for classes requesting services. If you have more than one constructor, you may get an exception stating:

```
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)
```

As the error message states, you can correct this problem having just a single constructor. You can also replace the default dependency injection support with a third party implementation, many of which support multiple constructors.

Action Injection with FromServices

Sometimes you don't need a service for more than one action within your controller. In this case, it may make sense to inject the service as a parameter to the action method. This is done by marking the parameter with the attribute `[FromServices]` as shown here:

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;

    return View();
}
```

Accessing Settings from a Controller

Accessing application or configuration settings from within a controller is a common pattern. This access should use the Options pattern described in configuration. You generally should not request settings directly from your controller using dependency injection. A better approach is to request an `IOptions<T>` instance, where `T` is the configuration class you need.

To work with the options pattern, you need to create a class that represents the options, such as this one:

```
namespace ControllerDI.Model
{
    public class SampleWebSettings
    {
        public string Title { get; set; }
        public int Updates { get; set; }
    }
}
```

Then you need to configure the application to use the options model and add your configuration class to the services collection in `ConfigureServices`:

```
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>();
}
```

Note

In the above listing, we are configuring the application to read the settings from a JSON-formatted file. You can also configure the settings entirely in code, as is shown in the commented code above. See Configuration for further configuration options.

Once you've specified a strongly-typed configuration object (in this case, `SampleWebSettings`) and added it to the services collection, you can request it from any Controller or Action method by requesting an instance of `IOptions<T>` (in this case, `IOptions<SampleWebSettings>`). The following code shows how one would request the settings from a controller:

```
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();
    }
}
```

Following the Options pattern allows settings and configuration to be decoupled from one another, and ensures the controller is following separation of concerns, since it doesn't need to know how or where to find the settings information. It also makes the controller easier to unit test Testing Controller Logic, since there is no static cling or direct instantiation of settings classes within the controller class.

Testing controllers

By Steve Smith

Controllers in ASP.NET MVC apps should be small and focused on user-interface concerns. Large controllers that deal with non-UI concerns are more difficult to test and maintain.

View or download sample from GitHub

Why Test Controllers

Controllers are a central part of any ASP.NET Core MVC application. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production. It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.

Controller logic should be minimal and not be focused on business logic or infrastructure concerns (for example, data access). Test controller logic, not the framework. Test how the controller *behaves* based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs.

Typical controller responsibilities:

Verify `ModelState.IsValid`

Return an error response if `ModelState` is invalid

Retrieve a business entity from persistence

Perform an action on the business entity

Save the business entity to persistence

Return an appropriate `IActionResult`

Unit Testing

Unit testing involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like filters, routing, or model binding. By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of integration testing.

If you're writing custom filters, routes, etc, you should unit test them, but not as part of your tests on a particular controller action. They should be tested in isolation.

> 泚 Tip
>
> Create and run unit tests with Visual Studio.

To demonstrate unit testing, review the following controller. It displays a list of brainstorming sessions and allows new brainstorming sessions to be created with a POST:

```
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);
            }

            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });

            return RedirectToAction("Index");
        }
    }
}
```

The controller is following the explicit dependencies principle, expecting dependency injection to provide it with an instance of `IBrainstormSessionRepository`. This makes it fairly easy to test using a mock object framework, like Moq. The `HTTP GET Index` method has no looping or branching and only calls one method. To test this `Index` method, we need to verify that a `ViewResult` is returned, with a `ViewModel` from the repository's `List` method.

```
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;
        }
    }
}
```

The `HomeController` `HTTP POST Index` method (shown above) should verify:

The action method returns a Bad Request `ViewResult` with the appropriate data when `ModelState.IsValid` is `false`

The `Add` method on the repository is called and a `RedirectToActionResult` is returned with the correct arguments when `ModelState.IsValid` is true.

Invalid model state can be tested by adding errors using `AddModelError` as shown in the first test below.

```
[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();
}
```

The first test confirms when `ModelState` is not valid, the same `ViewResult` is returned as for a `GET` request. Note that the test doesn't attempt to pass in an invalid model. That wouldn't work anyway since model binding isn't running (though an integration test would use exercise model binding). In this case, model binding is not being tested. These unit tests are only testing what the code in the action method does.

The second test verifies that when `ModelState` is valid, a new `BrainstormSession` is added (via the repository), and the method returns a `RedirectToActionResult` with the expected properties. Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows it to be verified in the test. This is done with the call to `mockRepo.Verify`, which will fail the test if the expected method was not called.

Note

The Moq library used in this sample makes it easy to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

Another controller in the app displays information related to a particular brainstorming session. It includes some logic to deal with invalid id values:

```
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("Index", "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);
        }
    }
}
```

The controller action has three cases to test, one for each `return` statement:

```
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;
    }
```

```
    }
}
```

The app exposes functionality as a web API (a list of ideas associated with a brainstorming session and a method for adding new ideas to a session):

```
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);
        }
    }
}
```

The `ForSession` method returns a list of `IdeaDTO` types. Avoid returning your business domain entities directly via API calls, since frequently they include more data than the API client requires, and they unnecessarily couple your app's internal domain model with the API you expose externally. Mapping between domain entities and the types you will return over the wire can be done manually (using a LINQ `Select` as shown here) or using a library like AutoMapper

The unit tests for the `Create` and `ForSession` API methods:

```
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;
        }
    }
}
```

As stated previously, to test the behavior of the method when `ModelState` is invalid, add a model error to the controller as part of the test. Don't try to test model validation or model binding in your unit tests - just test your action method's behavior when confronted with a particular `ModelState` value.

The second test depends on the repository returning null, so the mock repository is configured to return null. There's no need to create a test database (in memory or otherwise) and construct a query that will return this result - it can be done in a single statement as shown.

The last test verifies that the repository's `Update` method is called. As we did previously, the mock is called with `Verifiable` and then the mocked repository's `Verify` method is called to confirm the verifiable method was executed. It's not a unit test responsibility to ensure that the `Update` method saved the data; that can be done with an integration test.

Integration Testing

Integration testing is done to ensure separate modules within your app work correctly together. Generally, anything you can test with a unit test, you can also test with an integration test, but the reverse isn't true. However, integration tests tend to be much slower than unit tests. Thus, it's best to test whatever you can with unit tests, and use integration tests for scenarios that involve multiple collaborators.

Although they may still be useful, mock objects are rarely used in integration tests. In unit testing, mock objects are an effective way to control how collaborators outside of the unit being tested should behave for the purposes of the test. In an integration test, real collaborators are used to confirm the whole subsystem works together correctly.

Application State

One important consideration when performing integration testing is how to set your app's state. Tests need to run independent of one another, and so each test should start with the app in a known state. If your app doesn't use a database or have any persistence, this may not be an issue. However, most real-world apps persist their state to some kind of data store, so any modifications made by one test could impact another test unless the data store is reset. Using the built-in `TestServer`, it's very straightforward to host ASP.NET Core apps within our integration tests, but that doesn't necessarily grant access to the data it will use. If you're using an actual database, one approach is to have the app connect to a test database, which your tests can access and ensure is reset to a known state before each test executes.

In this sample application, I'm using Entity Framework Core's InMemoryDatabase support, so I can't just connect to it from my test project. Instead, I expose an `InitializeDatabase` method from the app's `Startup` class, which I call when the app starts up if it's in the `Development` environment. My integration tests automatically benefit from this as long as they set the environment to `Development`. I don't have to worry about resetting the database, since the InMemoryDatabase is reset each time the app restarts.

The `Startup` class:

```
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());

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

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

```
        {
            loggerFactory.AddConsole(LogLevel.Warning);

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();

                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;
        }
    }
}
```

You'll see the `GetTestSession` method used frequently in the integration tests below.

Accessing Views

Each integration test class configures the `TestServer` that will run the ASP.NET Core app. By default, `TestServer` hosts the web app in the folder where it's running - in this case, the test project folder. Thus, when you attempt to test controller actions that return `ViewResult`, you may see this error:

```
The view 'Index' was not found. The following locations were searched:
(list of locations)
```

To correct this issue, you need to configure the server's content root, so it can locate the views for the project being tested. This is done by a call to `UseContentRoot` in the `TestFixture` class, shown below:

```
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;
using Microsoft.Extensions.PlatformAbstractions;

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 const string SolutionName = "TestingControllersSample.sln";
        private readonly TestServer _server;

        public TestFixture()
            : this(Path.Combine("src"))
        {
        }

        protected TestFixture(string solutionRelativeTargetProjectParentDir)
        {
            var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
            var contentRoot = GetProjectPath(solutionRelativeTargetProjectParentDir, 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 that 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 path that we wish to test
        /// </summary>
        /// <param name="solutionRelativePath">
        /// The parent directory of the target project
```

```
        /// 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 solutionRelativePath, 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 = PlatformServices.Default.Application.ApplicationBasePath;

            // Find the folder which contains the solution file. We then use this information to find the
target
            // project which we want to test.
            var directoryInfo = new DirectoryInfo(applicationBasePath);
            do
            {
                var solutionFileInfo = new FileInfo(Path.Combine(directoryInfo.FullName, SolutionName));
                if (solutionFileInfo.Exists)
                {
                    return Path.GetFullPath(Path.Combine(directoryInfo.FullName, solutionRelativePath,
projectName));
                }

                directoryInfo = directoryInfo.Parent;
            }
            while (directoryInfo.Parent != null);

            throw new Exception($"Solution root could not be located using application root
{applicationBasePath}.");
        }
    }
}
```

The `TestFixture` class is responsible for configuring and creating the `TestServer`, setting up an `HttpClient` to communicate with the `TestServer`. Each of the integration tests uses the `Client` property to connect to the test server and make a request.

```
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.True(responseString.Contains(testSession.Name));
        }

        [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());
        }
    }
}
```

In the first test above, the `responseString` holds the actual rendered HTML from the View, which can be inspected to confirm it contains expected results.

The second test constructs a form POST with a unique session name and POSTs it to the app, then verifies that the expected redirect is returned.

API Methods

If your app exposes web APIs, it's a good idea to have automated tests confirm they execute as expected. The built-in `TestServer` makes it easy to test web APIs. If your API methods are using model binding, you should always check `ModelState.IsValid`, and integration tests are the right place to confirm that your model validation is working properly.

The following set of tests target the `Create` method in the IdeasController class shown above:

```
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.True(returnedSession.Ideas.Any(i => i.Name == testIdeaName));
    }

    [Fact]
    public async Task ForSessionReturnsNotFoundForBadSessionId()
    {
        // Arrange & Act
        var response = await _client.GetAsync("/api/ideas/forsession/500");

        // Assert
        Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
    }

    [Fact]
```

```
        [Fact]
        public async Task ForSessionReturnsIdeasForValidSessionId()
        {
            // 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);
        }
    }
}
```

Unlike integration tests of actions that returns HTML views, web API methods that return results can usually be deserialized as strongly typed objects, as the last test above shows. In this case, the test deserializes the result to a `BrainstormSession` instance, and confirms that the idea was correctly added to its collection of ideas.

You'll find additional examples of integration tests in this article's sample project.

Rendering HTML with views in ASP.NET Core MVC

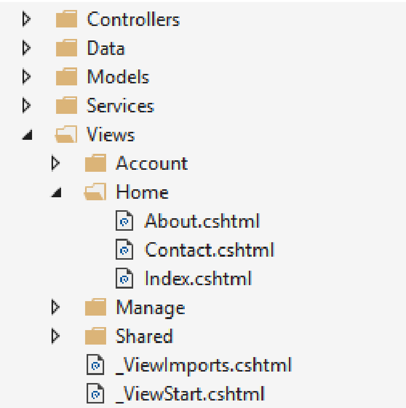
By Steve Smith

ASP.NET MVC Core controllers can return formatted results using *views*.

What are Views?

In the Model-View-Controller (MVC) pattern, the *view* encapsulates the presentation details of the user's interaction with the app. Views are HTML templates with embedded code that generate content to send to the client. Views use Razor syntax, which allows code to interact with HTML with minimal code or ceremony.

ASP.NET Core MVC views are *.cshtml* files stored by default in a *Views* folder within the application. Typically, each controller will have its own folder, in which are views for specific controller actions.



In addition to action-specific views, partial views, layouts, and other special view files can be used to help reduce repetition and allow for reuse within the app's views.

Benefits of Using Views

Views provide separation of concerns within an MVC app, encapsulating user interface level markup separately from business logic. ASP.NET MVC views use Razor syntax to make switching between HTML markup and server side logic painless. Common, repetitive aspects of the app's user interface can easily be reused between views using layout and shared directives or partial views.

Creating a View

Views that are specific to a controller are created in the *Views/[ControllerName]* folder. Views that are shared among controllers are placed in the */Views/Shared* folder. Name the view file the same as its associated controller action, and add the *.cshtml* file extension. For example, to create a view for the *About* action on the *Home* controller, you would create the *About.cshtml* file in the */Views/Home* folder.

A sample view file (*About.cshtml*):

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor code is denoted by the `@` symbol. C# statements are run within Razor code blocks set off by curly braces (`{ }`), such as the assignment of "About" to the `ViewData["Title"]` element shown above. Razor can be used to display values within HTML by simply referencing the value with the `@` symbol, as shown within the `<h2>` and `<h3>` elements above.

This view focuses on just the portion of the output for which it is responsible. The rest of the page's layout, and other common aspects of the view, are specified elsewhere. Learn more about layout and shared view logic.

How do Controllers Specify Views?

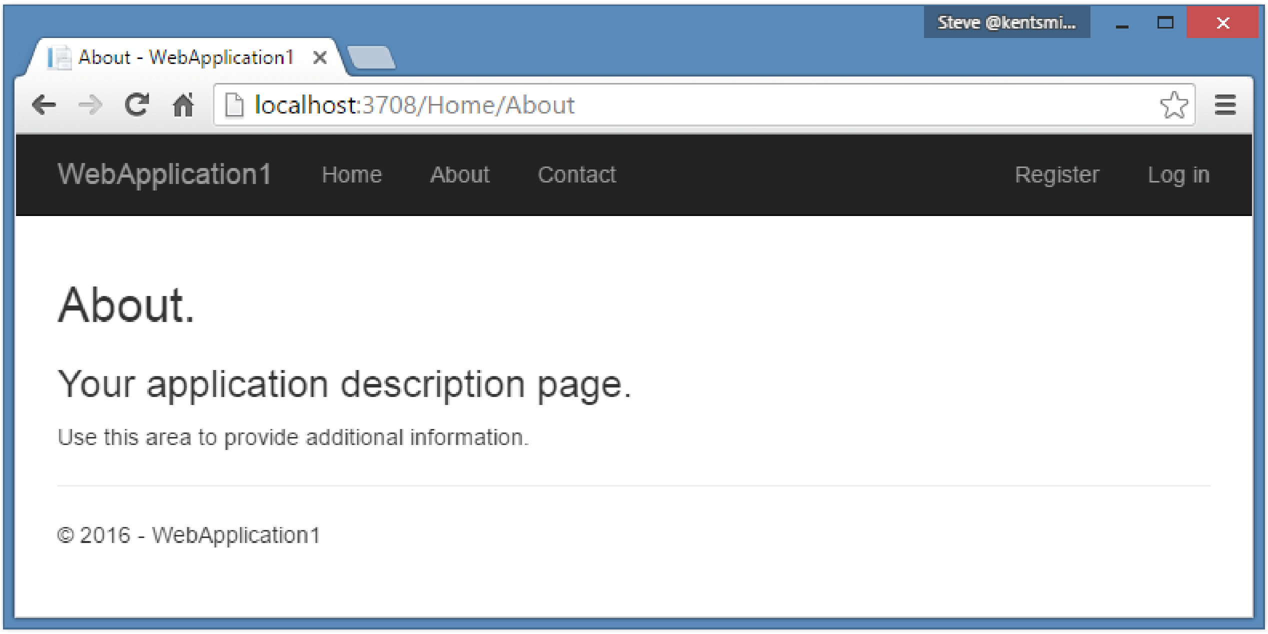
Views are typically returned from actions as a `ViewResult`. Your action method can create and return a `ViewResult` directly, but more commonly if your controller inherits from `Controller`, you'll simply use the `View` helper method, as this example demonstrates:

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

The `View` helper method has several overloads to make returning views easier for app developers. You can optionally specify a view to return, as well as a model object to pass to the view.

When this action returns, the *About.cshtml* view shown above is rendered:



View Discovery

When an action returns a view, a process called *view discovery* takes place. This process determines which view file will be used. Unless a specific view file is specified, the runtime looks for a controller-specific view first, then looks for matching view name in the *Shared* folder.

When an action returns the `View` method, like so `return View();`, the action name is used as the view name. For example, if this were called from an action method named "Index", it would be equivalent to passing in a view name of "Index". A view name can be explicitly passed to the method (`return View("SomeView");`). In both of these cases, view discovery searches for a matching view file in:

Views//.cshtml

Views/Shared/.cshtml

詼 Tip

We recommend following the convention of simply returning `View()` from actions when possible, as it results in more flexible, easier to refactor code.

A view file path can be provided, instead of a view name. In this case, the *.cshtml* extension must be specified as part of the file path. The path should be relative to the application root (and can optionally start with "/" or "~/"). For example: `return View("Views/Home/About.cshtml");`

▯ Note

Partial views and view components use similar (but not identical) discovery mechanisms.

▯ Note

You can customize the default convention regarding where views are located within the app by using a custom `IViewLocationExpander`.

詼 Tip

View names may be case sensitive depending on the underlying file system. For compatibility across operating systems, always match case between controller and action names and associated view folders and filenames.

Passing Data to Views

You can pass data to views using several mechanisms. The most robust approach is to specify a *model* type in the view (commonly referred to as a *viewmodel*, to distinguish it from business domain model types), and then pass an instance of this type to the view from the action. We recommend you use a model or view model to pass data to a view. This allows the view to take advantage of strong type checking. You can specify a model for a view using the `@model` directive:

```
@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>
```

Once a model has been specified for a view, the instance sent to the view can be accessed in a strongly-typed manner using `@Model` as shown above. To provide an instance of the model type to the view, the controller passes it as a parameter:

```
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);
   }
```

There are no restrictions on the types that can be provided to a view as a model. We recommend passing Plain Old CLR Object (POCO) view models with little or no behavior, so that business logic can be encapsulated elsewhere in the app. An example of this approach is the *Address* viewmodel used in the example above:

```
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; }
       }
   }
```

 Note

Nothing prevents you from using the same classes as your business model types and your display model types. However, keeping them separate allows your views to vary independently from your domain or persistence model, and can offer some security benefits as well (for models that users will send to the app using model binding).

Loosely Typed Data

In addition to strongly typed views, all views have access to a loosely typed collection of data. This same collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewBag` property is a wrapper around `ViewData` that provides a dynamic view over that collection. It is not a separate collection.

`ViewData` is a dictionary object accessed through `string` keys. You can store and retrieve objects in it, and you'll need to cast them to a specific type when you extract them. You can use `ViewData` to pass data from a controller to views, as well as within views (and partial views and layouts). String data can be stored and used directly, without the need for a cast.

Set some values for `ViewData` in an action:

```
public IActionResult SomeAction()
   {
       ViewData["Greeting"] = "Hello";
       ViewData["Address"]  = new Address()
       {
           Name = "Steve",
           Street = "123 Main St",
           City = "Hudson",
           State = "OH",
           PostalCode = "44236"
       };

       return View();
   }
```

Work with the data in a view:

```
@{
       // Requires cast
       var address = ViewData["Address"] as Address;
   }

   @ViewData["Greeting"] World!

   <address>
       @address.Name<br />
       @address.Street<br />
       @address.City, @address.State @address.PostalCode
   </address>
```

The `ViewBag` objects provides dynamic access to the objects stored in `ViewData`. This can be more convenient to work with, since it doesn't require casting. The same example as above, using `ViewBag` instead of a strongly typed `address` instance in the view:

```
@ViewBag.Greeting World!

   <address>
       @ViewBag.Address.Name<br />
       @ViewBag.Address.Street<br />
       @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
   </address>
```

Note

Since both refer to the same underlying `ViewData` collection, you can mix and match between `ViewData` and `ViewBag` when reading and writing values, if convenient.

Dynamic Views

Views that do not declare a model type but have a model instance passed to them can reference this instance dynamically. For example, if an instance of `Address` is passed to a view that doesn't declare an `@model`, the view would still be able to refer to the instance's properties dynamically as shown:

```
<address>
       @Model.Street<br />
       @Model.City, @Model.State @Model.PostalCode<br />
       <abbr title="Phone">P:</abbr>
       425.555.0100
   </address>
```

This feature can offer some flexibility, but does not offer any compilation protection or IntelliSense. If the property doesn't exist, the page will fail at runtime.

More View Features

Tag helpers make it easy to add server-side behavior to existing HTML tags, avoiding the need to use custom code or helpers within views. Tag helpers are applied as attributes to HTML elements, which are ignored by editors that aren't familiar with them, allowing view markup to be edited and rendered in a variety of tools. Tag helpers have many uses, and in particular can make working with forms much easier.

Generating custom HTML markup can be achieved with many built-in HTML Helpers, and more complex UI logic (potentially with its own data requirements) can be encapsulated in View Components. View components provide the same separation of concerns that controllers and views offer, and can eliminate the need for actions and views to deal with data used by common UI elements.

Like many other aspects of ASP.NET Core, views support dependency injection, allowing services to be injected into views.

Razor syntax

By Taylor Mullen and Rick Anderson

What is Razor?

Razor is a markup syntax for embedding server based code into web pages. The Razor syntax consists of Razor markup, C# and HTML. Files containing Razor generally have a *.cshtml* file extension.

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor is no different than in an HTML file. A Razor file with the following markup:

```
<p>Hello World</p>
```

Is rendered unchanged as `<p>Hello World</p>` by the server.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output. Razor can transition from HTML into C# or into Razor specific markup. When an `@` symbol is followed by a Razor reserved keyword it transitions into Razor specific markup, otherwise it transitions into plain C# .

HTML containing `@` symbols may need to be escaped with a second `@` symbol. For example:

```
<p>@@Username</p>
```

would render the following HTML:

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don’t treat the `@` symbol as a transition character.

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code. For example:

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword implicit expressions must not contain spaces. For example, you can intermingle spaces as long as the C# statement has a clear ending:

```
<p>@await DoSomething("hello", "world")</p>
```

Explicit Razor expressions

Explicit Razor expressions consists of an @ symbol with balanced parenthesis. For example, to render last weeks’ time:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the @() parenthesis is evaluated and rendered to the output.

Implicit expressions generally cannot contain spaces. For example, in the code below, one week is not subtracted from the current time:

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

Which renders the following HTML:

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

You can use an explicit expression to concatenate text with an expression result:

```
@{
    var joe = new Person("Joe", 33);
 }

<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` would be treated as an email address and `<p>Age@joe.Age</p>` would be rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through *IHtmlContent.WriteTo*. C# expressions that don't evaluate to *IHtmlContent* are converted to a string (by *ToString*) and encoded before they are rendered. For example, the following Razor markup:

```
@("<span>Hello World</span>")
```

Renders this HTML:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

Which the browser renders as:

`<span>Hello World</span>`

`HtmlHelper` `Raw` output is not encoded but rendered as HTML markup.

> **Warning**
>
> Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult, avoid using `HtmlHelper.Raw` on user input.

The following Razor markup:

```
@Html.Raw("<span>Hello World</span>")
```

Renders this HTML:

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks is not rendered. Code blocks and expressions in a Razor page share the same scope and are defined in order (that is, declarations in a code block will be in scope for later code blocks and expressions).

```
@{
    var output = "Hello World";
}

<p>The rendered result: @output</p>
```

Would render:

```
<p>The rendered result: Hello World</p>
```

Implicit transitions

The default language in a code block is C#, but you can transition back to HTML. HTML within a code block will transition back into rendering HTML:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a sub-section of a code block that should render HTML, surround the characters to be rendered with the Razor `<text>` tag:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

You generally use this approach when you want to render HTML that is not surrounded by an HTML tag. Without an HTML or Razor tag, you get a Razor runtime error.

Explicit Line Transition with `@:`

To render the rest of an entire line as HTML inside a code block, use the `@:` syntax:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the `@:` in the code above, you'd get a Razor run time error.

Control Structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures.

Conditionals `@if`, `else if`, `else` and `@switch`

The `@if` family controls when code runs:

```
@if (value % 2 == 0)
{
    <p>The value was even</p>
}
```

`else` and `else if` don't require the `@` symbol:

```
@if (value % 2 == 0)
{
    <p>The value was even</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value was not large and is odd.</p>
}
```

You can use a switch statement like this:

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number was not 1 or 1337.</p>
        break;
}
```

Looping `@for`, `@foreach`, `@while`, and `@do while`

You can render templated HTML with looping control statements. For example, to render a list of people:

```
@{
    var people = new Person[]
    {
          new Person("John", 33),
          new Person("Doe", 41),
    };
}
```

You can use any of the following looping statements:

`@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);
```

Compound `@using`

In C# a using statement is used to ensure an object is disposed. In Razor this same mechanism can be used to create HTML helpers that contain additional content. For instance, we can utilize HTML Helpers to render a form tag with the `@using` statement:

```
@using (Html.BeginForm())
{
    <div>
        email:
        <input type="email" id="Email" name="Email" value="" />
        <button type="submit"> Register </button>
    </div>
}
```

You can also perform scope level actions like the above with Tag Helpers.

`@try`, `catch`, `finally`

Exception handling is similar to C#:

```
@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 has the capability to protect critical sections with lock statements:

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments. The following markup:

```
@{
    /* C# comment. */
    // Another C# comment.
}
<!-- HTML comment -->
```

Is rendered by the server as:

```
<!-- HTML comment -->
```

Razor comments are removed by the server before the page is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server will not render any markup:

```
 @*
 @{
     /* C# comment. */
     // Another C# comment.
 }
 <!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive will typically change the way a page is parsed or enable different functionality within your Razor page.

Understanding how Razor generates code for a view will make it easier to understand how directives work. A Razor page is used to generate a C# file. For example, this Razor page:

```
@{
    var output = "Hello World";
}

<div>Output: @output</div>
```

Generates a class similar to the following:

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Hello World";

        WriteLiteral("/r/n<div>Output: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Viewing the Razor C# class generated for a view explains how to view this generated class.

`@using`

The `@using` directive will add the c# `using` directive to the generated razor page:

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

`@model`

The `@model` directive allows you to specify the type of the model passed to your Razor page. It uses the following syntax:

```
@model TypeNameOfModel
```

For example, if you create an ASP.NET Core MVC app with individual user accounts, the *Views/Account/Login.cshtml* Razor view contains the following model declaration:

```
@model LoginViewModel
```

In the class example in , the class generated inherits from `RazorPage<dynamic>`. By adding an `@model` you control what's inherited. For example

```
@model LoginViewModel
```

Generates the following class

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor pages expose a `Model` property for accessing the model passed to the page.

```
<div>The Login Email: @Model.Email</div>
```

The `@model` directive specified the type of this property (by specifying the `T` in `RazorPage<T>` that the generated class for your page derives from). If you don't specify the `@model` directive the `Model` property will be of type `dynamic`. The value of the model is passed from the controller to the view. See Strongly typed models and the @model keyword for more information.

`@inherits`

The `@inherits` directive gives you full control of the class your Razor page inherits:

```
@inherits TypeNameOfClassToInheritFrom
```

For instance, let's say we had the following custom Razor page type:

```
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } = "Hello World.";
}
```

The following Razor would generate `<div>Custom text: Hello World</div>`.

```
@inherits CustomRazorPage<TModel>

<div>Custom text: @CustomText</div>
```

You can't use `@model` and `@inherits` on the same page. You can have `@inherits` in a *_ViewImports.cshtml* file that the Razor page imports. For example, if your Razor view imported the following *_ViewImports.cshtml* file:

```
@inherits CustomRazorPage<TModel>
```

The following strongly typed Razor page

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

Generates this HTML markup:

```
<div>The Login Email: Rick@contoso.com</div>
<div>Custom text: Hello World</div>
```

When passed "Rick@contoso.com" in the model:

See Layout for more information.

`@inject`

The `@inject` directive enables you to inject a service from your service container into your Razor page for use. See Dependency injection into views.

`@functions`

The `@functions` directive enables you to add function level content to your Razor page. The syntax is:

```
@functions { // C# Code }
```

For example:

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

Generates the following HTML markup:

```
<div>From method: Hello</div>
```

The generated Razor C# looks like:

```
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`

The `@section` directive is used in conjunction with the layout page to enable views to render content in different parts of the rendered HTML page. See Sections for more information.

TagHelpers

The following Tag Helpers directives are detailed in the links provided.

@addTagHelper
@removeTagHelper
@tagHelperPrefix

Razor reserved keywords

Razor keywords

functions
inherits
model
section
helper (Not supported by ASP.NET Core.)

Razor keywords can be escaped with `@(Razor Keyword)`, for example `@(functions)`. See the complete sample below.

C# Razor keywords

case
do
default
for
foreach
if
lock
switch
try
using
while

C# Razor keywords need to be double escaped with `@(@C# Razor Keyword)`, for example `@(@case)`. The first `@` escapes the Razor parser, the second `@` escapes the C# parser. See the complete sample below.

Reserved keywords not used by Razor

namespace
class

Viewing the Razor C# class generated for a view

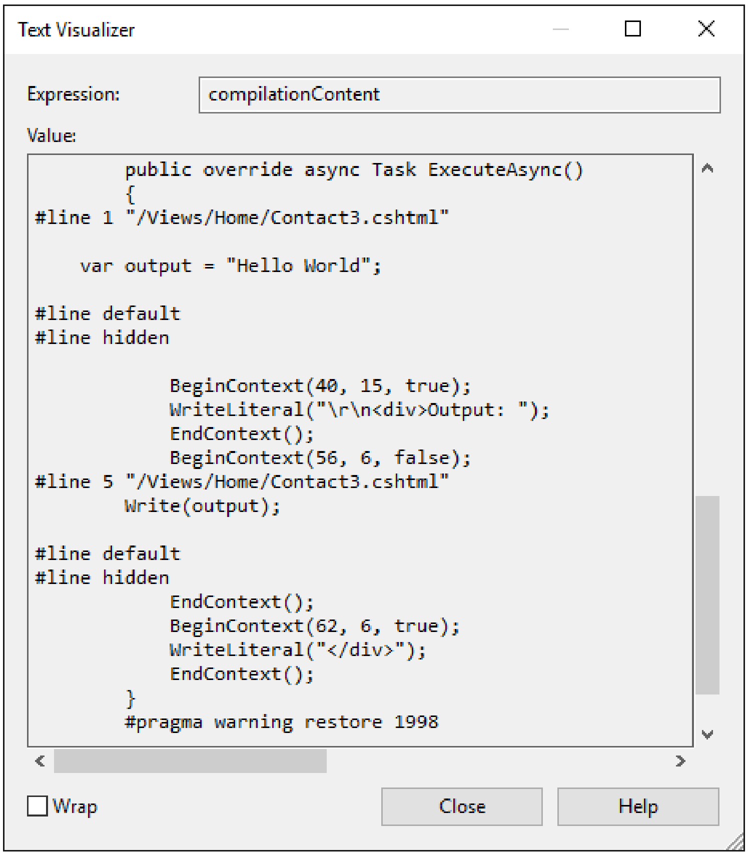
Add the following class to your ASP.NET Core MVC project:

```
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Razor;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.Internal;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

public class CustomCompilationService : DefaultRoslynCompilationService, ICompilationService
{
    public CustomCompilationService(ApplicationPartManager partManager,
        IOptions<RazorViewEngineOptions> optionsAccessor,
        IRazorViewEngineFileProviderAccessor fileProviderAccessor,
        ILoggerFactory loggerFactory)
        : base(partManager, optionsAccessor, fileProviderAccessor, loggerFactory)
    {
    }

    CompilationResult ICompilationService.Compile(RelativeFileInfo fileInfo,
        string compilationContent)
    {
        return base.Compile(fileInfo, compilationContent);
    }
}
```

Override the `ICompilationService` added by MVC with the above class;

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddSingleton<ICompilationService, CustomCompilationService>();
}
```

Set a break point on the `Compile` method of `CustomCompilationService` and view `compilationContent`.



Note

View lookups are case sensitive. If your controller routing seeks a view named `Index` (*Index.cshtml*) but you've named your view file `index` (*index.cshtml*), you'll receive an exception: `InvalidOperationException: The view 'Index' was not found.`

Razor view compilation and precompilation in ASP.NET Core

By Rick Anderson

Razor views are compiled at runtime when the view is invoked. ASP.NET Core 1.1.0 and higher can optionally compile Razor views and deploy them with the app (precompilation). To deploy precompiled views, set `MvcRazorCompileOnPublish` to true and include a package reference to `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation`. The following *.csproj* sample highlights these settings:

```
<Project ToolsVersion="15.0" Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.0" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="1.1.0-*" />
  </ItemGroup>

</Project>
```

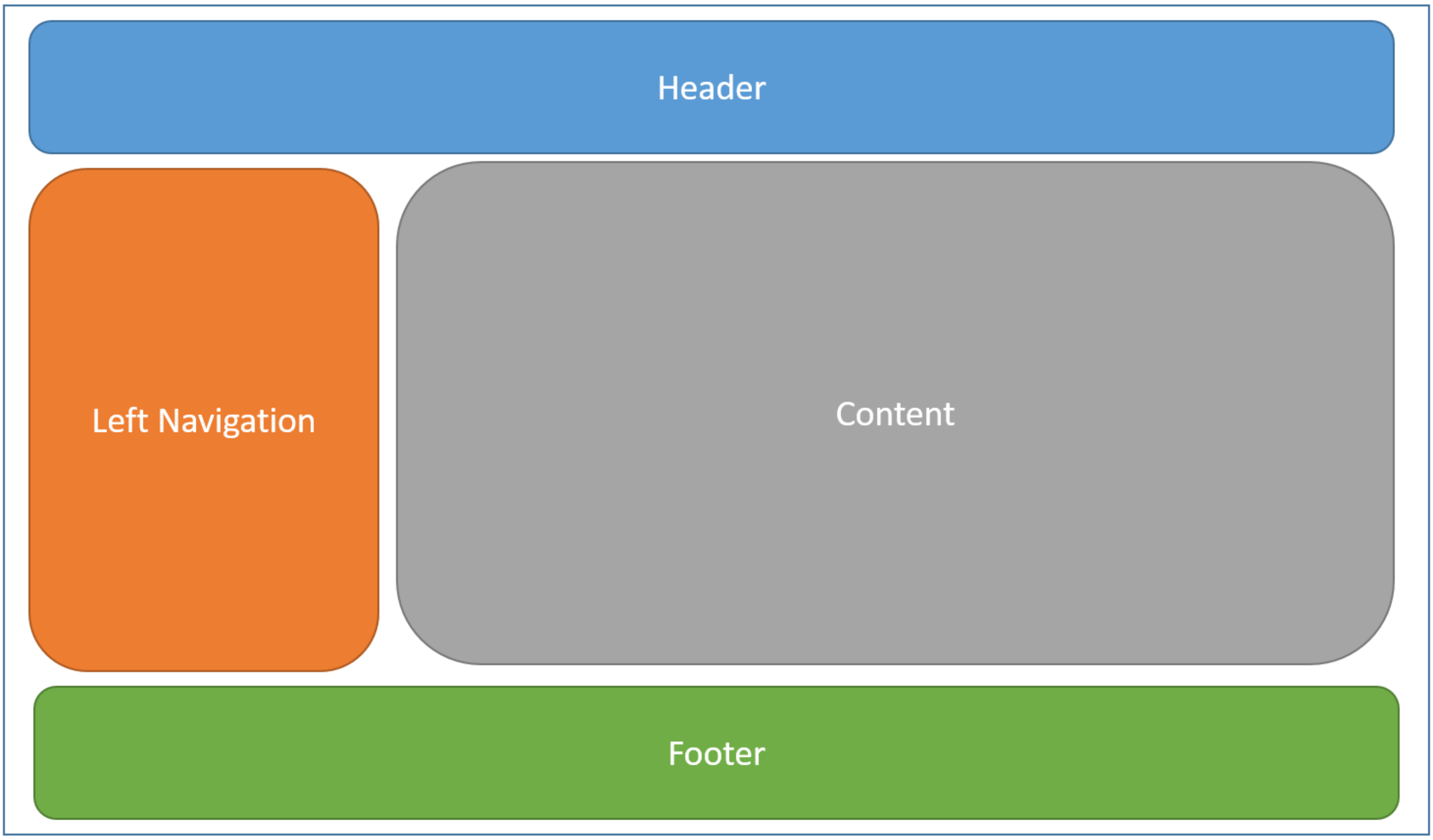
Layout

By Steve Smith

Views frequently share visual and programmatic elements. In this article, you'll learn how to use common layouts, share directives, and run common code before rendering views in your ASP.NET app.

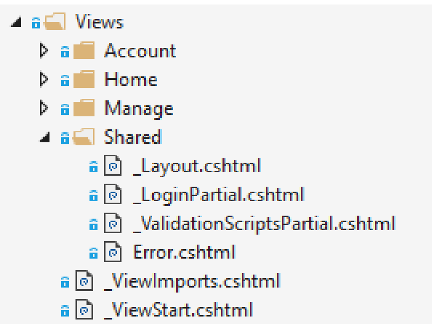
What is a Layout

Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views, helping them follow the Don't Repeat Yourself (DRY) principle.

By convention, the default layout for an ASP.NET app is named `_Layout.cshtml`. The Visual Studio ASP.NET Core MVC project template includes this layout file in the `Views/Shared` folder:



This layout defines a top level template for views in the app. Apps do not require a layout, and apps can define more than one layout, with different views specifying different layouts.

An example `_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>
```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

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

The layout specified can use a full path (example: `/Views/Shared/_Layout.cshtml`) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine will search for the layout file using its standard discovery process. The controller-associated folder is searched first, followed by the `Shared` folder. This discovery process is identical to the one used to discover partial views.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections

A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional. If a required section is not found, an exception will be thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in a view:

```
@section Scripts {
     <script type="text/javascript" src="/scripts/main.js"></script>
   }
```

In the code above, validation scripts are added to the `scripts` section on a view that includes a form. Other views in the same application might not require any additional scripts, and so wouldn't need to define a scripts section.

Sections defined in a view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections

By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views can use Razor directives to do many things, such as importing namespaces or performing dependency injection. Directives shared by many views may be specified in a common `_ViewImports.cshtml` file. The `_ViewImports` file supports the following directives:

`@addTagHelper`

`@removeTagHelper`

`@tagHelperPrefix`

`@using`

`@model`

`@inherits`

`@inject`

The file does not support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `_ViewImports.cshtml` file for an ASP.NET Core MVC app is typically placed in the `Views` folder. A `_ViewImports.cshtml` file can be placed within any folder, in which case it will only be applied to views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level, and then for each folder leading up to the location of the view itself, so settings specified at the root level may be overridden at the folder level.

For example, if a root level `_ViewImports.cshtml` file specifies `@model` and `@addTagHelper`, and another `_ViewImports.cshtml` file in the controller-associated folder of the view specifies a different `@model` and adds another `@addTagHelper`, the view will have access to both tag helpers and will use the latter `@model`.

If multiple `_ViewImports.cshtml` files are run for a view, combined behavior of the directives included in the `ViewImports.cshtml` files will be as follows:

`@addTagHelper`, `@removeTagHelper`: all run, in order

`@tagHelperPrefix`: the closest one to the view overrides any others

`@model`: the closest one to the view overrides any others

`@inherits`: the closest one to the view overrides any others

`@using`: all are included; duplicates are ignored

`@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

If you have code you need to run before every view, this should be placed in the `_ViewStart.cshtml` file. By convention, the `_ViewStart.cshtml` file is located in the `Views` folder. The statements listed in `_ViewStart.cshtml` are run before every full view (not layouts, and not partial views). Like ViewImports.cshtml, `_ViewStart.cshtml` is hierarchical. If a `_ViewStart.cshtml` file is defined in the controller-associated view folder, it will be run after the one defined in the root of the `Views` folder (if any).

A sample `_ViewStart.cshtml` file:

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

The file above specifies that all views will use the `_Layout.cshtml` layout.

▯ Note

Neither `_ViewStart.cshtml` nor `_ViewImports.cshtml` are typically placed in the `/Views/Shared` folder. The app-level versions of these files should be placed directly in the `/Views` folder.

Introduction to Tag Helpers in ASP.NET Core

By Rick Anderson

What are Tag Helpers?

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with HTML Helpers, Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. Tag Helpers compared to HTML Helpers explains the differences in more detail.

What Tag Helpers provide

An HTML-friendly development experience For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. Tag Helpers compared to HTML Helpers explains the differences in more detail. IntelliSense support for Tag Helpers explains the IntelliSense environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only available on the server For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element. For example, the `<input>` element used in many of the views in the *Views/Account* folder contains the `asp-for` attribute, which extracts the name of the specified model property into the rendered HTML. The following Razor markup:

```
<label asp-for="Email"></label>
```

Generates the following HTML:

```
<label for="Email">Email</label>
```

The `asp-for` attribute is made available by the `For` property in the `LabelTagHelper`. See Authoring Tag Helpers for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the "!" opt-out character.

`@addTagHelper` makes Tag Helpers available

If you create a new ASP.NET Core web app named *AuthoringTagHelpers* (with no authentication), the following *Views/_ViewImports.cshtml* file will be added to your project:

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

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is *Views/_ViewImports.cshtml*, which by default is inherited by all view files in the *Views* folder and sub-directories; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or sub-directory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "*, AuthoringTagHelpers"
```

If your project contains an `EmailTagHelper` with the default namespace (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), you can provide the fully qualified name (FQN) of the Tag Helper:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers"
```

To add a Tag Helper to a view using an FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the "*" wildcard syntax. The wildcard syntax allows you to insert the wildcard character "*" as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper`:

```
@addTagHelper "AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers"
@addTagHelper "AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers"
```

As mentioned previously, adding the `@addTagHelper` directive to the *Views/_ViewImports.cshtml* file makes the Tag Helper available to all view files in the *Views* directory and sub-directories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a *Views/Folder/_ViewImports.cshtml* file removes the specified Tag Helper from all of the views in *Folder*.

Controlling Tag Helper scope with the *_ViewImports.cshtml* file

You can add a *_ViewImports.cshtml* to any view folder, and the view engine adds the directives from that *_ViewImports.cshtml* file to those contained in the *Views/_ViewImports.cshtml* file. If you added an empty *Views/Home/_ViewImports.cshtml* file for the *Home* views, there would be no change because the *_ViewImports.cshtml* file is additive. Any `@addTagHelper` directives you add to the *Views/Home/_ViewImports.cshtml* file (that are not in the default *Views/_ViewImports.cshtml* file) would expose those Tag Helpers to views only in the *Home* folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `<span>` with the Tag Helper opt-out character:

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

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. In the code image below, the Tag Helper prefix is set to `th:`, so only those elements using the prefix `th:` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<input>` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `<span>` element does not.

```
<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>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

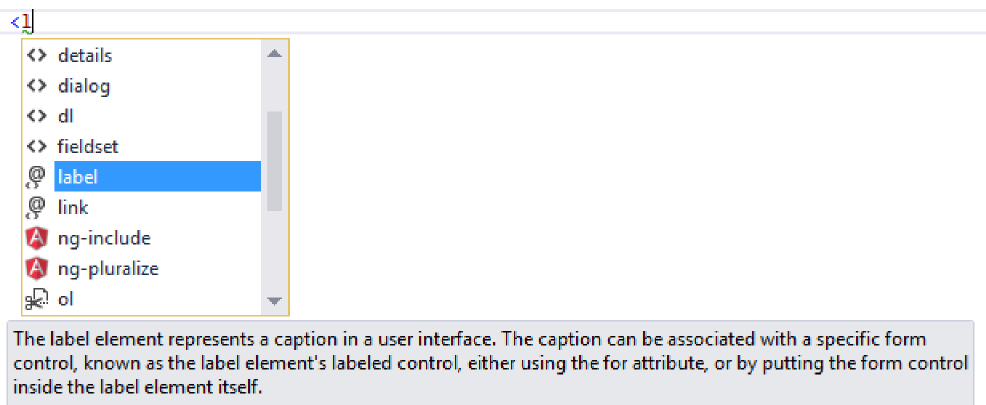
IntelliSense support for Tag Helpers

Note

You must install the Razor Language Service extension for IntelliSense to work with Tag Helpers. See ASP.NET Core Known Issues for more information.

When you create a new ASP.NET web app in Visual Studio, it adds the NuGet package "Microsoft.AspNetCore.Razor.Tools". This is the package that adds Tag Helper tooling.

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



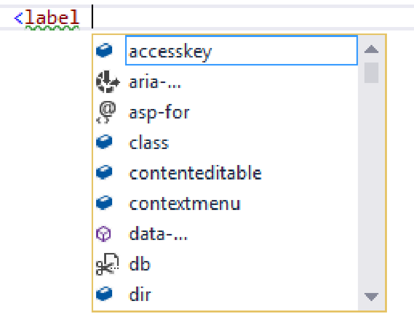
Not only do you get HTML help, but the icon (the "@" symbol with "<>" under it).

identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the "<>" icon.

A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.

```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:

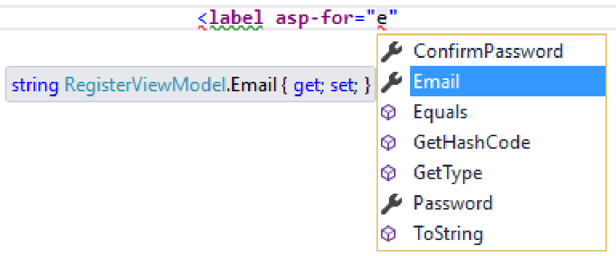


IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:

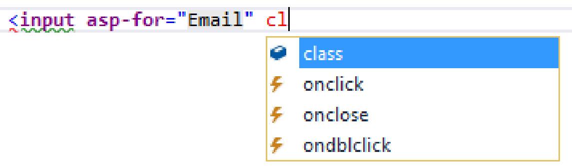
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

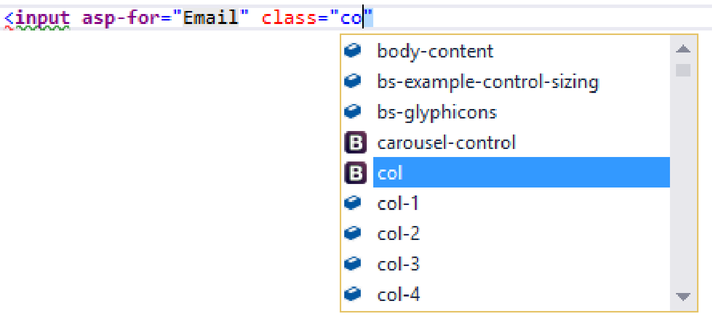
```
<label asp-for
```

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the default) inside the double quotes (""), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.



IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:





Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

The at (`@`) symbol tells Razor this is the start of code. The next two parameters ("FirstName" and "First Name:") are strings, so IntelliSense can't help. The last argument:

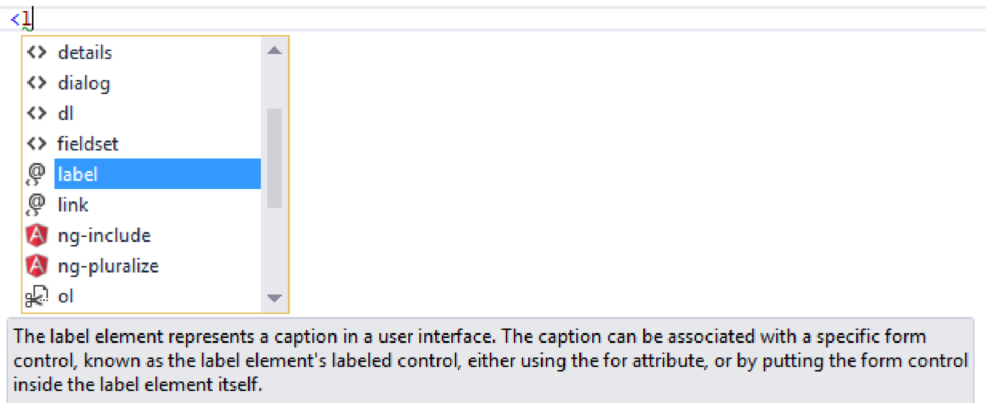
```
new {@class="caption"}
```

Is an anonymous object used to represent attributes. Because **class** is a reserved keyword in C#, you use the `@` symbol to force C# to interpret "@class=" as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

```
<label class="caption" asp-for="FirstName"></label>
```

With the Tag Helper version, as soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



IntelliSense helps you write the entire line. The `LabelTagHelper` also defaults to setting the content of the `asp-for` attribute value ("FirstName") to "First Name"; It converts camel-cased properties to a sentence composed of the property name with a space where each new upper-case letter occurs. In the following markup:

```
<label class="caption" asp-for="FirstName"></label>
```

generates:

```
<label class="caption" for="FirstName">First Name</label>
```

The camel-cased to sentence-cased content is not used if you add content to the `<label>`. For example:

```
<label class="caption" asp-for="FirstName">Name First</label>
```

generates:

```
<label class="caption" for="FirstName">Name First</label>
```

The following code image shows the Form portion of the *Views/Account/Register.cshtml* Razor view generated from the legacy ASP.NET 4.5.x MVC template included with Visual Studio 2015.

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @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>
}
```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:

```
<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>
```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

```
<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>
```

Each of the "asp-" attributes has a value of "Email", but "Email" is not a string. In this context, "Email" is the C# model expression property for the `RegisterViewModel`.

The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML Helpers approach. IntelliSense support for Tag Helpers goes into detail on working with Tag Helpers in the Visual Studio editor.

Tag Helpers compared to Web Server Controls

Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET Web Server controls are declared and invoked on a page.

Web Server controls have a non-trivial lifecycle that can make developing and debugging difficult.

Web Server controls allow you to add functionality to the client Document Object Model (DOM) elements by using a client control. Tag Helpers have no DOM.

Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.

Multiple Tag Helpers can act on the same element (see Avoiding Tag Helper conflicts) while you typically can't compose Web Server controls.

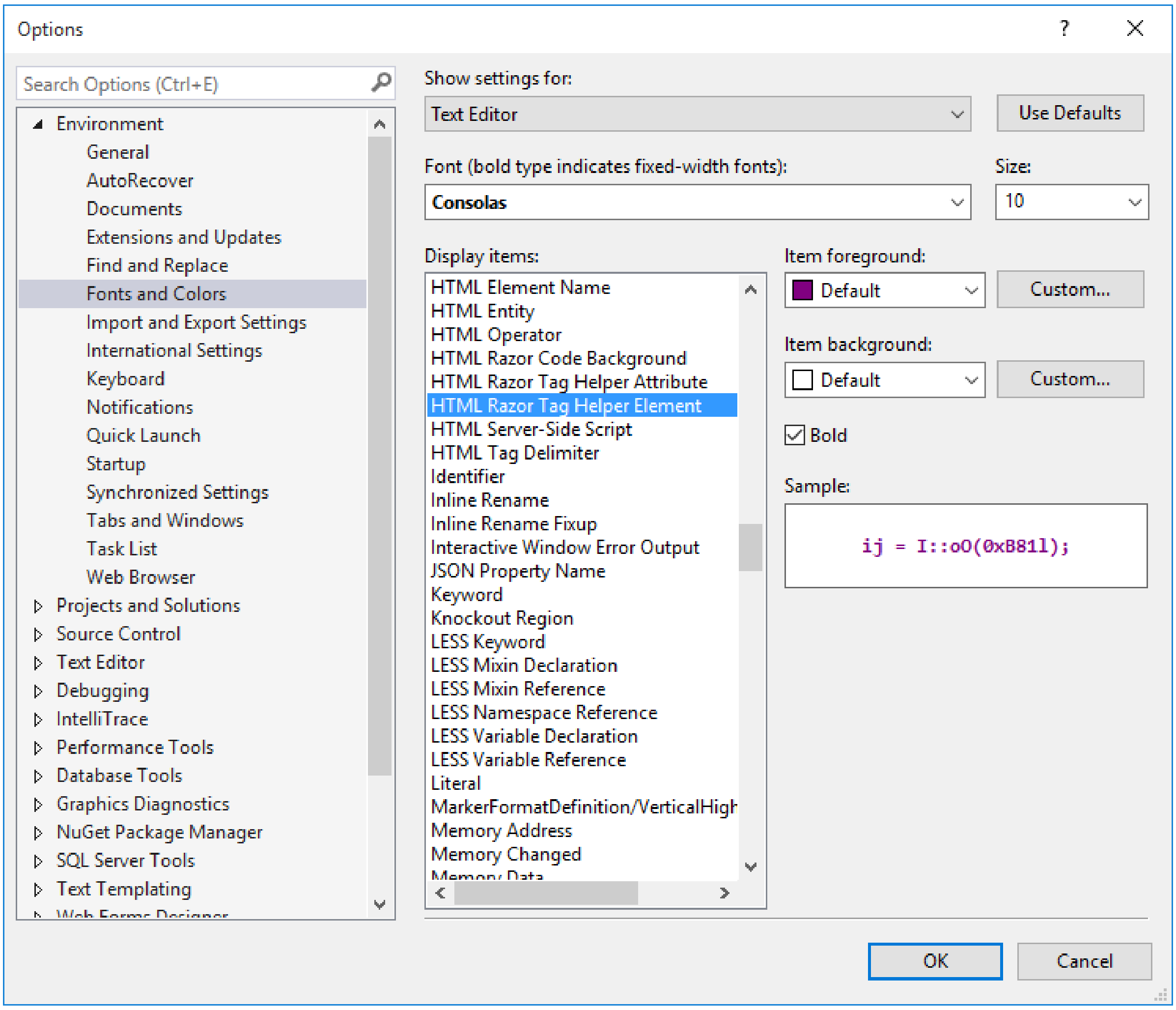
Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.

Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.

Web Server controls use System.ComponentModel to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools** > **Options** > **Environment** > **Fonts and Colors**:



Additional Resources

Authoring Tag Helpers
Working with Forms
TagHelperSamples on GitHub contains Tag Helper samples for working with Bootstrap.

Authoring Tag Helpers in ASP.NET Core, a walkthrough with samples

By Rick Anderson

View or download sample code

Getting started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. Introduction to Tag Helpers describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method. We will introduce the `TagHelper` methods and properties as we use them in this tutorial.

Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.

Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it is a reasonable convention. Now let's get started writing some simple tag helpers.

A minimal Tag Helper

In this section we will write a tag helper that updates an email tag. For example:

```
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

Add the following `EmailTagHelper` class to the *TagHelpers* folder.

```
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
        }
    }
}
```

Notes:

Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **Email**TagHelper is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.

The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.

The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.

The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.

The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.

Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```
public class Email : TagHelper
```

To make the `EmailTagHelper` class available to all our Razor views, we will add the `addTagHelper` directive to the *Views/_ViewImports.cshtml* file:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "*, AuthoringTagHelpers"
```

The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (we are using "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET Core MVC tag helpers using the wildcard syntax (those helpers are discussed in Introduction to Tag Helpers.) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper "AuthoringTagHelpers.TagHelpers3.EmailTagHelper, AuthoringTagHelpers"
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers will prefer to use the wildcard syntax. Introduction to Tag Helpers goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

Update the markup in the *Views/Home/Contact.cshtml* file with these changes:

```
@{
    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>
```

Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support</a>`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

Note: Like HTML tags and attributes, tags, class names and attributes in Razor, and C# are not case-sensitive.

SetAttribute and SetContent

In this section, we'll update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update it to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

```
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);
    }
}
```

Notes:

Pascal-cased class and property names for tag helpers are translated into their lower kebab case. Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.

The last line sets the completed content for our minimally functional tag helper.

The highlighted line shows the syntax for adding attributes:

```
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);
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

Update the markup in the *Views/Home/Contact.cshtml* file with these changes:

```
@{
    ViewData["Title"] = "Contact Copy";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way Copy Version <br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email mail-to="Support"></email><br />
    <strong>Marketing:</strong><email mail-to="Marketing"></email>
</address>
```

Run the app and verify that it generates the correct links.

Note

If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a start tag (`<email mail-to="Rick">`) you must decorate the class with the following:

```
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

With a self-closing email tag helper, the output would be `<a href="mailto:Rick@contoso.com" />`. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that is self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

ProcessAsync

In this section we'll write an asynchronous email helper.

Replace the `EmailTagHelper` class with the following code:

```
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);
    }
}
```

Notes:

This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.

We use the `output` parameter to get contents of the HTML element.

Make the following change to the *Views/Home/Contact.cshtml* file so the tag helper can get the target email.

```
@{
    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>
```

Run the app and verify that it generates valid email links.

RemoveAll, PreContent.SetHtmlContent and PostContent.SetHtmlContent

Add the following `BoldTagHelper` class to the *TagHelpers* folder.

```
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>");
        }
    }
}

/*
 * public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";
    return View("AboutBoldOnly");
    // return View();
}
*/
```

Notes:

The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.

Because we don't want to replace the existing tag content, we must write the opening `<strong>` tag with the `PreContent.SetHtmlContent` method and the closing `</strong>` tag with the `PostContent.SetHtmlContent` method.

Modify the *About.cshtml* view to contain a `bold` attribute value. The completed code is shown below.

```
@{
    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>
```

Run the app. You can use your favorite browser to inspect the source and verify the markup.

The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of "bold". The `<bold>` element was not modified by the tag helper.

Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **Bold**TagHelper to `<bold>` tags.

Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

```
[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>");
    }
}
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named "bold" with an attribute named "bold" ( ) to match.

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the `BoldTagHelper` to target `<MyBold>` tags, you would use the following attribute:

```
[HtmlTargetElement("MyBold")]
```

Passing a model to a Tag Helper

Add a *Models* folder.

Add the following `WebsiteContext` class to the *Models* folder:

```
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; }
    }
}
```

Add the following `WebsiteInformationTagHelper` class to the *TagHelpers* folder.

```
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;
      }
   }
}
```

Notes:

As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into lower kebab case. Therefore, to use the `WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.

We are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

```
[HtmlTargetElement("WebsiteInformation")]
```

The lower kebab case tag `<website-information />` would not match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

```
[HtmlTargetElement("Website-Information")]
```

Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a section element (which is not self-closing and we are writing content inside the `section` element). Therefore, we need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)

The `$` (dollar sign) in the following line uses an interpolated string:

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

Add the following markup to the *About.cshtml* view. The highlighted markup displays the web site information.

```
@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 }" />
```

 Note

In the Razor markup shown below:

```
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" />
```

Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

Run the app, and navigate to the About view to see the web site information.

 Note

You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:

```
<website-information info="new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 }" >
</website-information>
```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

```
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();
            }
        }
    }
}
```

Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
@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>
```

Replace the `Index` method in the `Home` controller with the following code:

```
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

Run the app and browse to the home page. The markup in the conditional `div` will not be rendered. Append the query string `?approved=true` to the URL (for example, `http://localhost:1235/Home/Index?approved=true`). `approved` is set to true and the conditional markup will be displayed.

Note

We use the nameof operator to specify the attribute to target rather than specifying a string as we did with the bold tag helper:

```
[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();
      }
   }
}
```

The nameof operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoiding Tag Helper conflicts

In this section, we will write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and we may refactor them in the future, we'll keep them in the same file.

Add the following `AutoLinkerHttpTagHelper` class to the *TagHelpers* folder.

```
[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}
    }
}
```

Note

The `AutoLinkerHttpTagHelper` class targets `p` elements and uses Regex to create the anchor.

Add the following markup to the end of the *Views/Home/Contact.cshtml* file:

```
@{
    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>
```

Run the app and verify that the tag helper renders the anchor correctly.

Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert www text to an anchor tag that also contains the original www text. The updated code is highlighted below:

```
    [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
        }
    }
}
```

Run the app. Notice the www text is rendered as a link but the HTTP text is not. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. Later in the tutorial we'll see how to control the order that tag helpers run in. We'll fix the code with the following:

```
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
    }
}
```

Note

In the first edition of the auto-linking tag helpers, we got the content of the target with the following code:

```
var childContent = await output.GetChildContentAsync();
```

That is, we call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. We fixed that problem with the following code:

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links will not be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

```
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;   }
    }
```

The above code will guarantee that the HTTP tag helper runs before the WWW tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspecting and retrieving child content

The tag-helpers provide several properties to retrieve content.

The result of `GetChildContentAsync` can be appended to `output.Content`.
You can inspect the result of `GetChildContentAsync` with `GetContent`.
If you modify `output.Content`, the TagHelper body will not be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

```
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}
    }
}
```

Multiple calls to `GetChildContentAsync` will return the same value and will not re-execute the `TagHelper` body unless you pass in a false parameter indicating not use the cached result.

Introduction to using tag helpers in forms in ASP.NET Core

By Rick Anderson, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers do not replace HTML Helpers and there is not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it is mentioned.

The Form Tag Helper

The Form Tag Helper:

Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route

Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)

Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.

Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
     <!-- Input and Submit elements -->
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>" />
    </form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is very difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

Note

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>" />
```

The Input Tag Helper:

Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`, that is the attribute value literally is part of an expression. The name of the expression is what's used for the `asp-for` attribute value.

Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property

Will not overwrite the HTML `type` attribute value when one is specified

Generates HTML5 validation attributes from data annotation attributes applied to model properties

Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.

Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
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?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte	type="number"

.NET TYPE	INPUT TYPE
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[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"

Sample:

```
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>
```

The code above generates the following 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 e-mail 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>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it is displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` will not. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper does not. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
       var joe = "Joe";
   }
   <input asp-for="@joe" />
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe" />
```

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
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; }
}
```

In the view, we bind to `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>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="" />
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
   {
       ViewData["Index"] = colorIndex;
       return View(GetPerson(id));
   }
```

The following Razor shows how you access a specific `Color` element:

```
@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>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@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>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@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>
*@
```

Note

Always use `for` (and *not* `foreach`) to iterate over a list. Evaluating an indexer in a LINQ expression can be expensive and should be minimized.

Note

The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.

Provides strong typing.

HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
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>
```

The following HTML is generated:

```
<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
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   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>
```

The Label Tag Helper

Generates the label caption and `for` attribute on a element for an expression name

HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

Less markup in source code

Strong typing with the model property.

Sample:

```
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>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.

Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.

HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

Note

You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

Targets `<div>` elements with the `asp-validation-summary` attribute

HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model is decorated with `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
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>
```

The generated HTML (when the model is valid):

```
<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 e-mail 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>
```

The Select Tag Helper

Generates select and associated option elements for properties of your model.

Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
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"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[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);
}
```

The `Index` view:

```
@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>
```

Which generates the following HTML (with "CA" selected):

```
<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>
```

Note

We do not recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    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
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@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>
```

You can decorate your enumerator list with the `Display` attribute to get a richer 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
    }
}
```

The following HTML is generated:

```
<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>
```

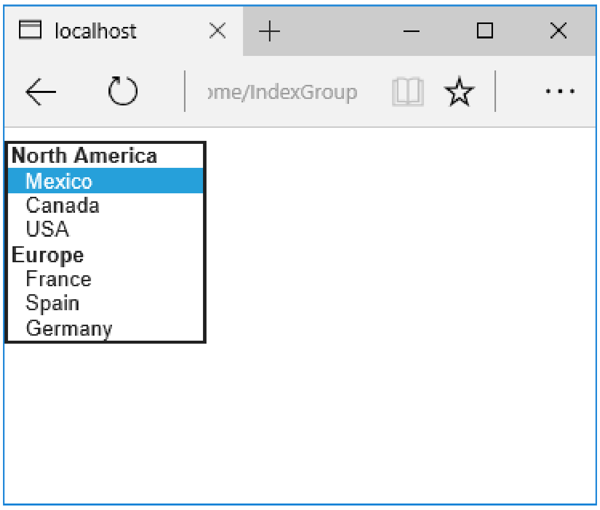
Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

```
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; }
```

The two groups are shown below:



The generated 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>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        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"}
         };
    }
}
```

With the following view:

```
@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>
```

Generates the following 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>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements is not limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
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>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
 <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>
```

Additional Resources

Tag Helpers

HTML Form element

Request Verification Token

Model Binding

Model Validation

data annotations

Code snippets for this document.

Partial Views

By Steve Smith, Maher JENDOUBI, and Rick Anderson

ASP.NET Core MVC supports partial views, which are useful when you have reusable parts of web pages you want to share between different views.

View or download sample code

What are Partial Views?

A partial view is a view that is rendered within another view. The HTML output generated by executing the partial view is rendered into the calling (or parent) view. Like views, partial views use the *.cshtml* file extension.

When Should I Use Partial Views?

Partial views are an effective way of breaking up large views into smaller components. They can reduce duplication of view content and allow view elements to be reused. Common layout elements should be specified in _Layout.cshtml. Non-layout reusable content can be encapsulated into partial views.

If you have a complex page made up of several logical pieces, it can be helpful to work with each piece as its own partial view. Each piece of the page can be viewed in isolation from the rest of the page, and the view for the page itself becomes much simpler since it only contains the overall page structure and calls to render the partial views.

Tip: Follow the Don't Repeat Yourself Principle in your views.

Declaring Partial Views

Partial views are created like any other view: you create a *.cshtml* file within the *Views* folder. There is no semantic difference between a partial view and a regular view - they are just rendered differently. You can have a view that is returned directly from a controller's `ViewResult`, and the same view can be used as a partial view. The main difference between how a view and a partial view are rendered is that partial views do not run *_ViewStart.cshtml* (while views do - learn more about *_ViewStart.cshtml* in Layout).

Referencing a Partial View

From within a view page, there are several ways in which you can render a partial view. The simplest is to use `Html.Partial`, which returns an `IHtmlString` and can be referenced by prefixing the call with `@`:

```
@Html.Partial("AuthorPartial")
```

The `PartialAsync` method is available for partial views containing asynchronous code (although code in views is generally discouraged):

```
@await Html.PartialAsync("AuthorPartial")
```

You can render a partial view with `RenderPartial`. This method doesn't return a result; it streams the rendered output directly to the response. Because it doesn't return a result, it must be called within a Razor code block (you can also call `RenderPartialAsync` if necessary):

```
@{
    Html.RenderPartial("AuthorPartial");
}
```

Because it streams the result directly, `RenderPartial` and `RenderPartialAsync` may perform better in some scenarios. However, in most cases it's recommended you use `Partial` and `PartialAsync`.

▯ Note

If your views need to execute code, the recommended pattern is to use a view component instead of a partial view.

Partial View Discovery

When referencing a partial view, you can refer to its location in several ways:

```
// Uses a view in current folder with this name
// If none is found, searches the Shared folder
@Html.Partial("ViewName")

// A view with this name must be in the same folder
@Html.Partial("ViewName.cshtml")

// Locate the view based on the application root
// Paths that start with "/" or "~/" refer to the application root
@Html.Partial("~/Views/Folder/ViewName.cshtml")
@Html.Partial("/Views/Folder/ViewName.cshtml")

// Locate the view using relative paths
@Html.Partial("../Account/LoginPartial.cshtml")
```

You can have different partial views with the same name in different view folders. When referencing the views by name (without file extension), views in each folder will use the partial view in the same folder with them. You can also specify a default partial view to use, placing it in the *Shared* folder. The shared partial view will be used by any views that don't have their own version of the partial view. You can have a default partial view (in *Shared*), which is overridden by a partial view with the same name in the same folder as the parent view.

Partial views can be *chained*. That is, a partial view can call another partial view (as long as you don't create a loop). Within each view or partial view, relative paths are always relative to that view, not the root or parent view.

▯ Note

If you declare a Razor `section` in a partial view, it will not be visible to its parent(s); it will be limited to the partial view.

Accessing Data From Partial Views

When a partial view is instantiated, it gets a copy of the parent view's `ViewData` dictionary. Updates made to the data within the partial view are not persisted to the parent view. `ViewData` changed in a partial view is lost when the partial view returns.

You can pass an instance of `ViewDataDictionary` to the partial view:

```
@Html.Partial("PartialName", customViewData)
```

You can also pass a model into a partial view. This can be the page's view model, or some portion of it, or a custom object. You can pass a model to `Partial`,`PartialAsync`, `RenderPartial`, or `RenderPartialAsync`:

```
@Html.Partial("PartialName", viewModel)
```

You can pass an instance of `ViewDataDictionary` and a view model to a partial view:

```
@Html.Partial("ArticleSection", section,
            new ViewDataDictionary(this.ViewData) { { "index", index } })
```

The markup below shows the *Views/Articles/Read.cshtml* view which contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. You can pass new `ViewData` dictionary while retaining the existing `ViewData` if you use the constructor overload of the `ViewDataDictionary` highlighted below:

```
@using Microsoft.AspNetCore.Mvc.ViewFeatures
@using PartialViewsSample.ViewModels
@model Article

<h2>@Model.Title</h2>
@*Pass the authors name to Views\Shared\AuthorPartial.cshtml*@
@Html.Partial("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)
    {
        @Html.Partial("ArticleSection", section,
                    new ViewDataDictionary(this.ViewData) { { "index", index } })
        index++;
    }
}
```

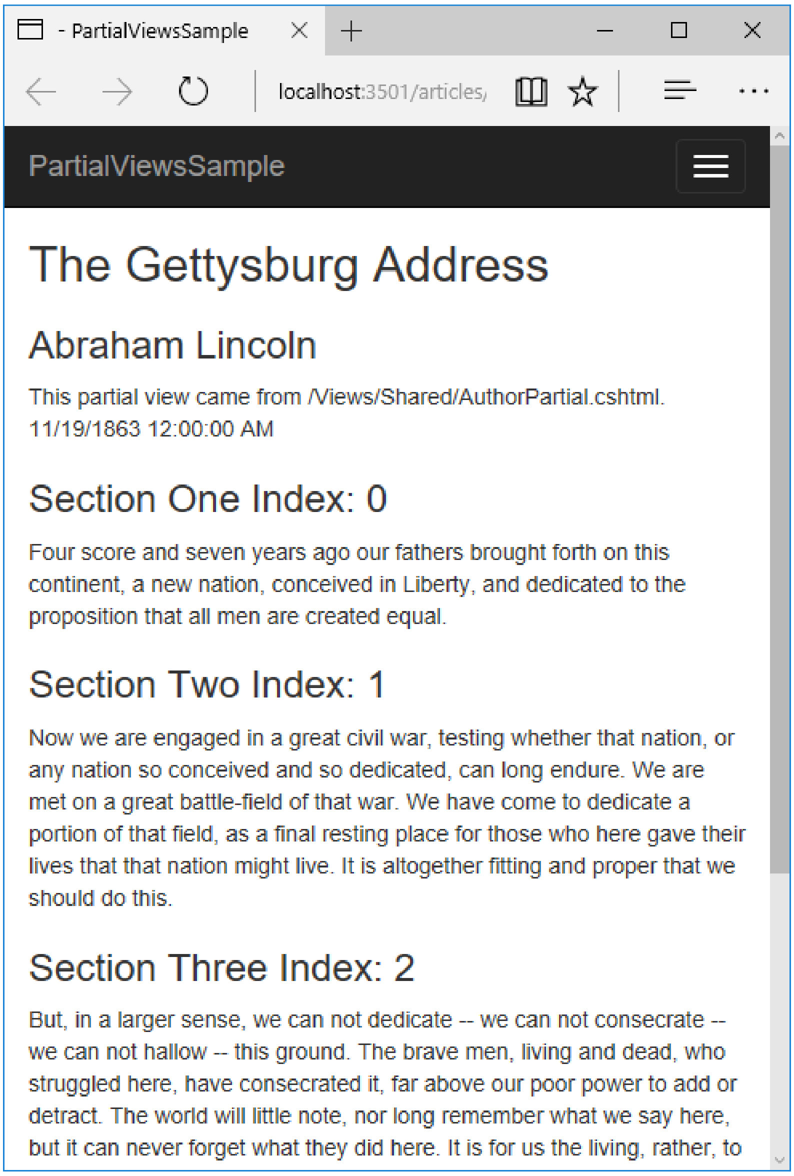
Views/Shared/AuthorPartial:

```
@model string
<div>
    <h3>@Model</h3>
    This partial view came from /Views/Shared/AuthorPartial.cshtml.<br />
</div>
```

The *ArticleSection* partial:

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title  Index: @ViewData["index"]  </h3>
<div>
    @Model.Content
</div>
```

At runtime, the partials are rendered into the parent view, which itself is rendered within the shared *_Layout.cshtml*


- PartialViewsSample
localhost:3501/articles/
PartialViewsSample
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

Dependency injection into views

By Steve Smith

ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. You should try to maintain separation of concerns between your controllers and views. Most of the data your views display should be passed in from the controller.

View or download sample code

A Simple Example

You can inject a service into a view using the `@inject` directive. You can think of `@inject` as adding a property to your view, and populating the property using DI.

The syntax for `@inject`: `@inject <type> <name>`

An example of `@inject` in action:

```
@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>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in *Startup.cs*:

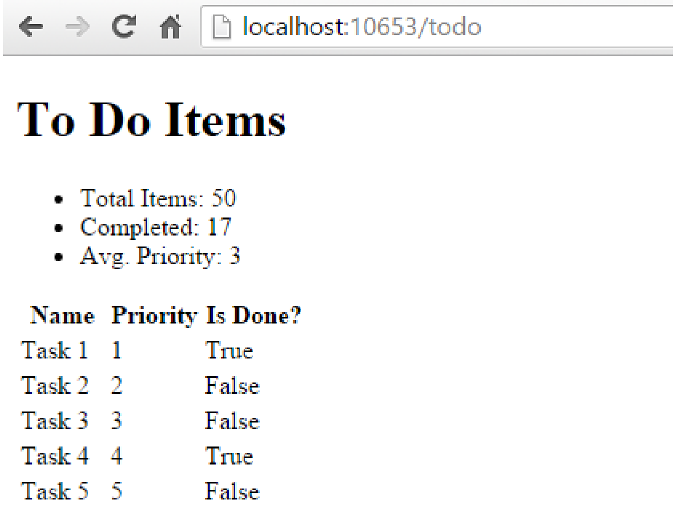
```
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
    services.AddTransient<StatisticsService>();
    services.AddTransient<ProfileOptionsService>();
```

The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

```
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);
        }
    }
}
```

The sample repository uses an in-memory collection. The implementation shown above (which operates on all of the data in memory) is not recommended for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:



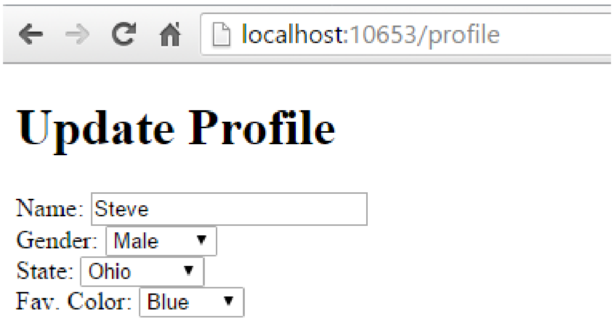
Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard MVC approach would require the controller to request data access services for each of these sets of options, and then populate a model or `ViewBag` with each set of options to be bound.

An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller, moving this view element construction logic into the view itself. The controller action to display a profile editing form only needs to pass the form the profile instance:

```
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);
        }
    }
}
```

The HTML form used to update these preferences includes dropdown lists for three of the properties:



These lists are populated by a service that has been injected into the view:

```
@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>
```

The `ProfileOptionsService` is a UI-level service designed to provide just the data needed for this form:

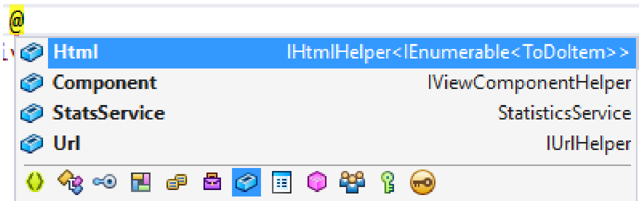
```
using System.Collections.Generic;

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" };
        }
    }
}
```

洸 Tip

Don't forget to register types you will request through dependency injection in the `ConfigureServices` method in *Startup.cs*.

Overriding Services

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



As you can see, the default fields include `Html`, `Component`, and `Url` (as well as the `StatsService` that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using `@inject`:

```
@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>
```

If you want to extend existing services, you can simply use this technique while inheriting from or wrapping the existing implementation with your own.

See Also

Simon Timms Blog: Getting Lookup Data Into Your View

View components

By Rick Anderson

View or download sample code

Introducing view components

New to ASP.NET Core MVC, view components are similar to partial views, but they are much more powerful. View components don't use model binding, and only depend on the data you provide when calling into it. A view component:

Renders a chunk rather than a whole response
Includes the same separation-of-concerns and testability benefits found between a controller and view
Can have parameters and business logic
Is typically invoked from a layout page

View components are intended anywhere you have reusable rendering logic that is too complex for a partial view, such as:

Dynamic navigation menus
Tag cloud (where it queries the database)
Login panel
Shopping cart
Recently published articles
Sidebar content on a typical blog
A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

A view component consists of two parts: the class (typically derived from ViewComponent) and the result it returns (typically a view). Like controllers, a view component can be a POCO, but most developers will want to take advantage of the methods and properties available by deriving from `ViewComponent`.

Creating a view component

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

The view component class

A view component class can be created by any of the following:

Deriving from *ViewComponent*
Decorating a class with the `[ViewComponent]` attribute, or deriving from a class with the `[ViewComponent]` attribute
Creating a class where the name ends with the suffix *ViewComponent*

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the "ViewComponent" suffix removed. It can also be explicitly specified using the `ViewComponentAttribute.Name` property.

A view component class:

Fully supports constructor dependency injection

Does not take part in the controller lifecycle, which means you can't use filters in a view component

View component methods

A view component defines its logic in an `InvokeAsync` method that returns an `IViewComponentResult`. Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the `View` method. In summary, view component methods:

Define an `InvokeAsync` method that returns an `IViewComponentResult`
Typically initializes a model and passes it to a view by calling the `ViewComponent` `View` method
Parameters come from the calling method, not HTTP, there is no model binding
Are not reachable directly as an HTTP endpoint, they are invoked from your code (usually in a view). A view component never handles a request
Are overloaded on the signature rather than any details from the current HTTP request

View search path

The runtime searches for the view in the following paths:

Views/<controller_name>/Components/<view_component_name>/<view_name>
Views/Shared/Components/<view_component_name>/<view_name>

The default view name for a view component is *Default*, which means your view file will typically be named *Default.cshtml*. You can specify a different view name when creating the view component result or when calling the `View` method.

We recommend you name the view file *Default.cshtml* and use the *Views/Shared/Components/<view_component_name>/<view_name>* path. The `PriorityList` view component used in this sample uses *Views/Shared/Components/PriorityList/Default.cshtml* for the view component view.

Invoking a view component

To use the view component, call the following inside a view:

```
@Component.InvokeAsync("Name of view component", <anonymous type containing parameters>)
```

The parameters will be passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the *Views/Todo/Index.cshtml* view file. In the following, the `InvokeAsync` method is called with two parameters:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Invoking a view component as a Tag Helper

For ASP.NET Core 1.1 and higher, you can invoke a view component as a Tag Helper:

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

Pascal-cased class and method parameters for Tag Helpers are translated into their lower kebab case. The Tag Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

Note: In order to use a View Component as a Tag Helper, you must register the assembly containing the View Component using the `@addTagHelper` directive. For example, if your View Component is in an assembly called "MyWebApp", add the following directive to the `_ViewImports.cshtml` file:

```
@addTagHelper "*, MyWebApp"
```

You can register a View Component as a Tag Helper to any file that references the View Component. See Managing Tag Helper

Scope for more information on how to register Tag Helpers.

The `InvokeAsync` method used in this tutorial:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

In Tag Helper markup:

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

In the sample above, the `PriorityList` view component becomes `priority-list`. The parameters to the view component are passed as attributes in lower kebab case.

Invoking a view component directly from a controller

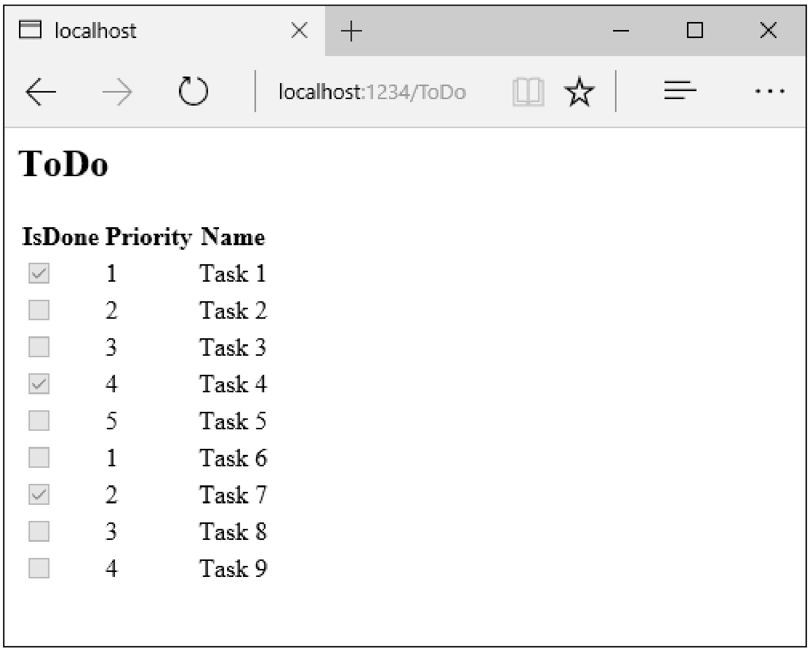
View components are typically invoked from a view, but you can invoke them directly from a controller method. While view components do not define endpoints like controllers, you can easily implement a controller action that returns the content of a `ViewComponentResult`.

In this example, the view component is called directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Walkthrough: Creating a simple view component

Download, build and test the starter code. It's a simple project with a `Todo` controller that displays a list of *Todo* items.



Add a ViewComponent class

Create a *ViewComponents* folder and add the following `PriorityListViewComponent` class:

```
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();
        }
    }
}
```

Notes on the code:

View component classes can be contained in **any** folder in the project.
Because the class name PriorityList**ViewComponent** ends with the suffix **ViewComponent**, the runtime will use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.

The `[ViewComponent]` attribute can change the name used to reference a view component. For example, we could have named the class `XYZ` and applied the `ViewComponent` attribute:

```
[ViewComponent(Name = "PriorityList")]
   public class XYZ : ViewComponent
```

The `[ViewComponent]` attribute above tells the view component selector to use the name `PriorityList` when looking for the views associated with the component, and to use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.

The component uses dependency injection to make the data context available.
`InvokeAsync` exposes a method which can be called from a view, and it can take an arbitrary number of arguments.
The `InvokeAsync` method returns the set of `ToDo` items that satisfy the `isDone` and `maxPriority` parameters.

Create the view component Razor view

Create the *Views/Shared/Components* folder. This folder **must** be named *Components*.

Create the *Views/Shared/Components/PriorityList* folder. This folder name must match the name of the view component class, or the name of the class minus the suffix (if we followed convention and used the *ViewComponent* suffix in the class name). If you used the `ViewComponent` attribute, the class name would need to match the attribute designation.

Create a *Views/Shared/Components/PriorityList/Default.cshtml* Razor view:

```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h3>Priority Items</h3>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

The Razor view takes a list of `TodoItem` and displays them. If the view component `InvokeAsync` method doesn't pass the name of the view (as in our sample), *Default* is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example *Views/Todo/Components/PriorityList/Default.cshtml)*.

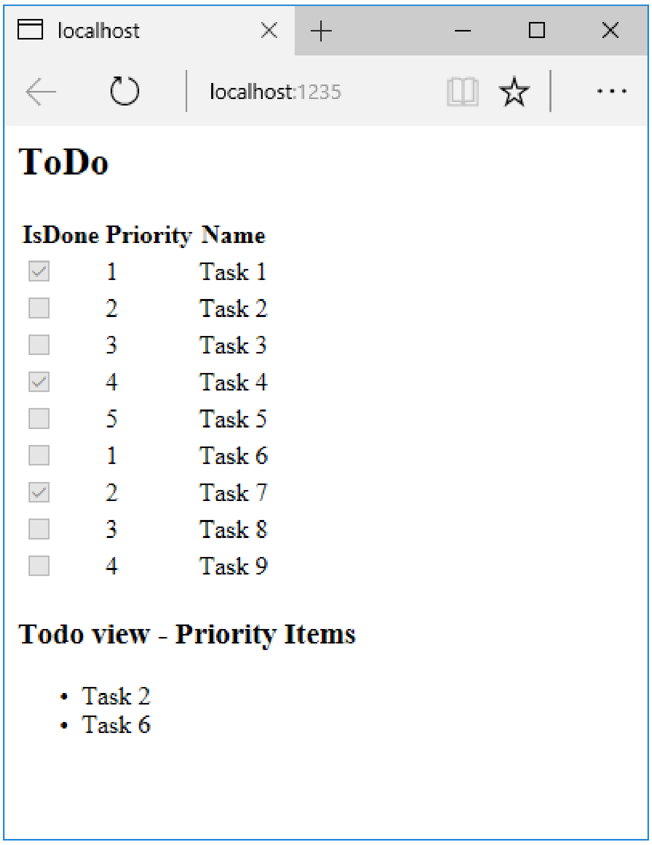
If the view component is controller-specific, you can add it to the controller-specific folder (*Views/Todo/Components/PriorityList/Default.cshtml*).

Add a `div` containing a call to the priority list component to the bottom of the *Views/Todo/index.cshtml* file:

```
</table>
<div >
    @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
</div>
```

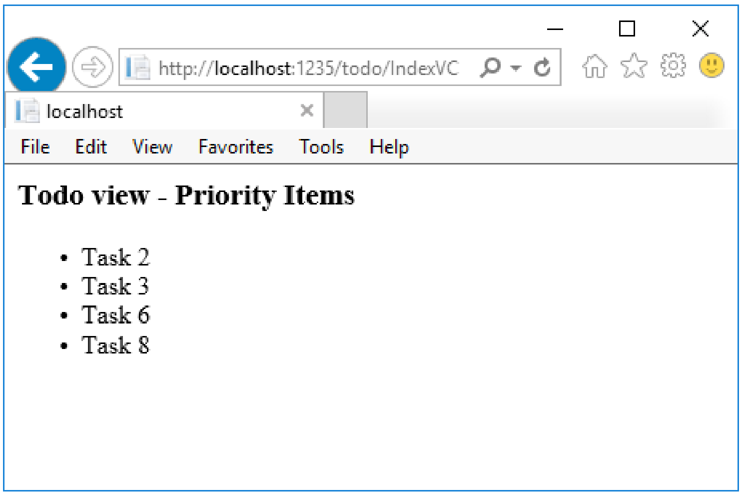
The markup `@await Component.InvokeAsync` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. `InvokeAsync` can take an arbitrary number of arguments.

Test the app. The following image shows the ToDo list and the priority items:



You can also call the view component directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```



Specifying a view name

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the `InvokeAsync` method. Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

```
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);
}
```

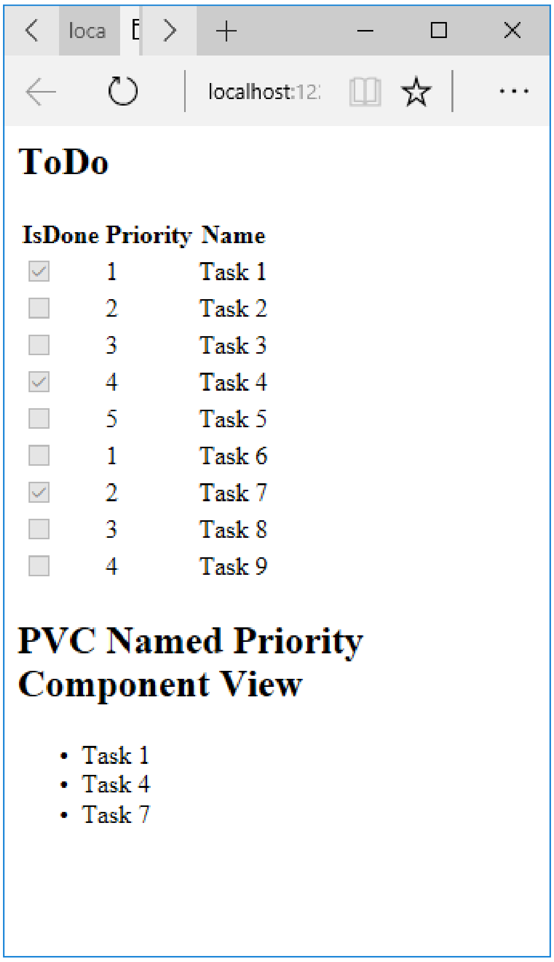
Copy the *Views/Shared/Components/PriorityList/Default.cshtml* file to a view named *Views/Shared/Components/PriorityList/PVC.cshtml*. Add a heading to indicate the PVC view is being used.

```
@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>
```

Update *Views/TodoList/Index.cshtml*:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Run the app and verify PVC view.



If the PVC view is not rendered, verify you are calling the view component with a priority of 4 or higher.

Examine the view path

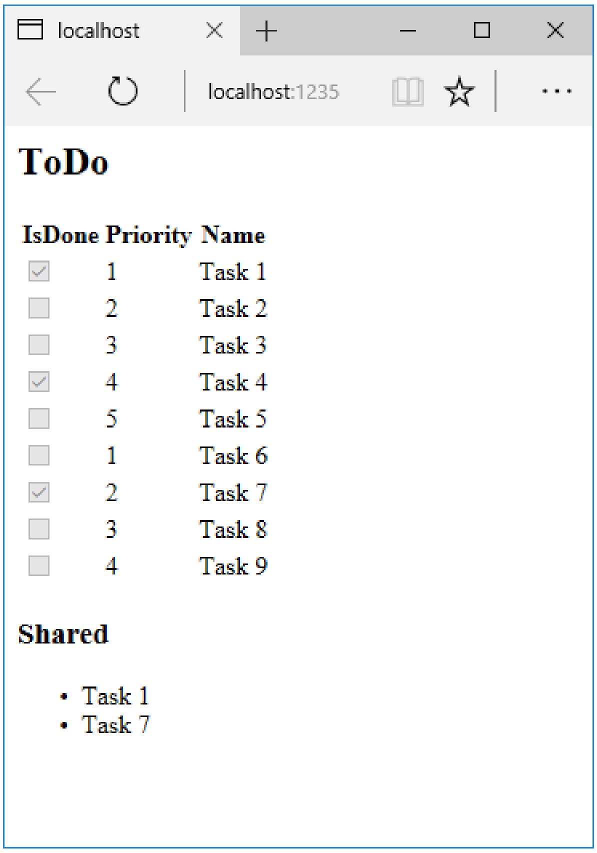
Change the priority parameter to three or less so the priority view is not returned.
Temporarily rename the *Views/Todo/Components/PriorityList/Default.cshtml* to *1Default.cshtml*.

Test the app, you'll get the following error:

```
An unhandled exception occurred while processing the request.
InvalidOperationException: The view 'Components/PriorityList/Default' was not found. The following locations
were searched:
/Views/ToDo/Components/PriorityList/Default.cshtml
/Views/Shared/Components/PriorityList/Default.cshtml
EnsureSuccessful
```

Copy *Views/Todo/Components/PriorityList/1Default.cshtml* to *Views/Shared/Components/PriorityList/Default.cshtml*.

Add some markup to the *Shared* Todo view component view to indicate the view is from the *Shared* folder.
Test the **Shared** component view.



Avoiding magic strings

If you want compile time safety, you can replace the hard-coded view component name with the class name. Create the view component without the "ViewComponent" suffix:

```
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();
        }
    }
}
```

Add a `using` statement to your Razor view file, and use the `nameof` operator:

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>
<!-- Markup removed for brevity.  -->
    }
</table>

<div>

    @await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
</div>
```

Additional Resources

Dependency injection into views

Building Web APIs

Create a web API with ASP.NET Core MVC and Visual Studio for Windows

By Rick Anderson and Mike Wasson

HTTP is not just for serving up web pages. It's also a powerful platform for building APIs that expose services and data. HTTP is flexible and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop apps.

In this tutorial, you'll build a web API for managing a list of "to-do" items. You won't build a UI.

ASP.NET Core has built-in support for MVC creating Web APIs.

There are 3 versions of this tutorial:

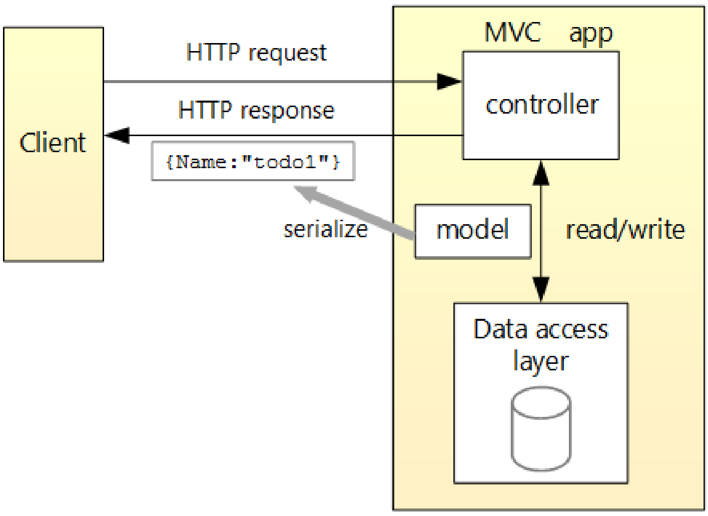
macOS: Web API with Visual Studio for Mac
Windows: Web API with Visual Studio for Windows
macOS, Linux, Windows: Web API with Visual Studio Code

Overview

Here is the API that you'll create:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item	None	None

The following diagram shows the basic design of the app.



The client is whatever consumes the web API (mobile app, browser, etc). We aren't writing a client in this tutorial. We'll use

[Postman] or [curl] to test the app.

A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as C# classes, also know as **P**lain **O**ld **C#** **O**bject (POCOs).

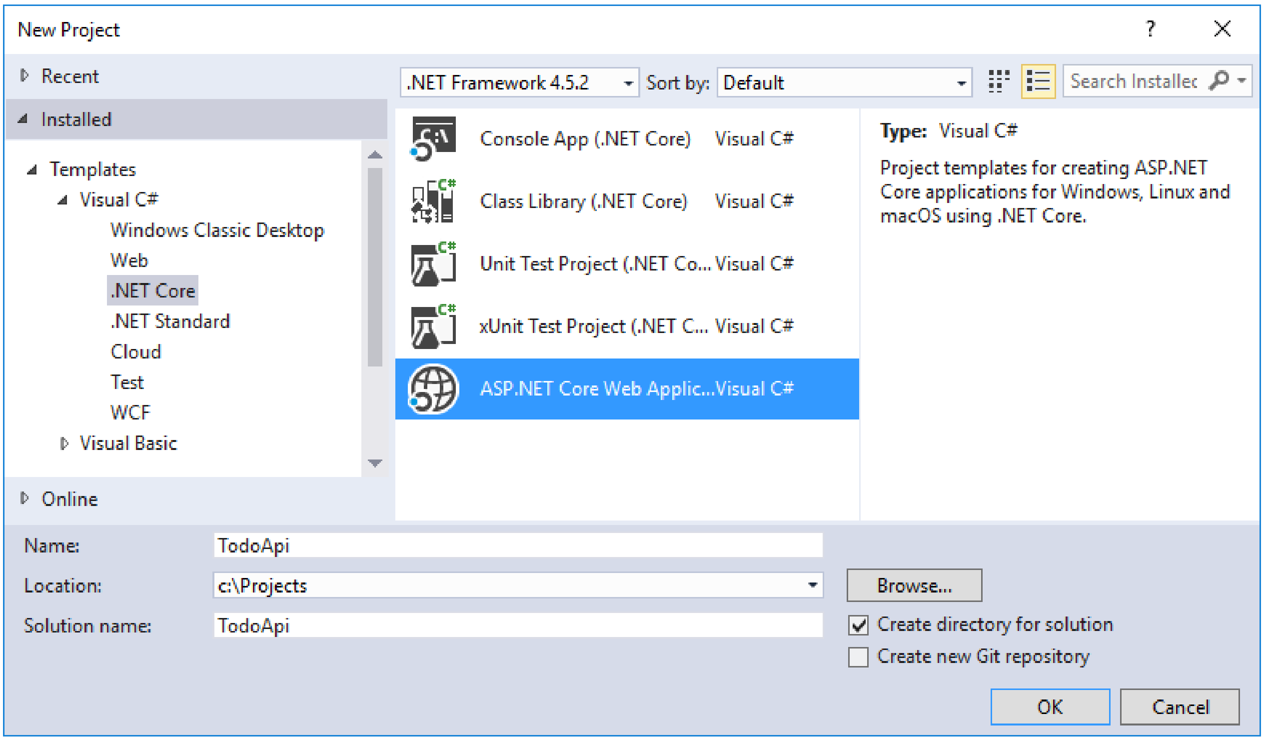
A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.

To keep the tutorial simple, the app doesn't use a persistent database. Instead, it stores to-do items in an in-memory database.

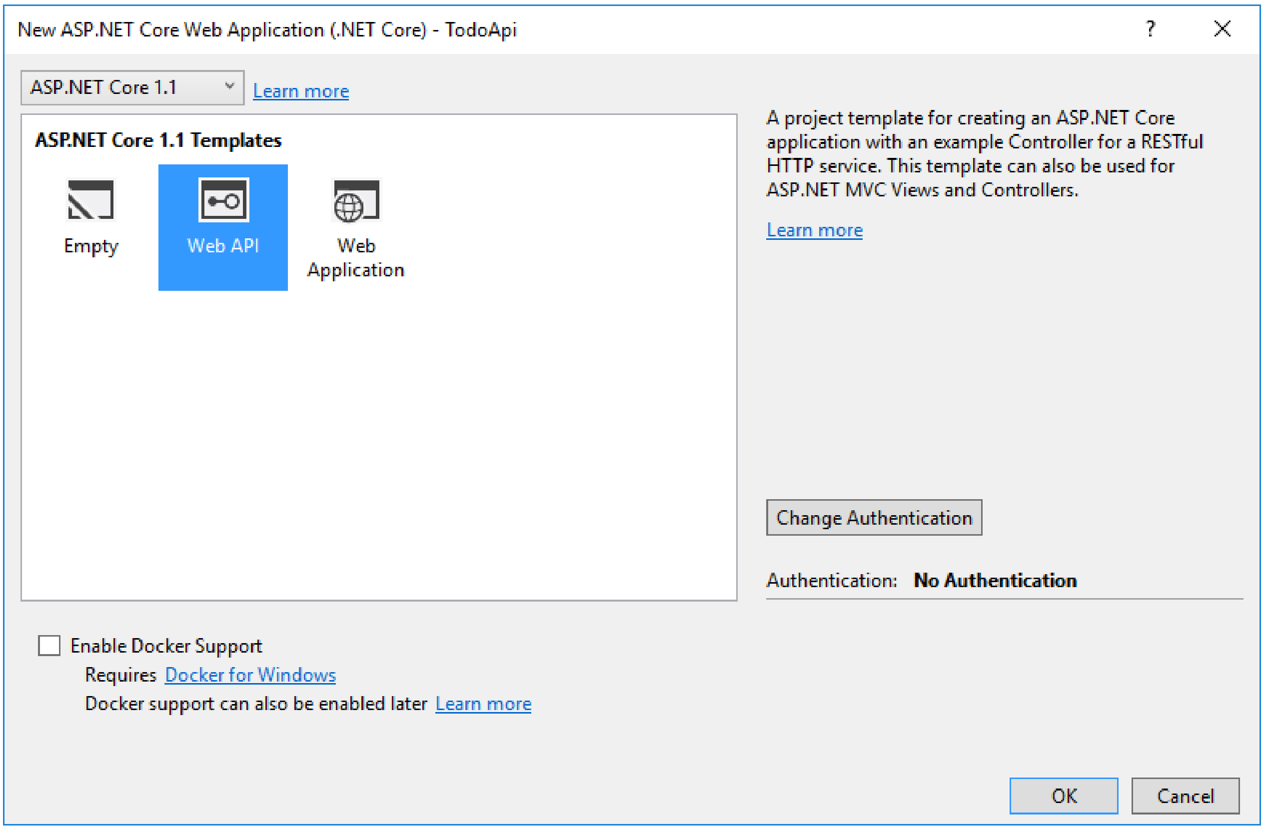
Create the project

From Visual Studio, select **File** menu, > **New** > **Project**.

Select the **ASP.NET Core Web Application (.NET Core)** project template. Name the project `TodoApi` and select **OK**.



In the **New ASP.NET Core Web Application (.NET Core) - TodoApi** dialog, select the **Web API** template. Select **OK**. Do **not** select **Enable Docker Support**.



Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the `ValuesController` data will be displayed:

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

If you're using IE, you are prompted to open or save the *values.json* file.

Add support for Entity Framework Core

Install the Entity Framework Core InMemory database provider. This database provider allows Entity Framework Core to be used with an in-memory database.

Edit the *TodoApi.csproj* file. In Solution Explorer, right-click the project. Select **Edit TodoApi.csproj**. In the `ItemGroup` element, add "Microsoft.EntityFrameworkCore.InMemory":

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="1.1.1" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="1.0.0" />
  </ItemGroup>

</Project>
```

Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

Add a folder named "Models". In Solution Explorer, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.

Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Add a `TodoItem` class. Right-click the *Models* folder and select **Add** > **Class**. Name the class `TodoItem` and select **Add**.

Replace the generated code with:

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

Create the database context

The *database context* is the main class that coordinates Entity Framework functionality for a given data model. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

Add a `TodoContext` class. Right-click the *Models* folder and select **Add** > **Class**. Name the class `TodoContext` and select **Add**.

```
using Microsoft.EntityFrameworkCore;

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

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

    }
}
```

Register the database context

In order to inject the database context into the controller, we need to register it with the dependency injection container. Register the database context with the service container using the built-in support for dependency injection. Replace the contents of the *Startup.cs* file with the following:

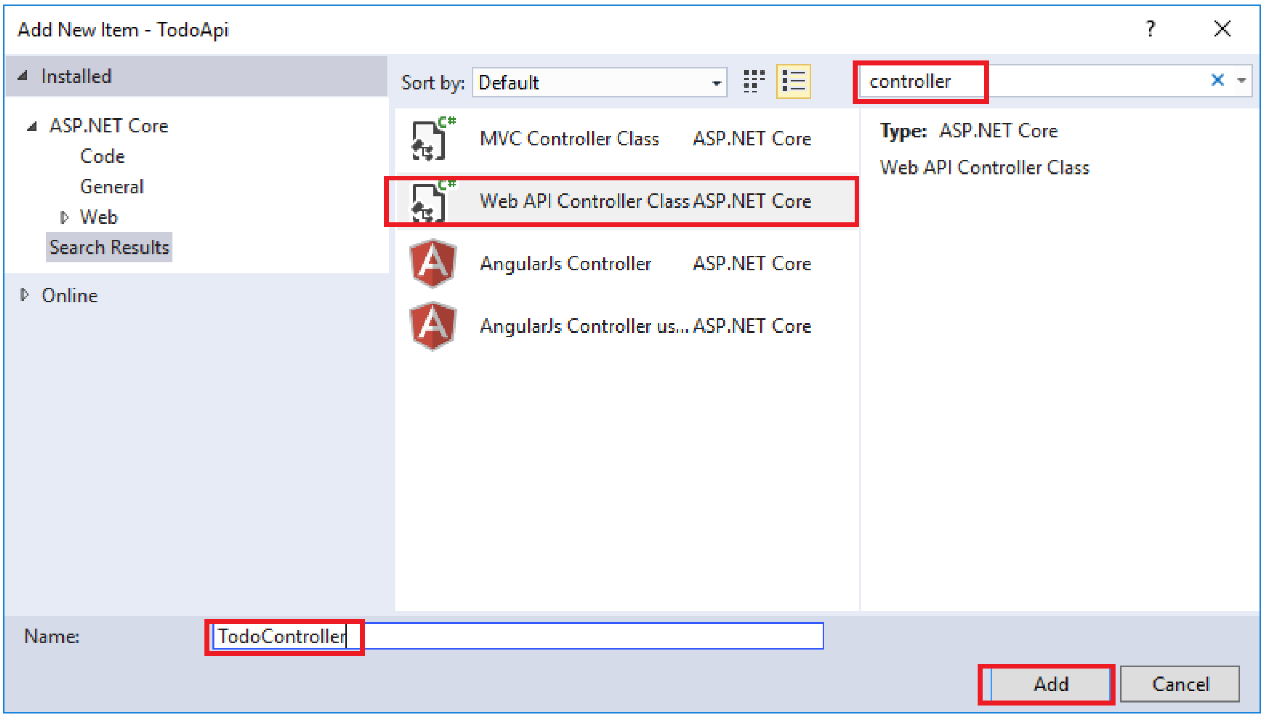
```
using Microsoft.AspNetCore.Builder;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase());
            services.AddMvc();
        }

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

The preceding code:

Removes the code we're not using.
Specifies an in-memory database is injected into the service container.

Add a controller

In Solution Explorer, right-click the *Controllers* folder. Select **Add** > **New Item**. In the **Add New Item** dialog, select the **Web API Controller Class** template. Name the class `TodoController`.



Replace the generated code with the following:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using TodoApi.Models;
using System.Linq;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

Defines an empty controller class. In the next sections, we'll add methods to implement the API.
The constructor uses Dependency Injection to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
The constructor adds an item to the in-memory database if one doesn't exist.

Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return _context.TodoItems.ToList();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
{
    var item = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

```
HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Server: Microsoft-IIS/10.0
   Date: Thu, 18 Jun 2015 20:51:10 GMT
   Content-Length: 82

   [{"Key":"1", "Name":"Item1","IsComplete":false}]
```

Later in the tutorial I'll show how you can view the HTTP response using Postman or or curl.

Routing and URL paths

The `[HttpGet]` attribute specifies an HTTP GET method. The URL path for each method is constructed as follows:

Take the template string in the controller's route attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        private readonly TodoContext _context;
```

Replace "[Controller]" with the name of the controller, which is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller and the root name is "todo". ASP.NET Core routing is not case sensitive. If the `[HttpGet]` attribute has a route template (such as `[HttpGet("/products")]`, append that to the path. This sample doesn't use a template. See Attribute routing with Http[Verb] attributes for more information.

In the `GetById` method:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

`"{id}"` is a placeholder variable for the ID of the `todo` item. When `GetById` is invoked, it assigns the value of "{id}" in the URL to the method's `id` parameter.

`Name = "GetTodo"` creates a named route and allows you to link to this route in an HTTP Response. I'll explain it with an example

later. See Routing to Controller Actions for detailed information.

Return values

The `GetAll` method returns an `IEnumerable`. MVC automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

In contrast, the `GetById` method returns the more general `IActionResult` type, which represents a wide range of return types. `GetById` has two different return types:

If no item matches the requested ID, the method returns a 404 error. This is done by returning `NotFound`.

Otherwise, the method returns 200 with a JSON response body. This is done by returning an `ObjectResult`

Launch the app

In Visual Studio, press CTRL+F5 to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/values`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the data will be displayed. If you're using IE, IE will prompt to you open or save the *values.json* file. Navigate to the `Todo` controller we just created `http://localhost:port/api/todo`.

Implement the other CRUD operations

We'll add `Create`, `Update`, and `Delete` methods to the controller. These are variations on a theme, so I'll just show the code and highlight the main differences. Build the project after adding or changing code.

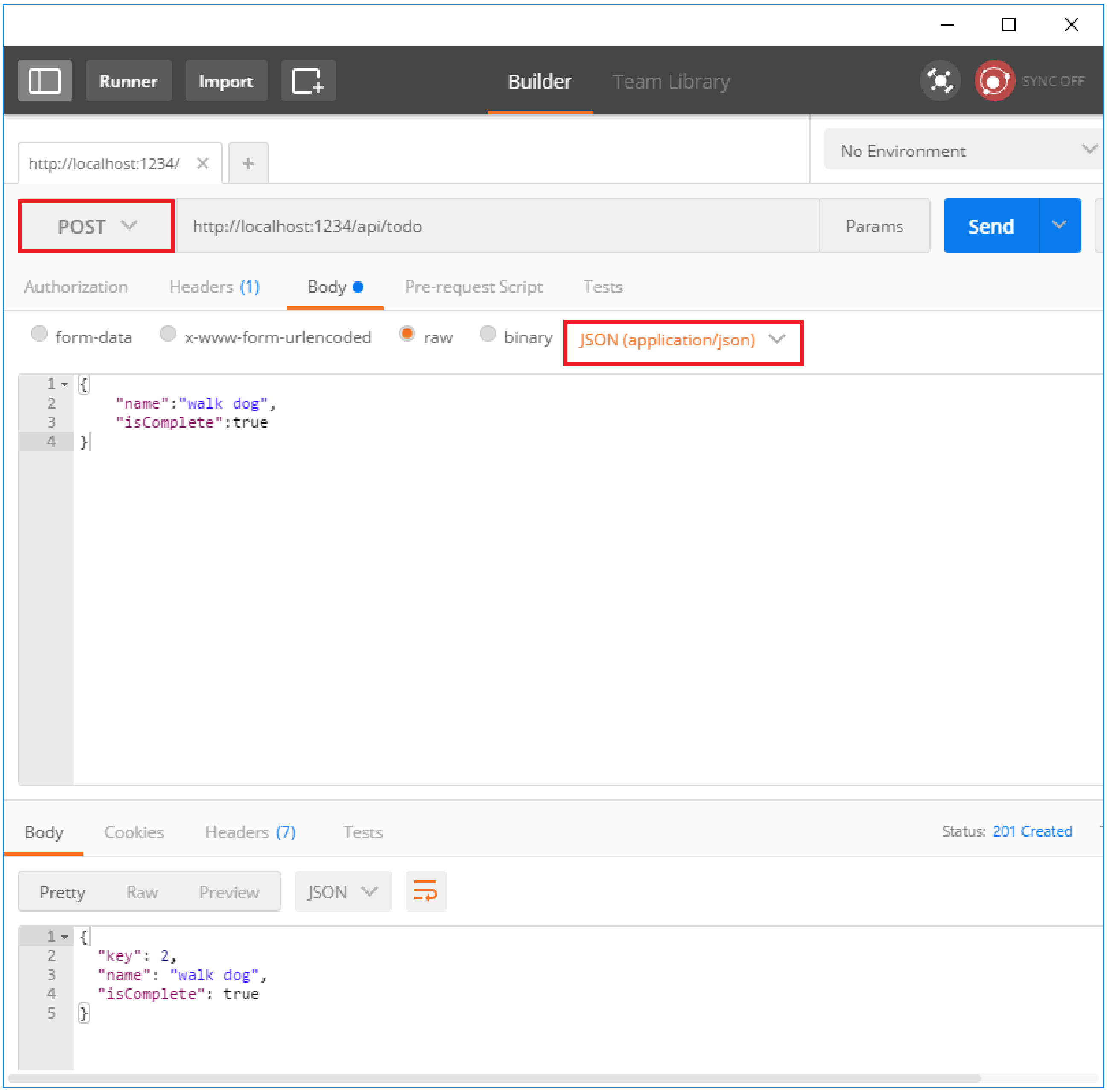
Create

```
[HttpPost]
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);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See 10.2.2 201 Created.

Use Postman to send a Create request

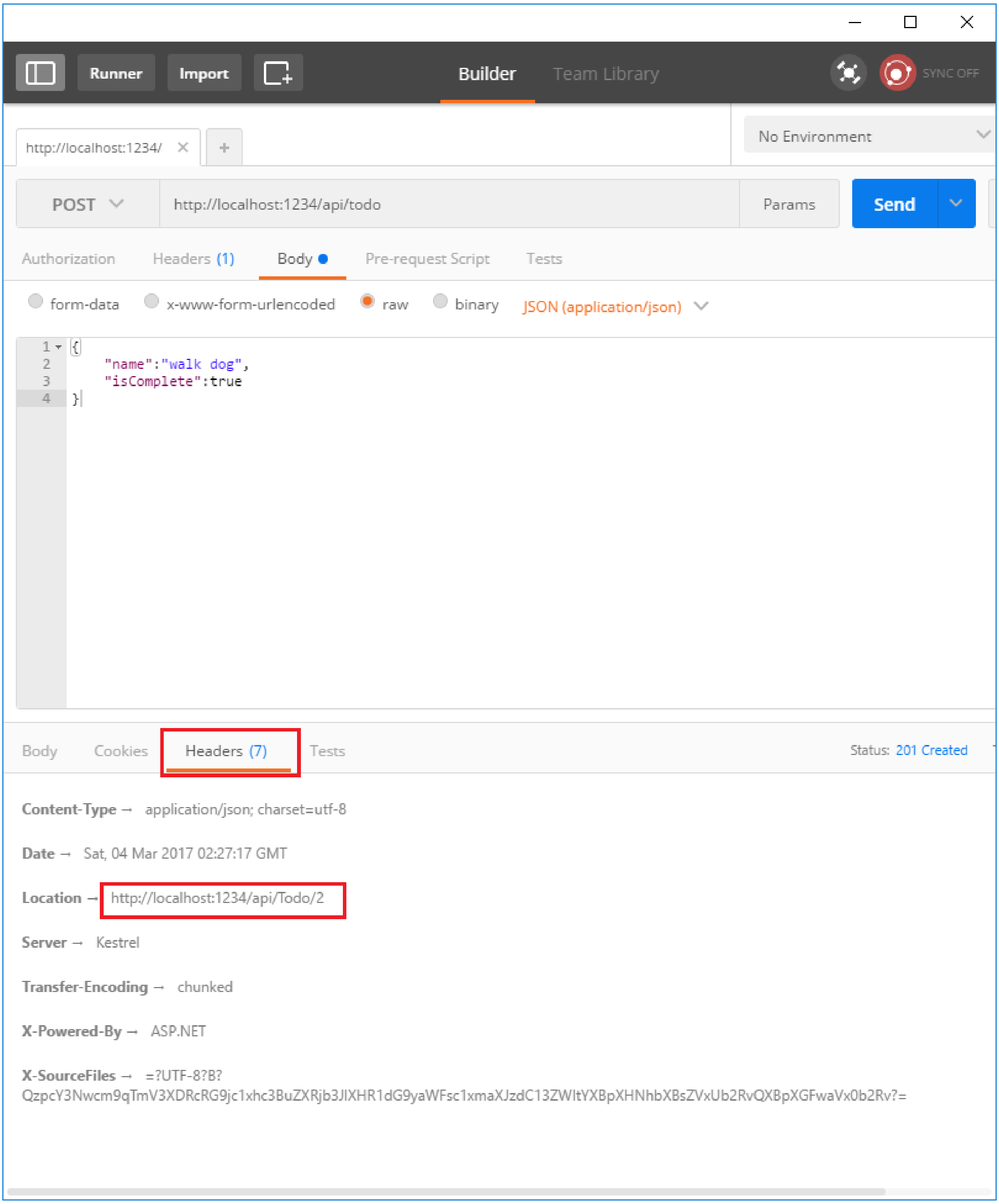


Set the HTTP method to `POST`
Select the **Body** radio button
Select the **raw** radio button
Set the type to JSON
In the key-value editor, enter a Todo item such as

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

Select **Send**

Select the Headers tab in the lower pane and copy the **Location** header:



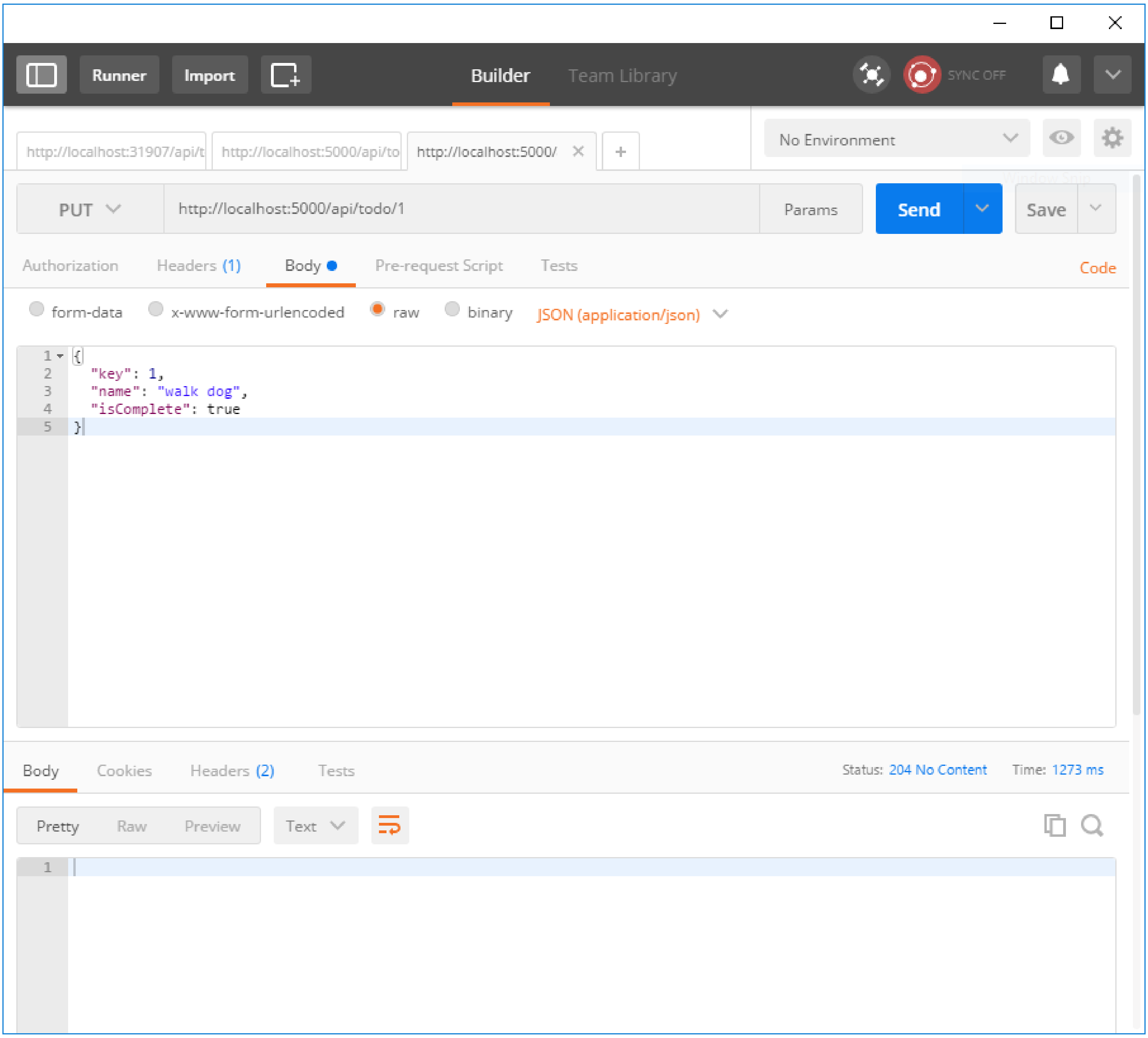
You can use the Location header URI to access the resource you just created. Recall the `GetById` method created the `"GetTodo"` named route:

```
[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(long id)
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(long id, [FromBody] TodoItem item)
{
    if (item == null || item.Id != id)
    {
        return BadRequest();
    }

    var todo = _context.TodoItems.FirstOrDefault(t => t.Id == id);
    if (todo == null)
    {
        return NotFound();
    }

    todo.IsComplete = item.IsComplete;
    todo.Name = item.Name;

    _context.TodoItems.Update(todo);
    _context.SaveChanges();
    return new NoContentResult();
}
```

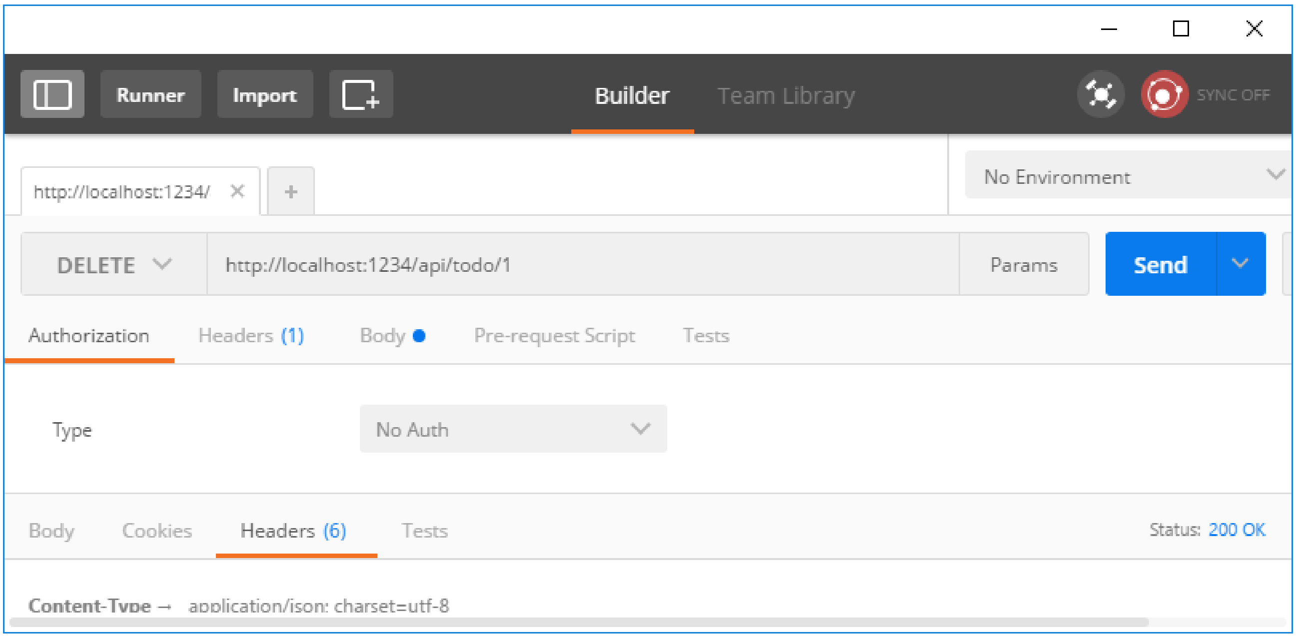
`Update` is similar to `Create`, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.



Delete

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

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

The response is 204 (No Content).



Next steps

Routing to Controller Actions

For information about deploying your API, see Publishing and Deployment.

View or download sample code. See how to download.

Postman

ASP.NET Web API Help Pages using Swagger

By Shayne Boyer

Understanding the various methods of an API can be a challenge for a developer when building a consuming application.

Generating good documentation and help pages as a part of your Web API using Swagger with the .NET Core implementation Swashbuckle.AspNetCore is as easy as adding a couple of NuGet packages and modifying the *Startup.cs*.

Swashbuckle.AspNetCore is an open source project for generating Swagger documents for Web APIs that are built with ASP.NET Core MVC.

Swagger is a machine readable representation of a RESTful API that enables support for interactive documentation, client SDK generation and discoverability.

This tutorial builds on the sample on Building Your First Web API with ASP.NET Core MVC and Visual Studio. If you'd like to follow along, download the sample at https://github.com/aspnet/Docs/tree/master/aspnetcore/tutorials/first-web-api/sample.

Getting Started

There are three main components to Swashbuckle

Swashbuckle.AspNetCore.Swagger : a Swagger object model and middleware to expose SwaggerDocument objects as JSON endpoints.

Swashbuckle.AspNetCore.SwaggerGen : a Swagger generator that builds SwaggerDocument objects directly from your routes, controllers and models. Typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.

Swashbuckle.AspNetCore.SwaggerUI : an embedded version of the Swagger UI tool which interprets Swagger JSON to build a rich customizable experience for describing the Web API functionality, and includes built in test harness capabilities for the public methods.

NuGet Packages

You can add Swashbuckle with any of the following approaches:

From the Package Manager Console:

```
Install-Package Swashbuckle.AspNetCore
```

In Visual Studio:

Right click your project in Solution Explorer > Manage NuGet Packages
Enter Swashbuckle.AspNetCore in the search box
Set the Package source to nuget.org
Tap the Swashbuckle.AspNetCore package and then tap Install

Add and configure Swagger to the middleware

Add SwaggerGen to the services collection in the Configure method, and in the ConfigureServices method, enable the middleware for serving generated JSON document and the SwaggerUI.

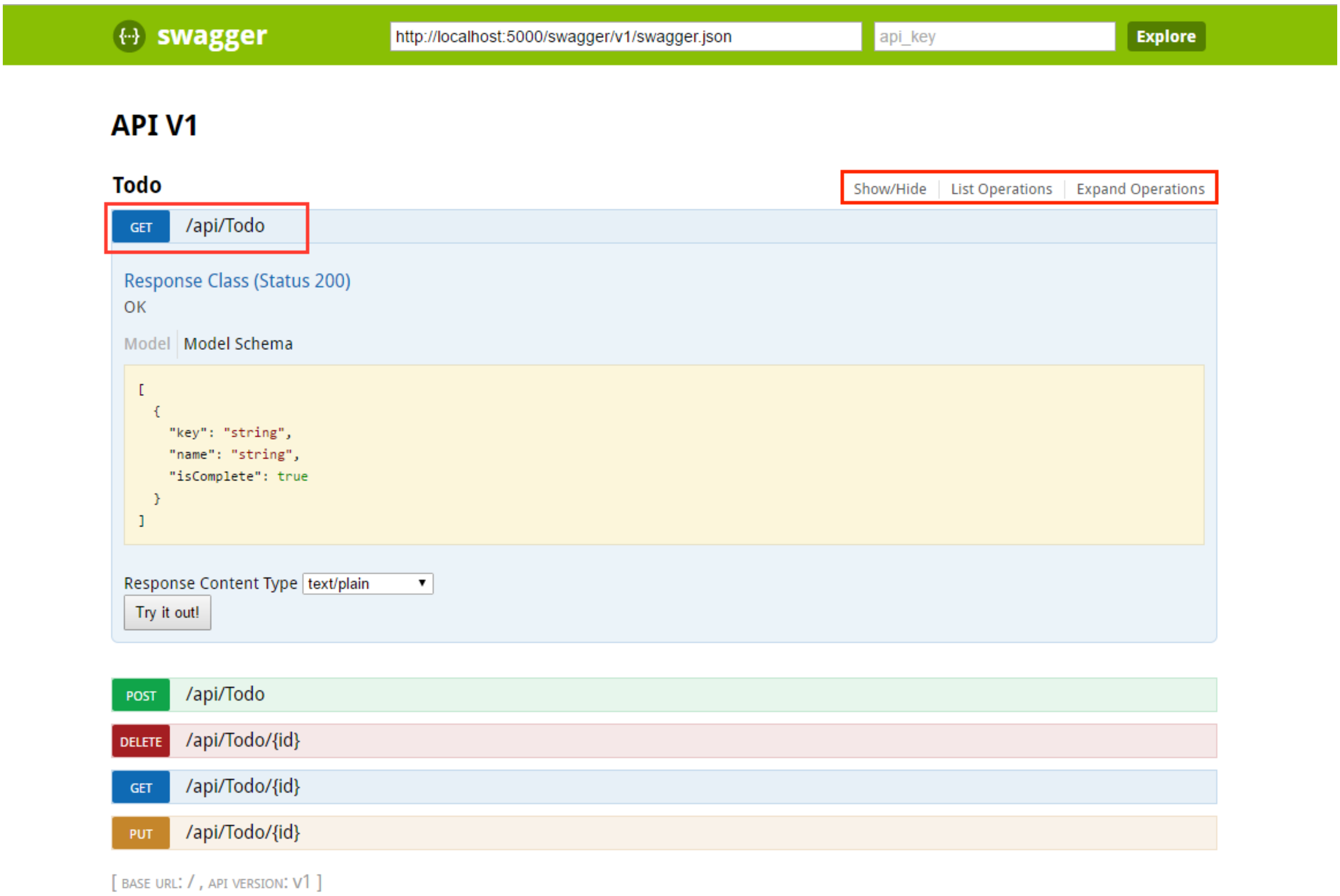
```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddLogging();

    // Add our repository type
    services.AddSingleton<ITodoRepository, TodoRepository>();

    // Register the Swagger generator, defining one or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
    });
}

// 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)
{
    app.UseMvcWithDefaultRoute();

    // 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");
    });
}
```

In Visual Studio, press ^F5 to launch the app and navigate to `http://localhost:<random_port>/swagger/v1/swagger.json` to see the document generated that describes the endpoints.

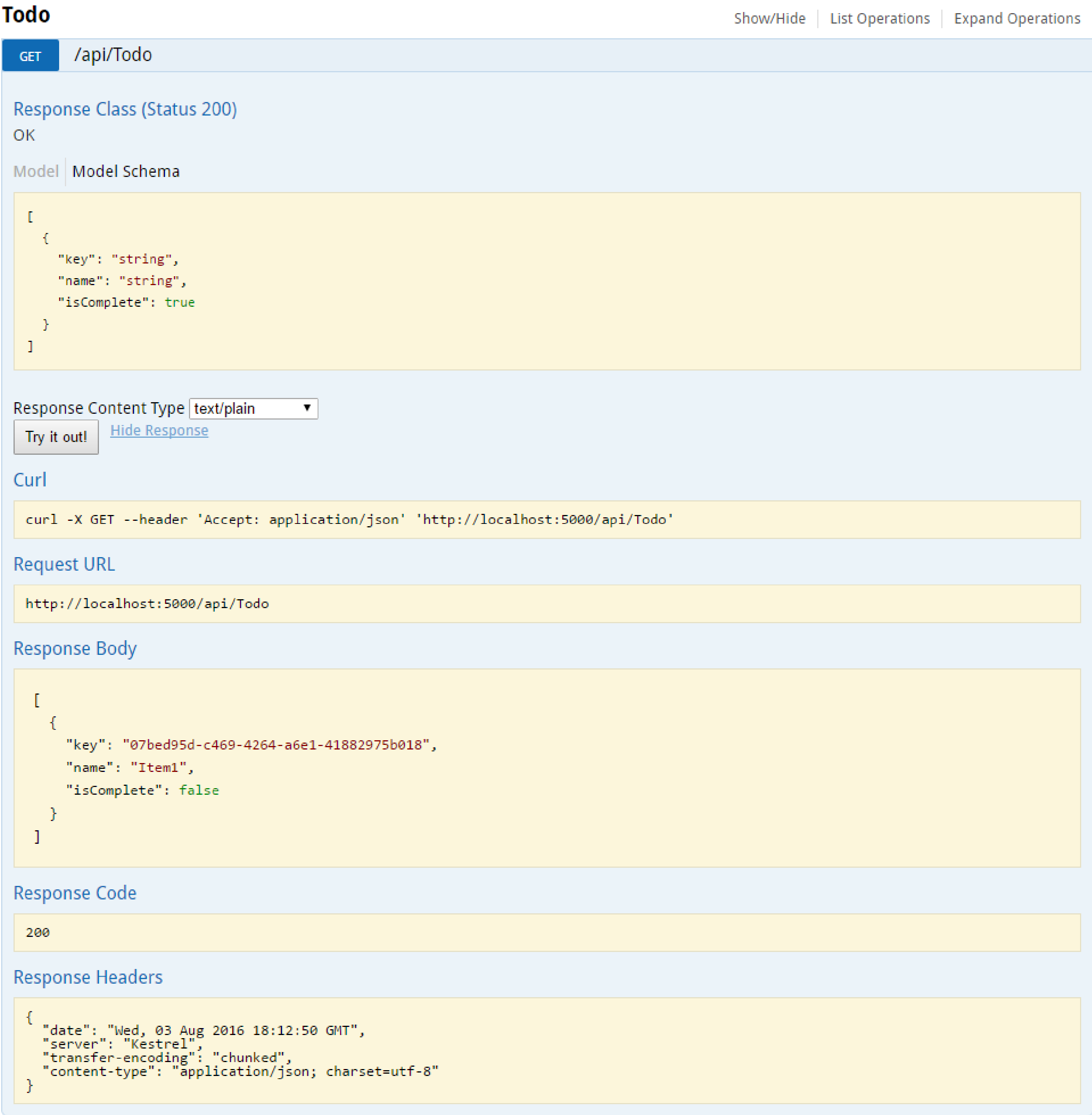
Note

Microsoft Edge, Google Chrome and Firefox display JSON documents natively. There are extensions for Chrome that will format the document for easier reading. *Example below reduced for brevity.*

This document is used to drive the Swagger UI which can be viewed by navigating to `http://localhost:<random_port>/swagger`



Each of the methods in the ToDo controller can be tested from the UI. Tap a method to expand the section, add any necessary parameters and tap "Try it out!".



Customization & Extensibility

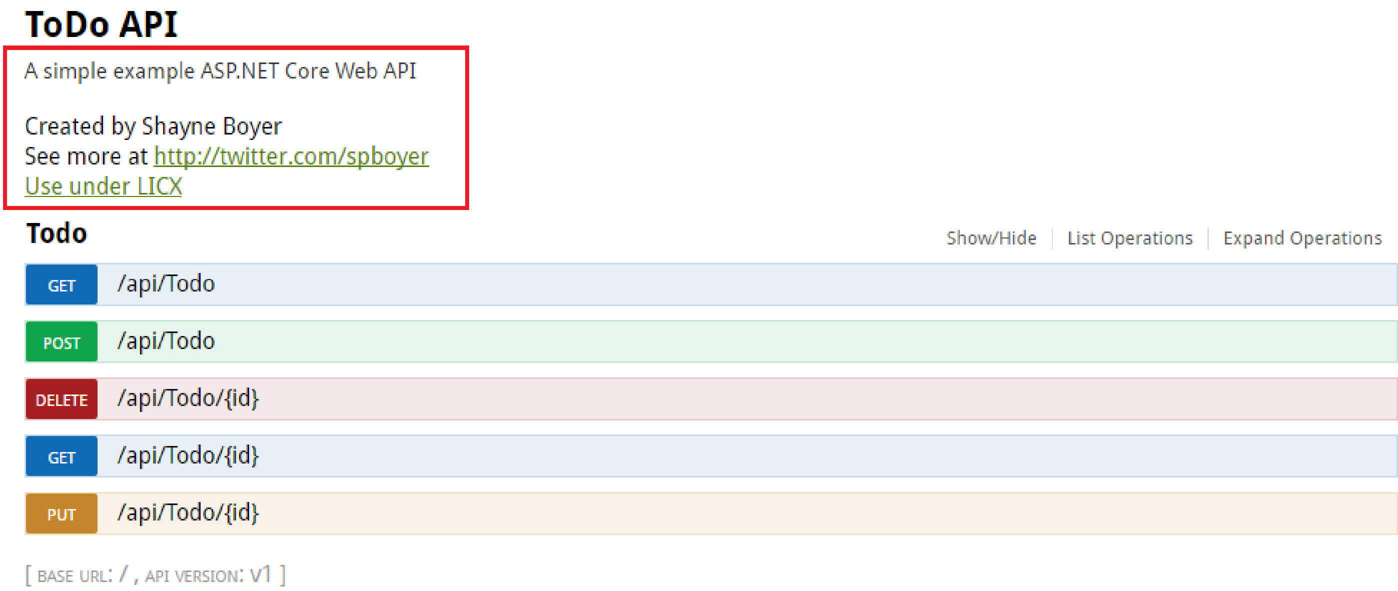
Swagger is not only a simple way to represent the API, but has options for documenting the object model, as well as customizing the interactive UI to match your look and feel or design language.

API Info and Description

The config. action passed to the `AddSwaggerGen` method can be used to add information such as the author, license, description.

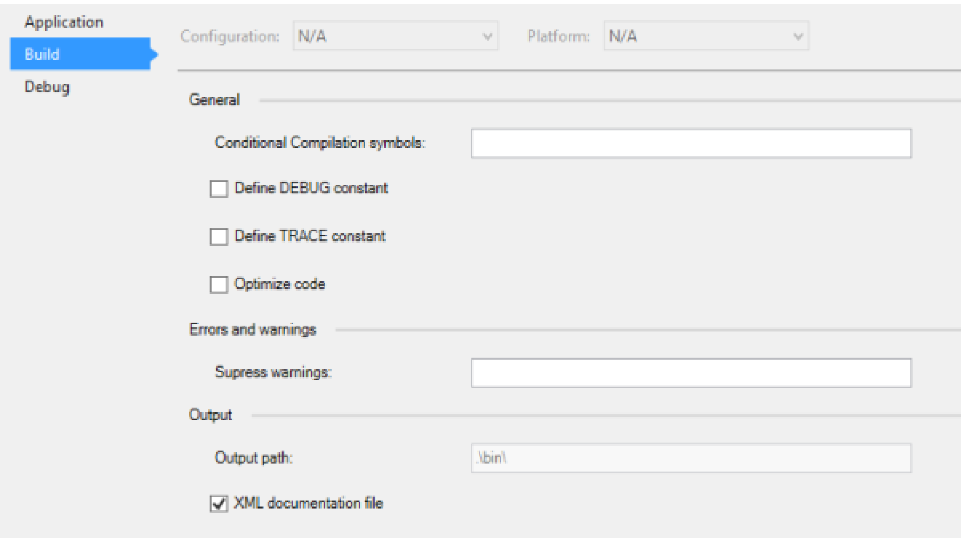
```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = "None",
        Contact = new Contact { Name = "Shayne Boyer", Email = "", Url = "https://twitter.com/spboyer"},
        License = new License { Name = "Use under LICX", Url = "https://example.com/license" }
    });
});
```

The following image shows the Swagger UI displaying the version information added.



XML Comments

To enable XML comments, right click the project in Visual Studio and select **Properties** and then check the **XML Documentation file** box under the **Output Settings** section.



Configure Swagger to use the generated XML file.

Note

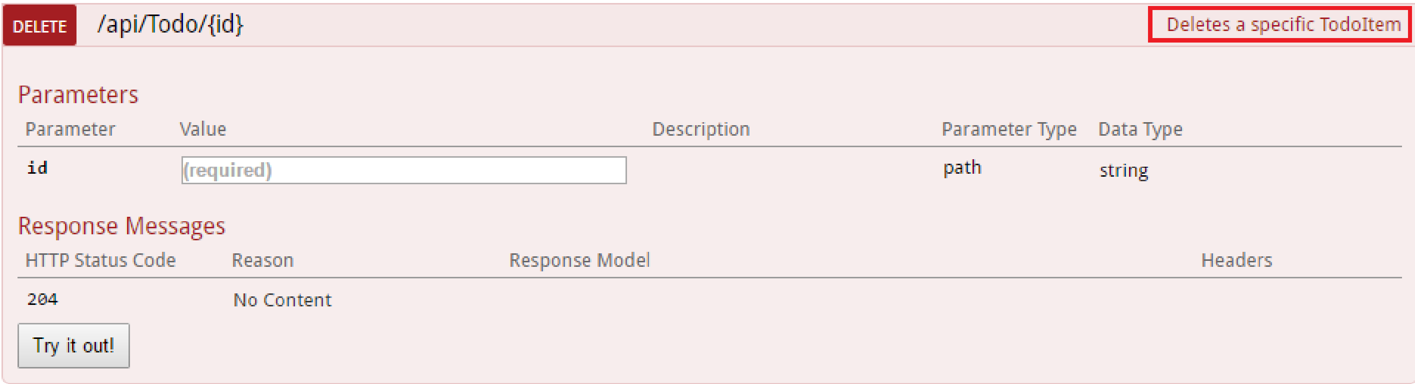
For Linux or non-Windows operating systems, file names and paths can be case sensitive. So `ToDoApi.XML` would be found on Windows but not CentOS for example.

```
// 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();

    services.AddLogging();

    // Add our repository type.
    services.AddSingleton<ITodoRepository, TodoRepository>();

 // 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 = "", Url = "http://twitter.com/spboyer"},
            License = new License { Name = "Use under LICX", Url = "http://url.com" }
        });

        //Set the comments path for the swagger json and ui.
        var basePath = PlatformServices.Default.Application.ApplicationBasePath;
        var xmlPath = Path.Combine(basePath, "TodoApi.xml");
        c.IncludeXmlComments(xmlPath);
    });
}

// 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)
{
    app.UseStaticFiles();

    app.UseMvcWithDefaultRoute();

    // 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");
    });
}
```

In the code above, ApplicationBasePath gets the base path of the app, which is needed to set the full path to the XML comments. `TodoApi.xml` only works for this example, the name of the generated XML comments file is based on the name of your application.

Adding the triple slash comments to the method enhances the Swagger UI by adding the description to the header of the section.

```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public void Delete(string id)
{
    TodoItems.Remove(id);
}
```

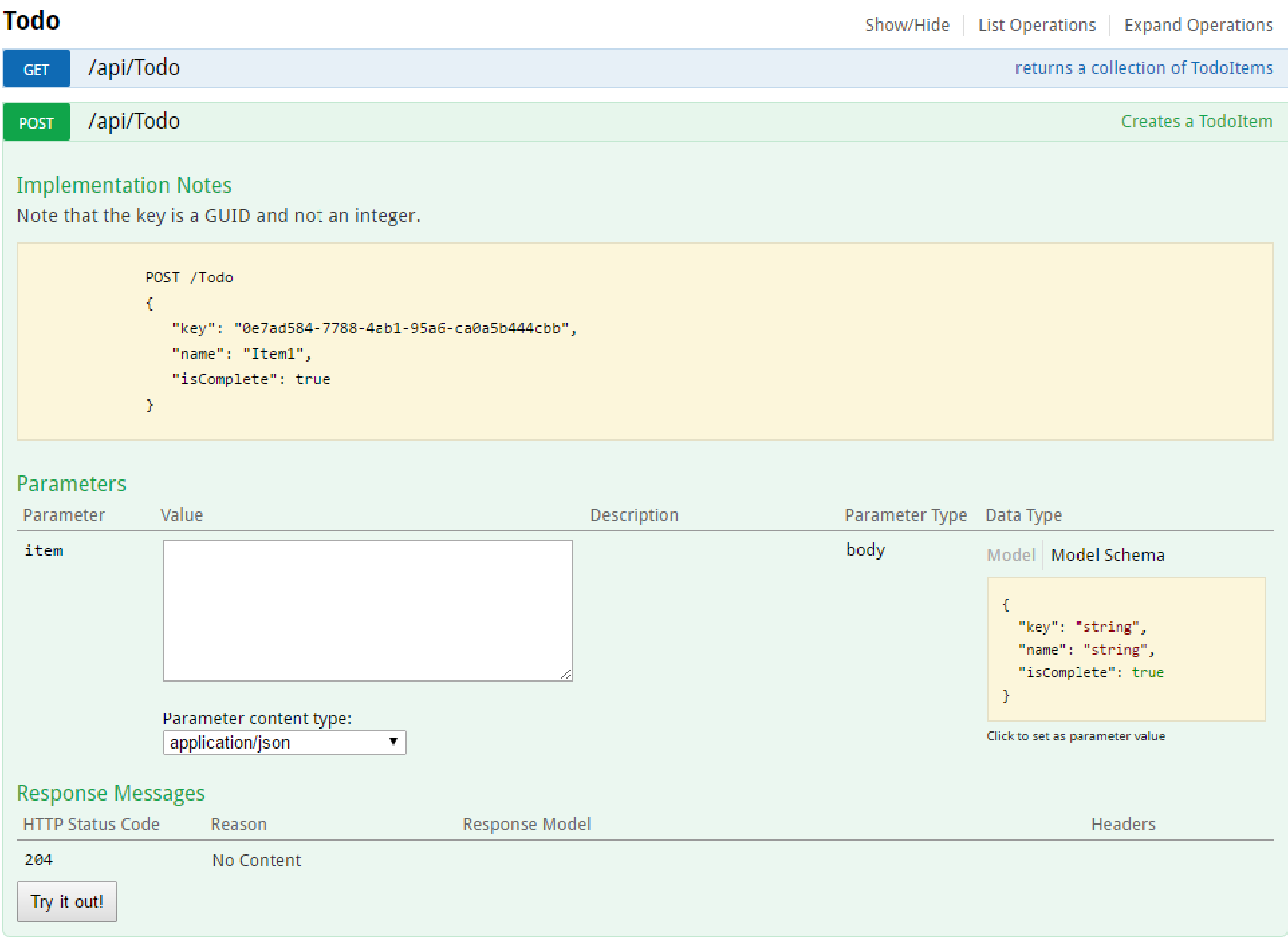


Note that the UI is driven by the generated JSON file, and these comments are also in that file as well.

Here is a more robust example, adding `<remarks />` where the content can be just text or adding the JSON or XML object for further documentation of the method.

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Note that the key is a GUID and not an integer.
///
///     POST /Todo
///     {
///        "key": "0e7ad584-7788-4ab1-95a6-ca0a5b444cbb",
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>New Created Todo Item</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(typeof(TodoItem), 400)]
public IActionResult Create([FromBody, Required] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }
    TodoItems.Add(item);
    return CreatedAtRoute("GetTodo", new { id = item.Key }, item);
}
```

Notice the enhancement of the UI with these additional comments.



DataAnnotations

You can decorate the API controller with `System.ComponentModel.DataAnnotations` to help drive the Swagger UI components.

Adding the `[Required]` annotation to the `Name` property of the `TodoItem` class will change the ModelSchema information in the UI. `[Produces("application/json")]`, `RegularExpression` validators and more will further detail the information delivered in the generated page. The more metadata that is in the code produces a more desciptive UI or API help page.

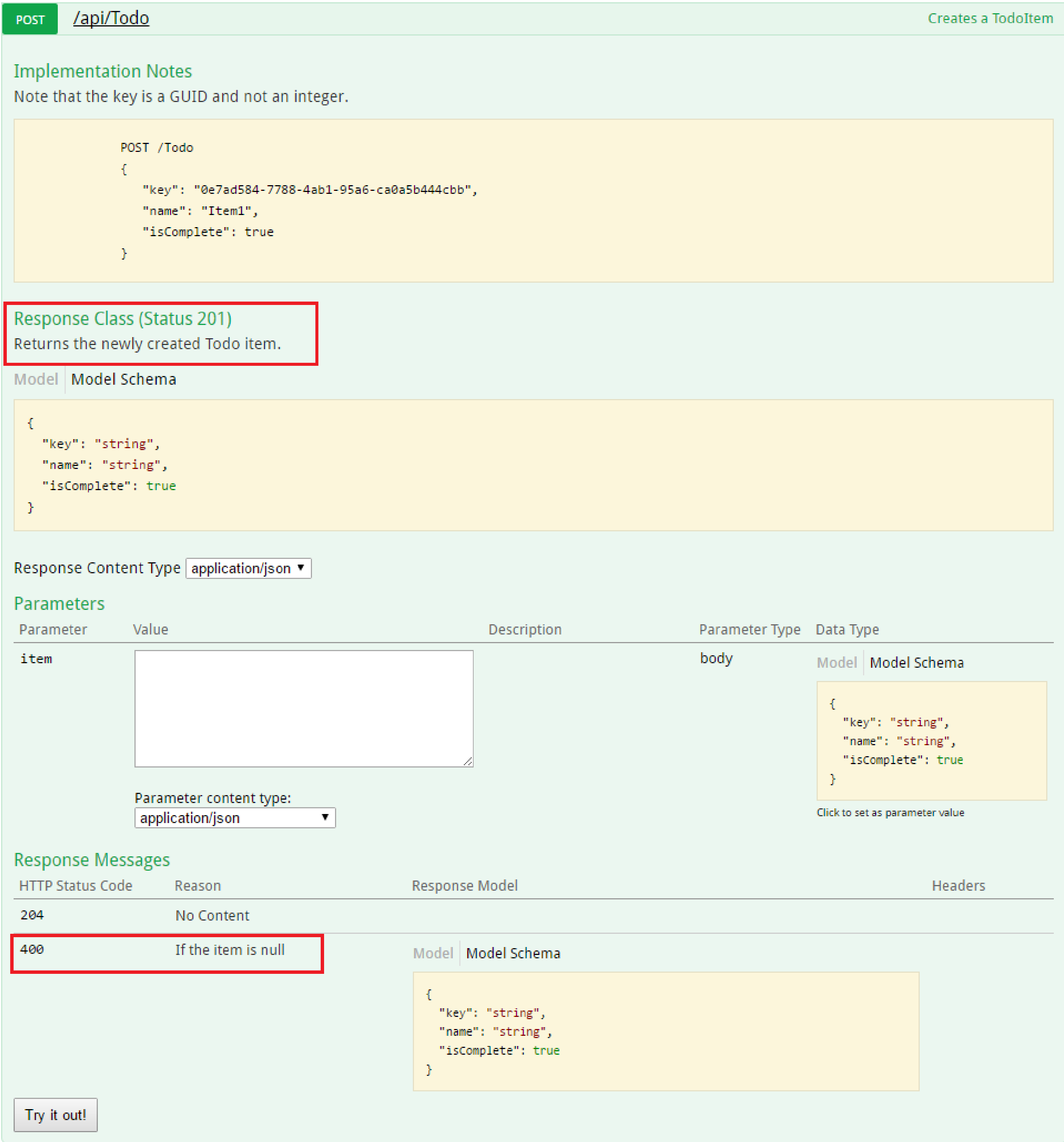
```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models
{
    public class TodoItem
    {
        public string Key { get; set; }
        [Required]
        public string Name { get; set; }
        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}
```

Describing Response Types

Consuming developers are probably most concerned with what is returned; specifically response types, error codes (if not standard). These are handled in the XML comments and DataAnnotations.

Take the `Create()` method for example, currently it returns only "201 Created" response by default. That is of course if the item is in fact created, or a "204 No Content" if no data is passed in the POST Body. However, there is no documentation to know that or any other response. That can be fixed by adding the following piece of code.

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Note that the key is a GUID and not an integer.
///
///     POST /Todo
///     {
///        "key": "0e7ad584-7788-4ab1-95a6-ca0a5b444cbb",
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>New Created Todo Item</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(typeof(TodoItem), 400)]
public IActionResult Create([FromBody, Required] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }
    TodoItems.Add(item);
    return CreatedAtRoute("GetTodo", new { id = item.Key }, item);
}
```



Customizing the UI

The stock UI is very functional as well as presentable, however when building documentation pages for your API you want it to represent your brand or look and feel.

Accomplishing that task with the Swashbuckle components is simple but requires adding the resources to serve static files that would not normally be included in a Web API project and then building the folder structure to host those files.

Add the `"Microsoft.AspNetCore.StaticFiles": "1.0.0-*"` NuGet package to the project.

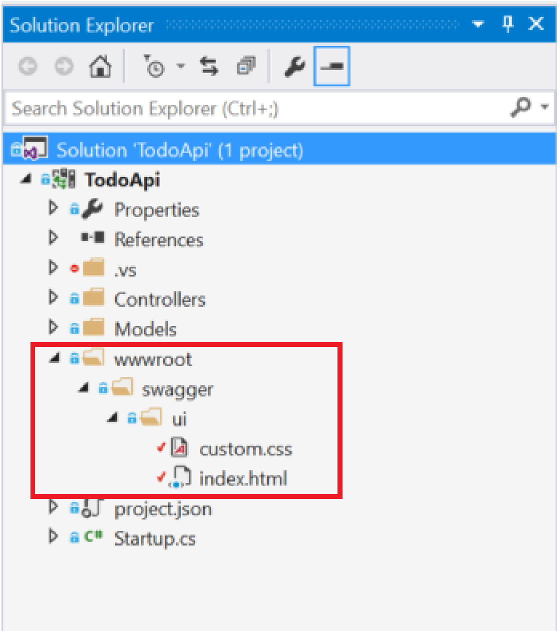
Enable static files middleware.

```
// 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)
    {
        // Enable static files middleware.
        app.UseStaticFiles();

        app.UseMvcWithDefaultRoute();

         // 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");
         });
    }
```

Acquire the core *index.html* file used for the Swagger UI page from the

```
Github repository <https://github.com/domaindrivendev/Swashbuckle.AspNetCore/blob/1.0.0-rc1/test/WebSites/CustomIndexHtml/wwwroot/swagger/index.html>
```

_ and put that in the `wwwroot/swagger` folder and also create a new `custom.css` file in the same folder.



Reference *custom.css* in the *index.html* file.

```
<link href='custom.css' media='screen' rel='stylesheet' type='text/css' />
```

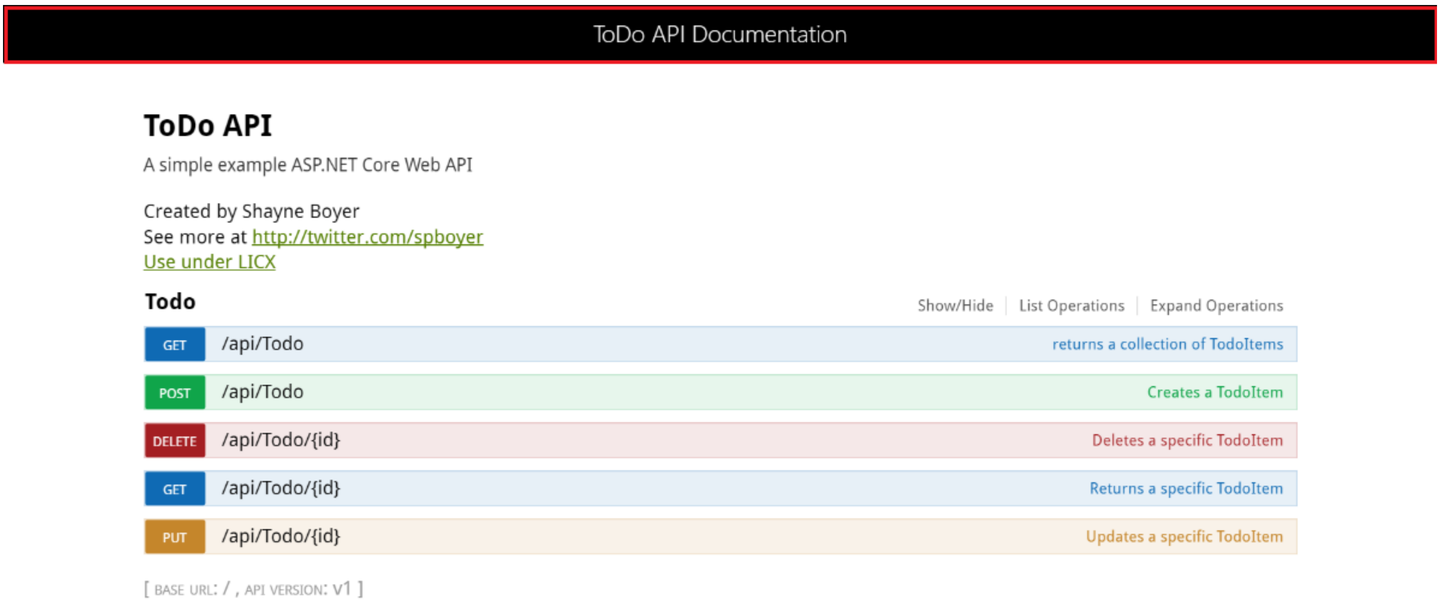
The following CSS provides a simple sample of a custom header title to the page.

custom.css file

```
.swagger-section #header
{
    border-bottom: 1px solid #000000;
    font-style: normal;
    font-weight: 400;
    font-family: "Segoe UI Light","Segoe WP Light","Segoe UI","Segoe WP",Tahoma,Arial,sans-serif;
    background-color: black;
}

.swagger-section #header h1
{
    text-align: center;
    font-size: 20px;
    color: white;
}
```

index.html body

```
<body class="swagger-section">
   <div id="header">
    <h1>ToDo API Documentation</h1>
   </div>

   <div id="message-bar" class="swagger-ui-wrap" data-sw-translate> </div>
   <div id="swagger-ui-container" class="swagger-ui-wrap"></div>
</body>
```



There is much more you can do with the page, see the full capabilities for the UI resources at the Swagger UI Github repository.

Creating Backend Services for Native Mobile Applications

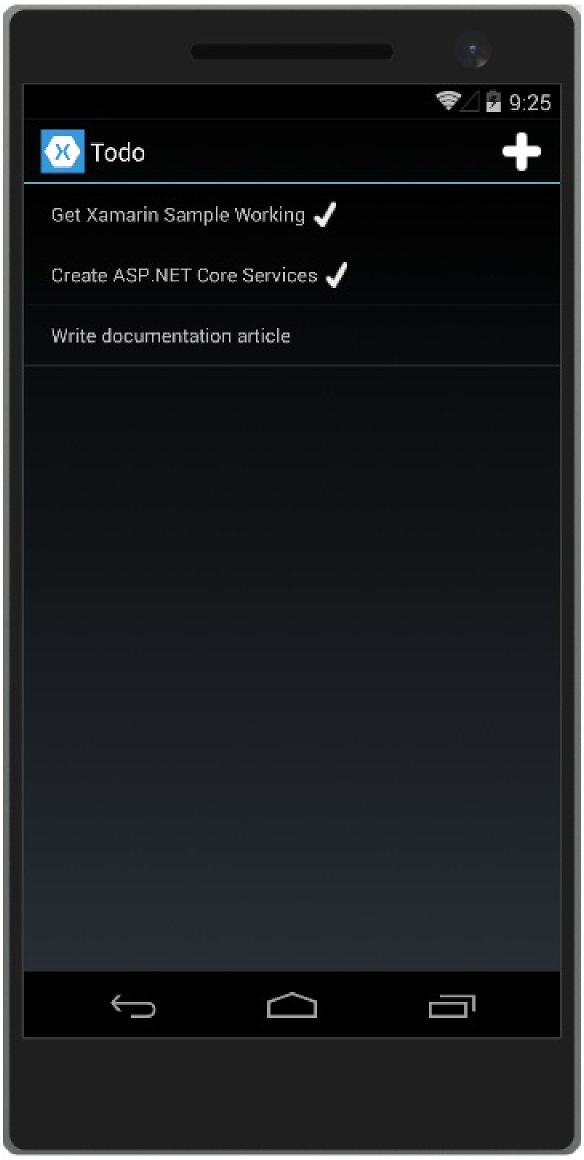
By Steve Smith

Mobile apps can easily communicate with ASP.NET Core backend services.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the Xamarin Forms ToDoRest app as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

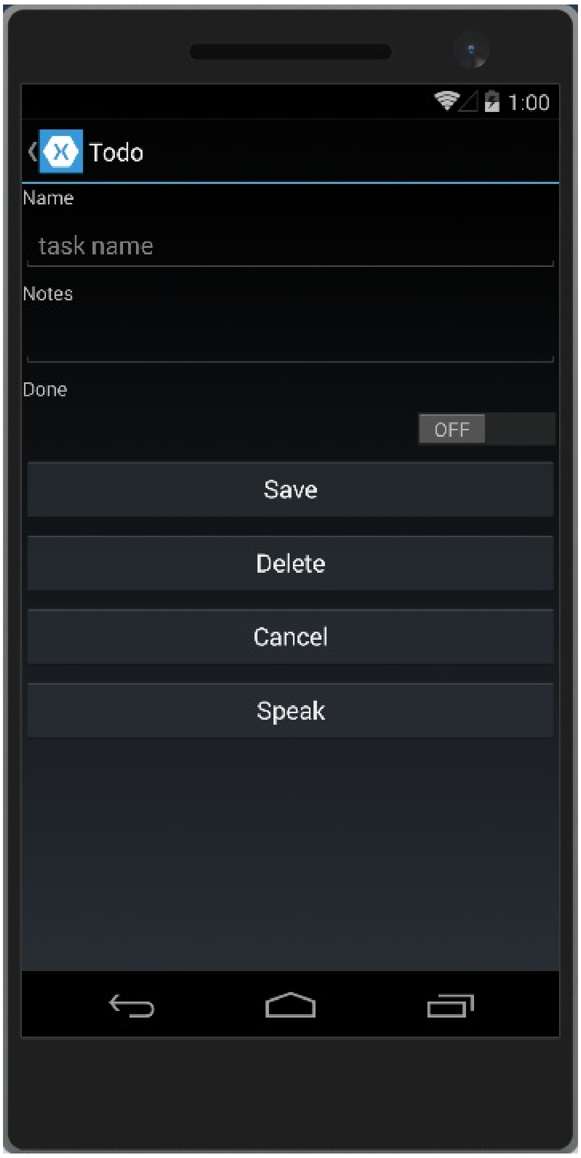


Features

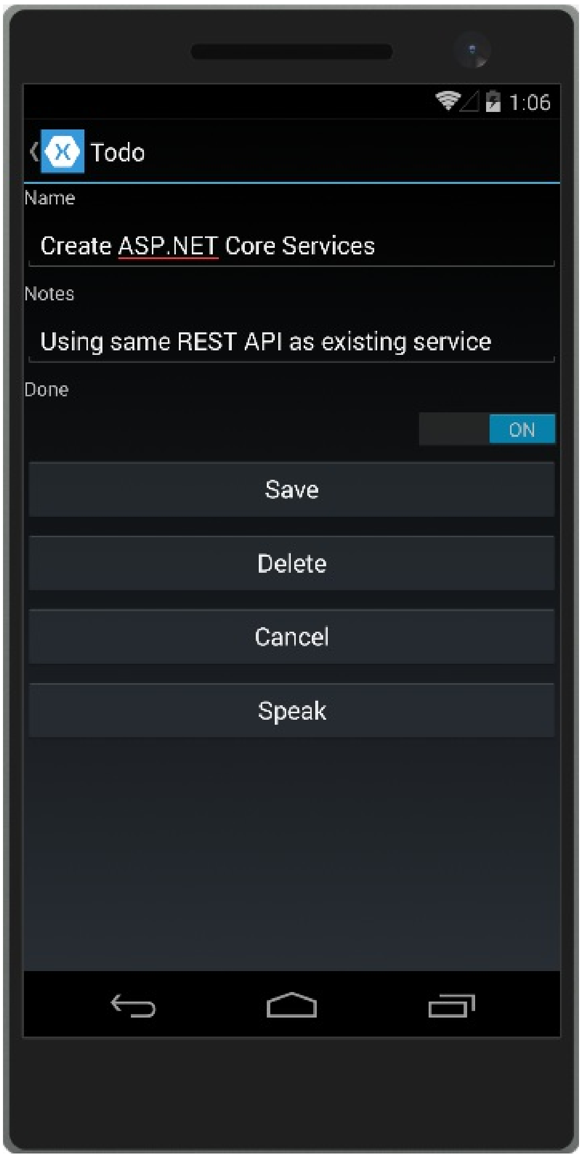
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the `+` icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:

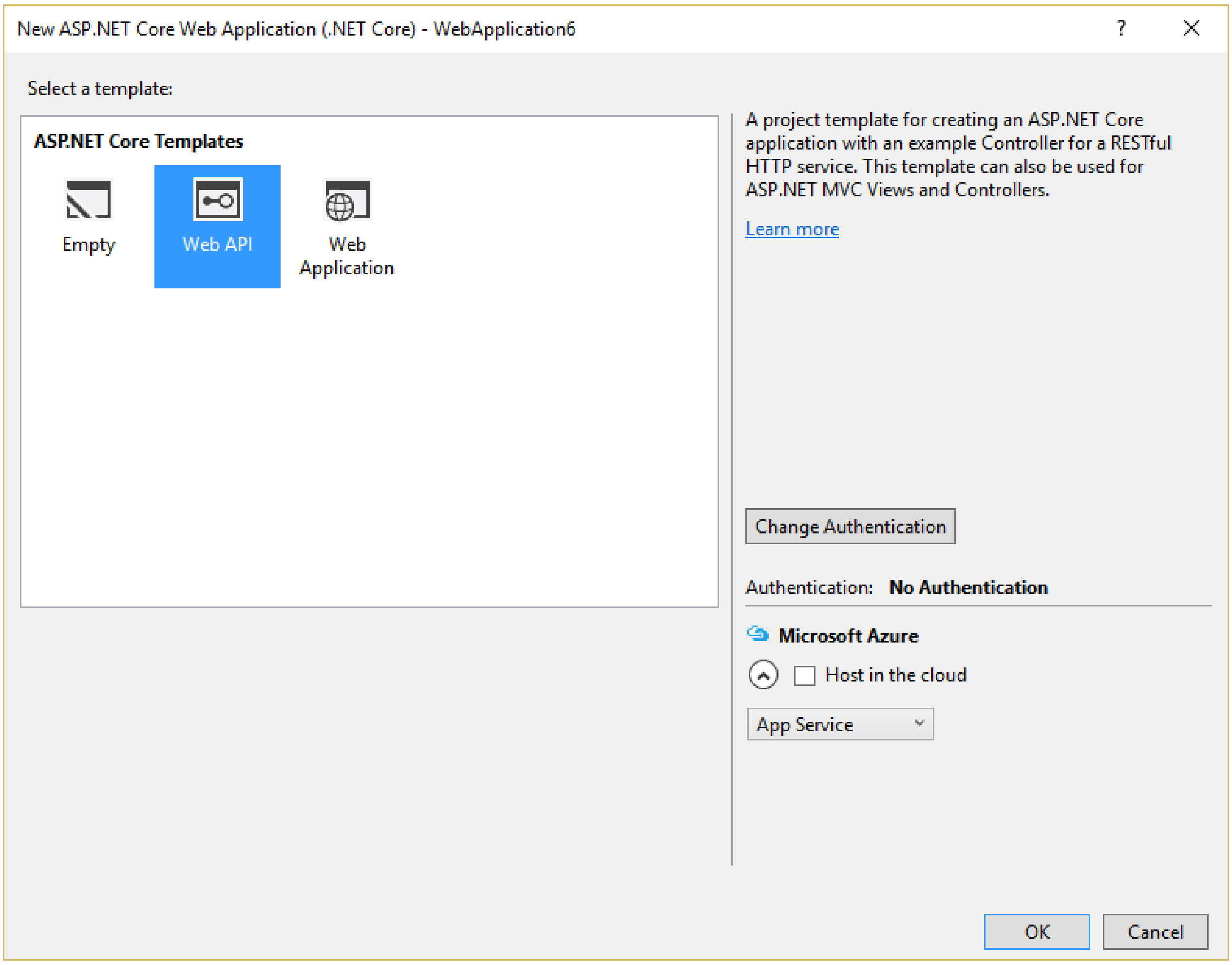


This sample is configured by default to use backend services hosted at developer.xamarin.com, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the *Constants.cs* file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// 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}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include `.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Note

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```
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; }
    }
}
```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```
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);
    }
}
```

For this sample, the implementation just uses a private collection of items:

```
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);
        }
    }
}
```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository,ToDoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

> 迷 Tip
>
> Learn more about creating web APIs in Building Your First Web API with ASP.NET Core MVC and Visual Studio.

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from Microsoft.AspNetCore.Mvc.Controller. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `IToDoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for dependency injection.

```
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;
        }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

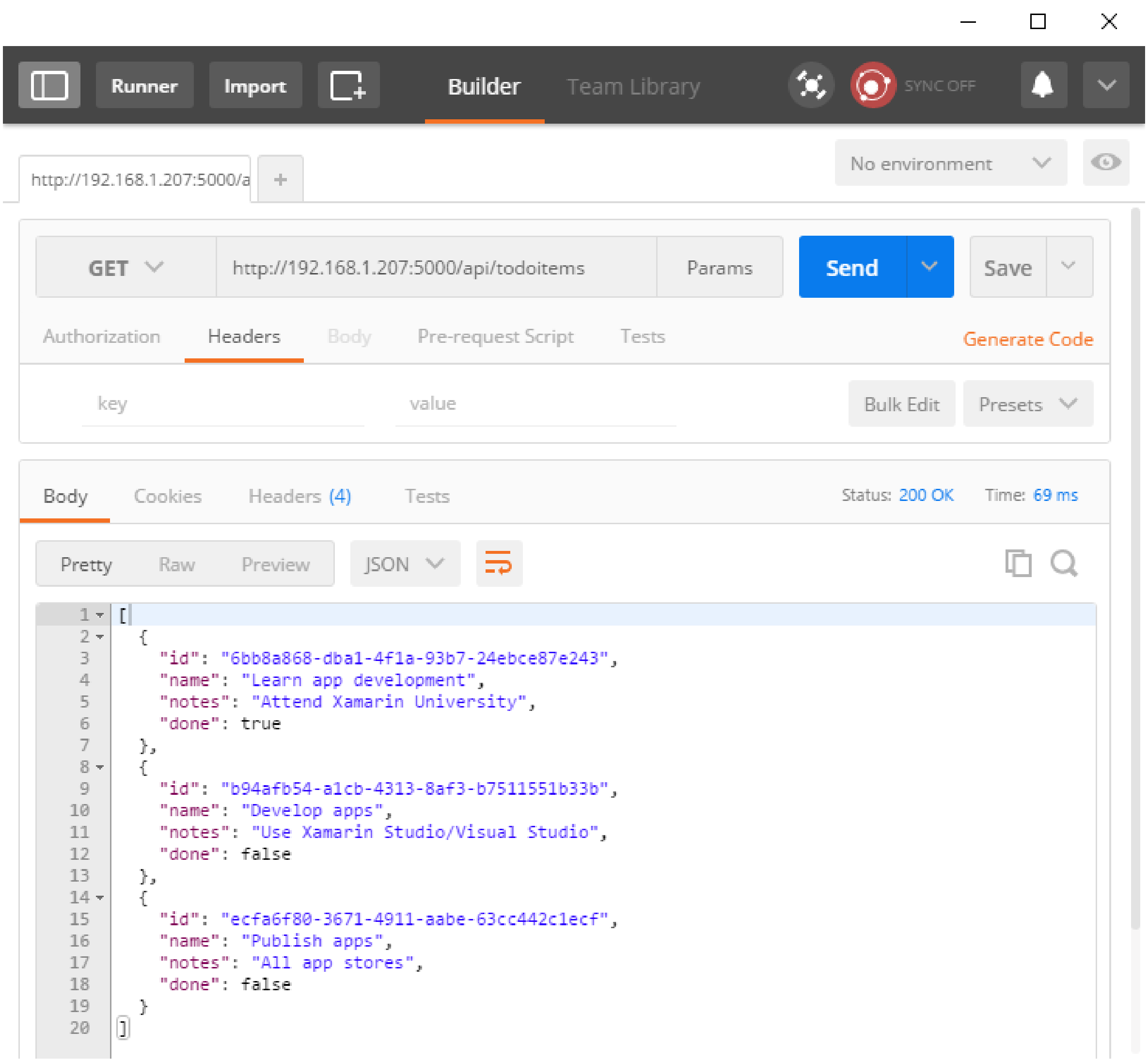
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_toDoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the ToDo items, serialized as JSON.

You can test your new API method using a variety of tools, such as Postman, shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts an ID parameter and a `ToDoItem` instance. The HTTP verb attributes, like `[HttpPost]`, optionally accept a route template string (`{id}` in this example). This has the same effect as adding a `[Route]` attribute to the action. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

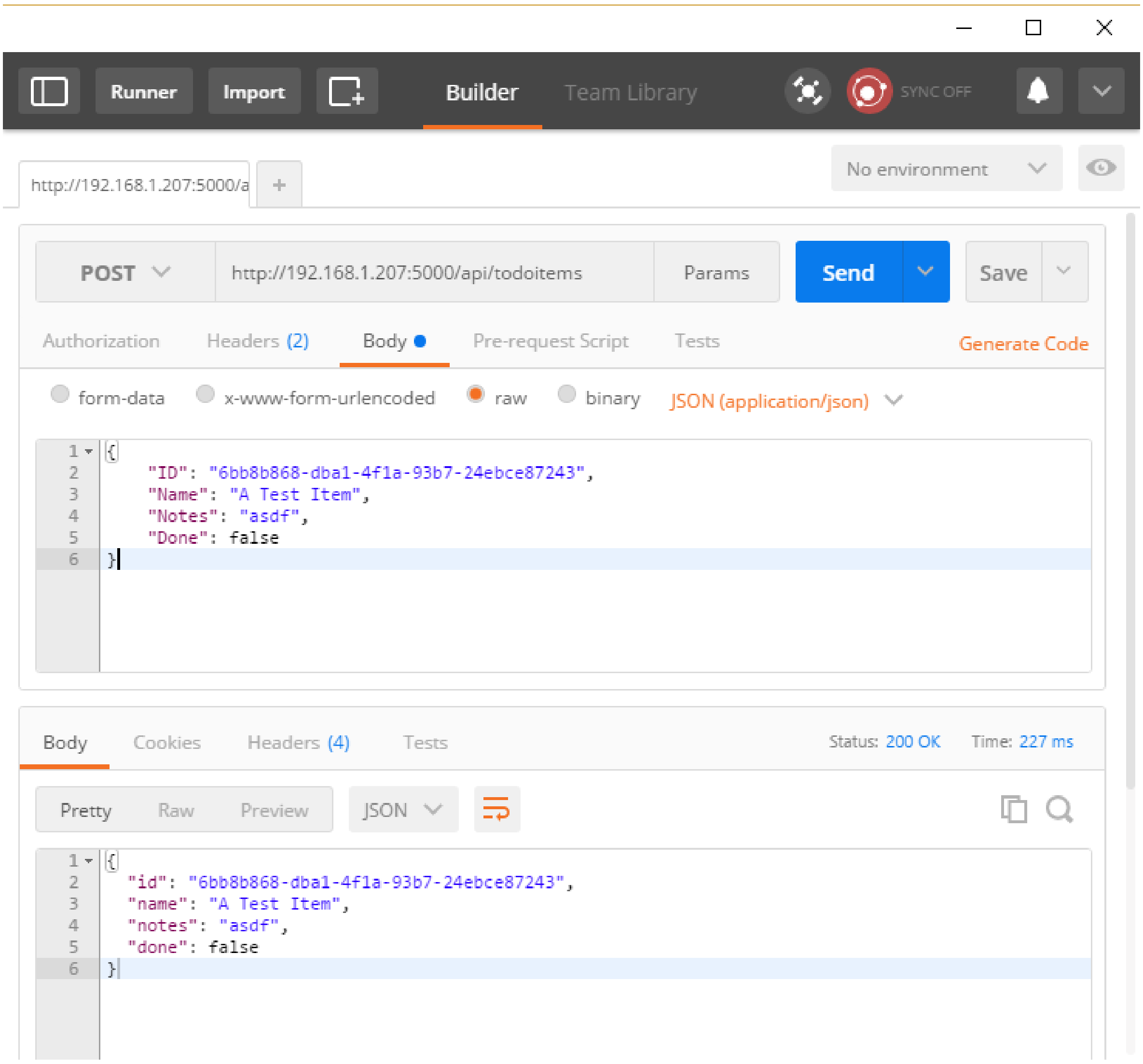
Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

```
[HttpPost("{id}")]
public IActionResult Create(string id, [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);
}
```

The sample uses an enum containing error codes that are passed to the mobile client:

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.



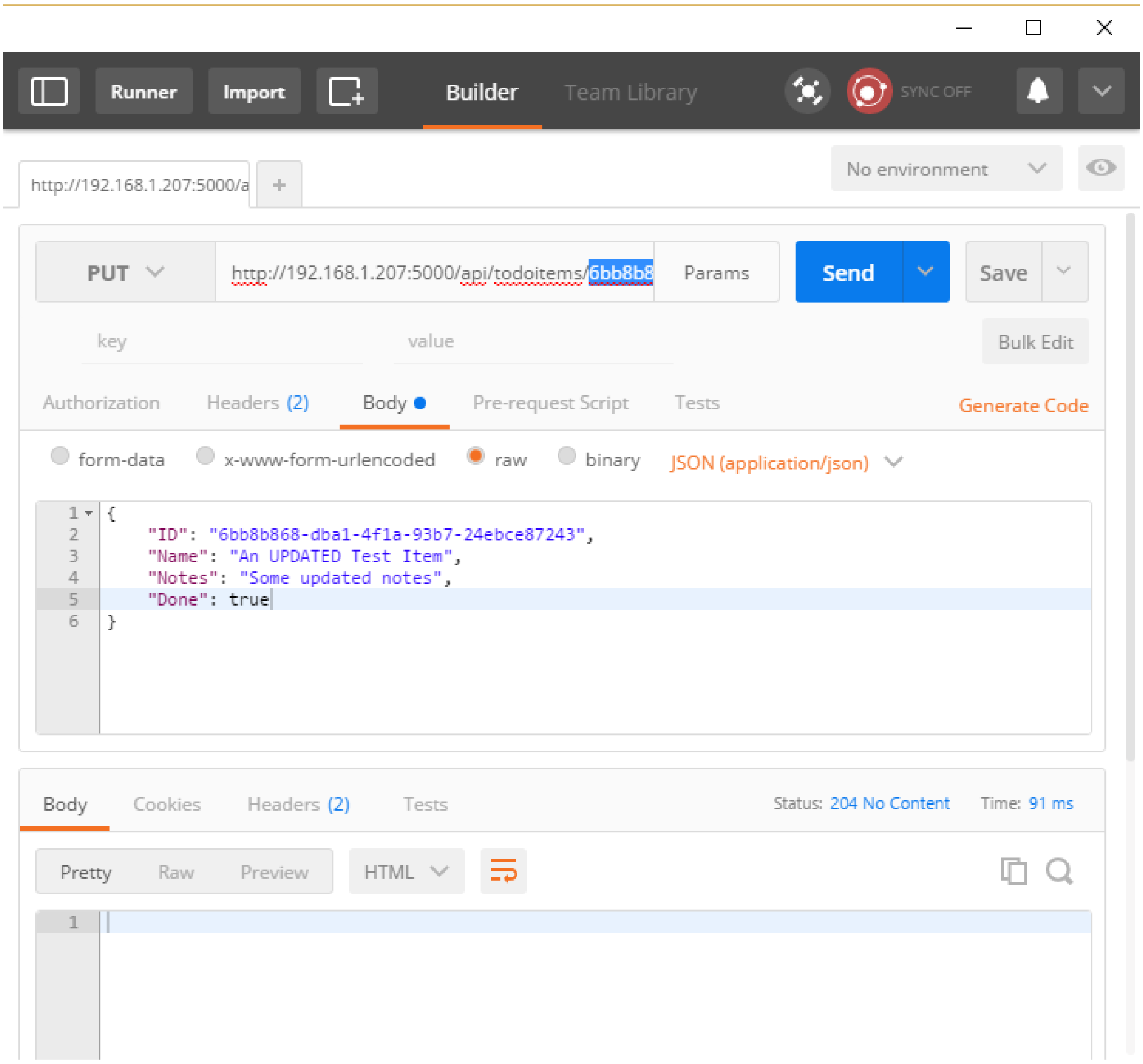
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut("{id}")]
public IActionResult Edit(string id, [FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _toDoRepository.Find(id);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT and add the ID of the record being updated to the URL. Specify the updated object data in the Body of the request.



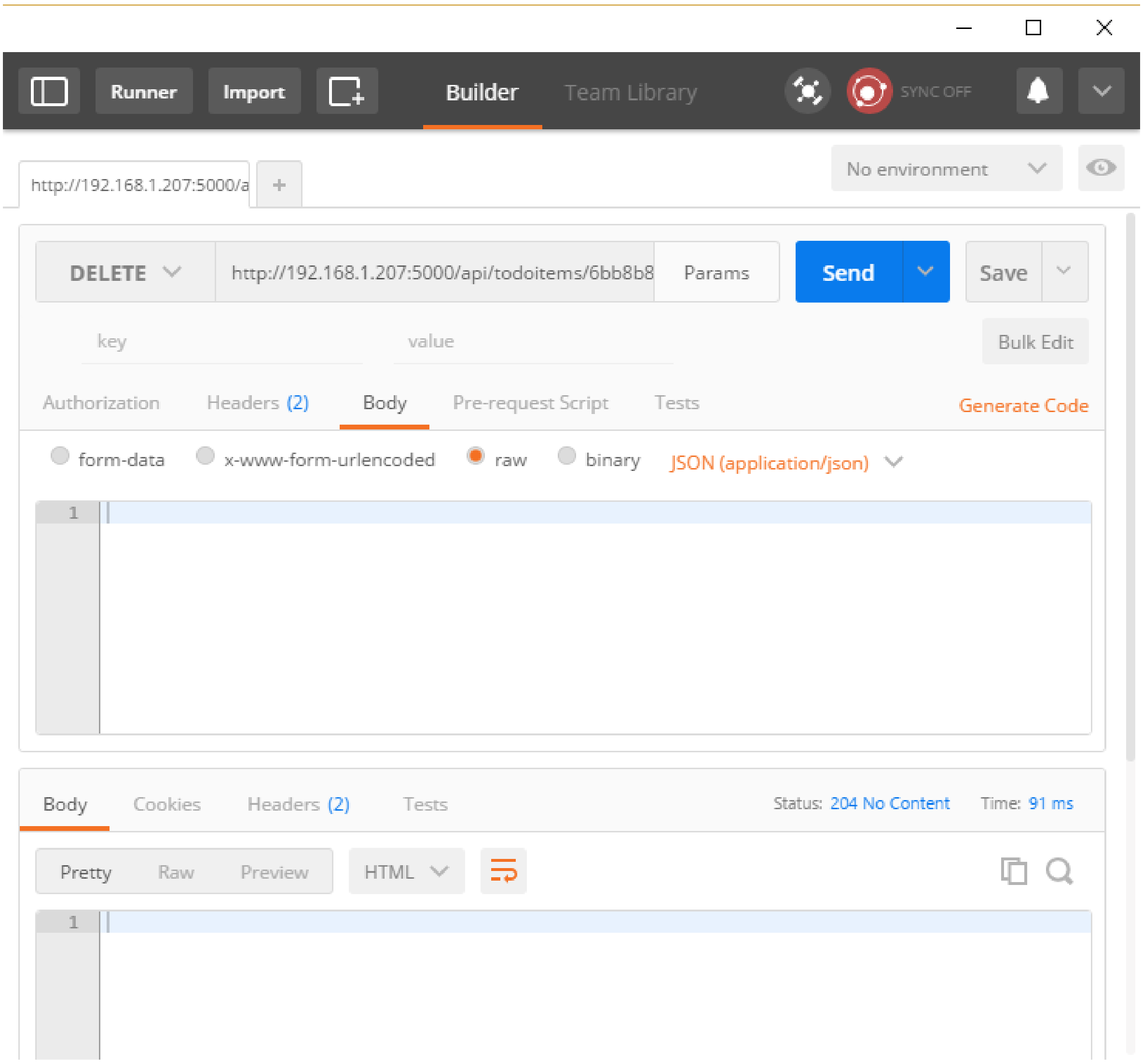
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[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();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Introduction to formatting response data in ASP.NET Core MVC

By Steve Smith

ASP.NET Core MVC has built-in support for formatting response data, using fixed formats or in response to client specifications.

View or download sample from GitHub.

Format-Specific Action Results

Some action result types are specific to a particular format, such as `JsonResult` and `ContentResult`. Actions can return specific results that are always formatted in a particular manner. For example, returning a `JsonResult` will return JSON-formatted data, regardless of client preferences. Likewise, returning a `ContentResult` will return plain-text-formatted string data (as will simply returning a string).

Note

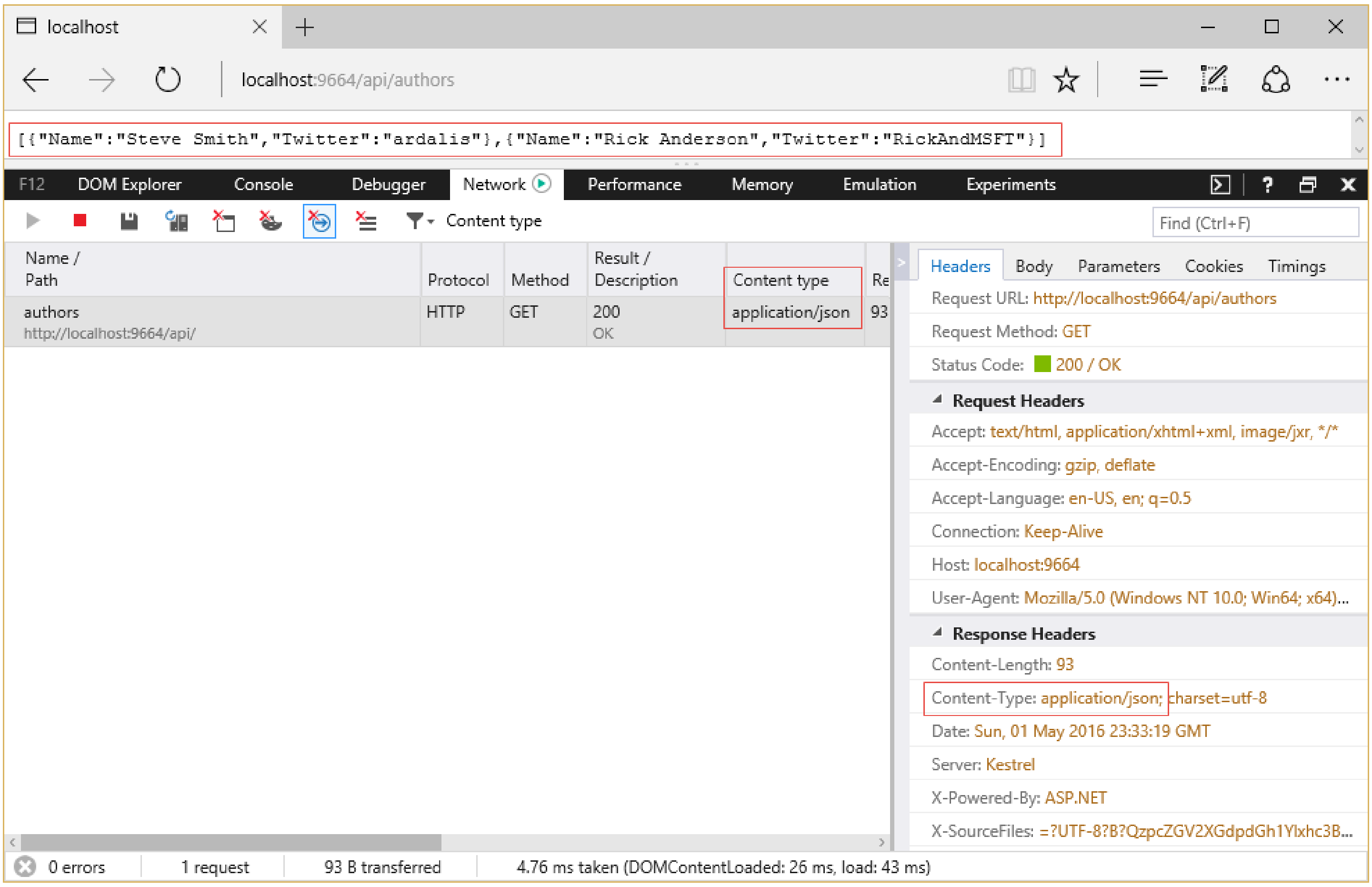
An action isn't required to return any particular type; MVC supports any object return value. If an action returns an `IActionResult` implementation and the controller inherits from `Controller`, developers have many helper methods corresponding to many of the choices. Results from actions that return objects that are not `IActionResult` types will be serialized using the appropriate `IOutputFormatter` implementation.

To return data in a specific format from a controller that inherits from the `Controller` base class, use the built-in helper method `Json` to return JSON and `Content` for plain text. Your action method should return either the specific result type (for instance, `JsonResult`) or `IActionResult`.

Returning JSON-formatted data:

```
// GET: api/authors
[HttpGet]
public JsonResult Get()
{
    return Json(_authorRepository.List());
}
```

Sample response from this action:

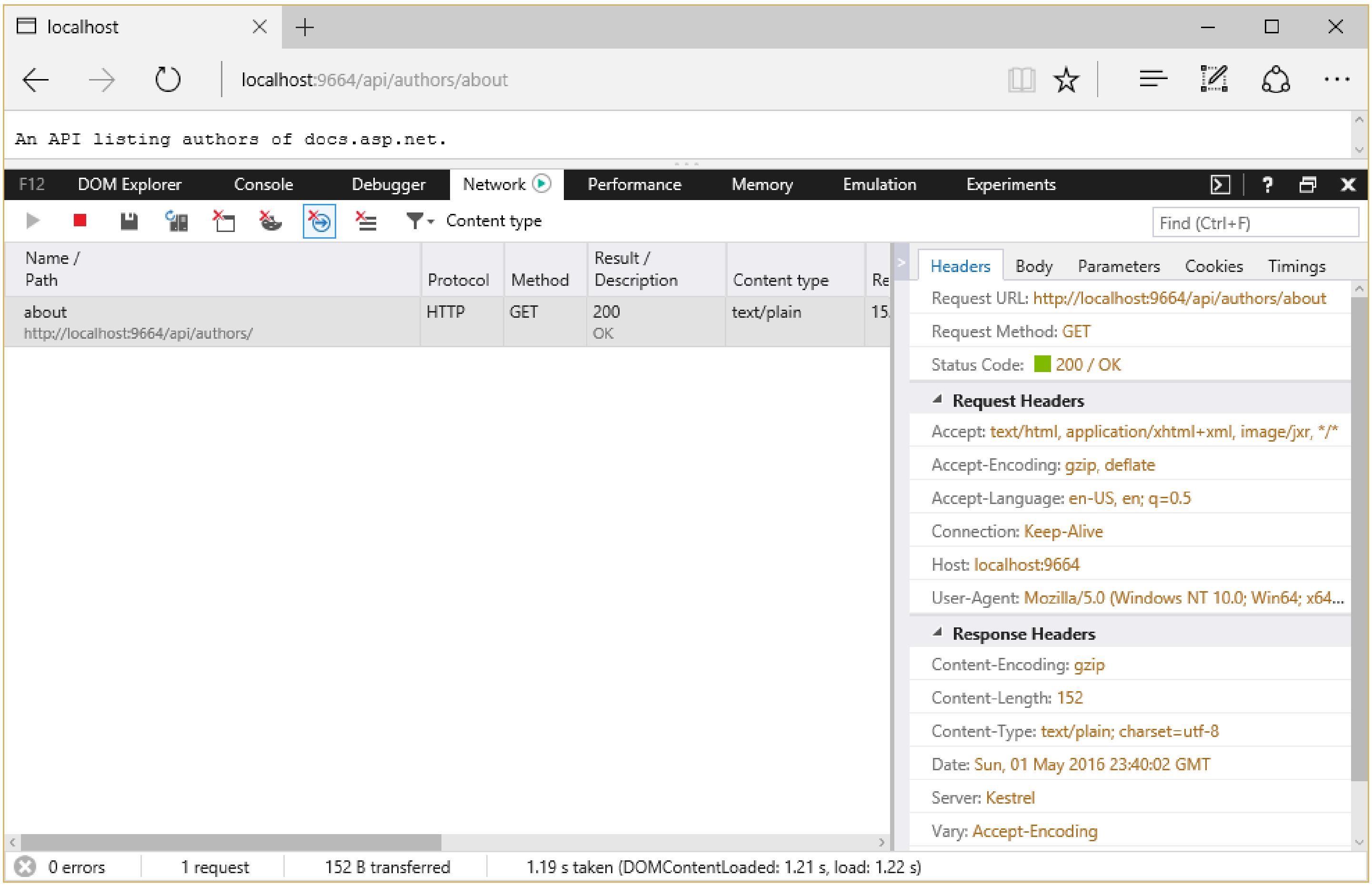


Note that the content type of the response is `application/json`, shown both in the list of network requests and in the Response Headers section. Also note the list of options presented by the browser (in this case, Microsoft Edge) in the Accept header in the Request Headers section. The current technique is ignoring this header; obeying it is discussed below.

To return plain text formatted data, use `ContentResult` and the `Content` helper:

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

A response from this action:



Note in this case the `Content-Type` returned is `text/plain`. You can also achieve this same behavior using just a string response type:

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

 Tip

For non-trivial actions with multiple return types or options (for example, different HTTP status codes based on the result of operations performed), prefer `IActionResult` as the return type.

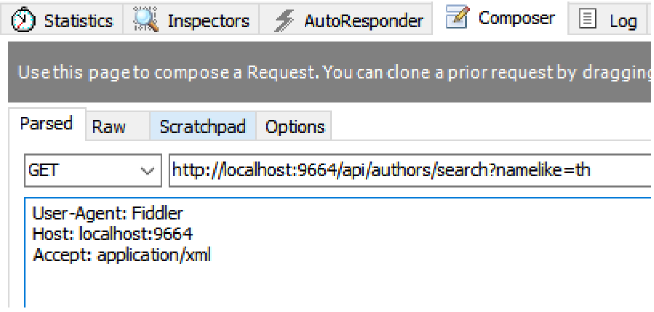
Content Negotiation

Content negotiation (*conneg* for short) occurs when the client specifies an Accept header. The default format used by ASP.NET Core MVC is JSON. Content negotiation is implemented by `ObjectResult`. It is also built into the status code specific action results returned from the helper methods (which are all based on `ObjectResult`). You can also return a model type (a class you've defined as your data transfer type) and the framework will automatically wrap it in an `ObjectResult` for you.

The following action method uses the `Ok` and `NotFound` helper methods:

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authorRepository.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

A JSON-formatted response will be returned unless another format was requested and the server can return the requested format. You can use a tool like Fiddler to create a request that includes an Accept header and specify another format. In that case, if the server has a *formatter* that can produce a response in the requested format, the result will be returned in the client-preferred format.



In the above screenshot, the Fiddler Composer has been used to generate a request, specifying `Accept: application/xml`. By default, ASP.NET Core MVC only supports JSON, so even when another format is specified, the result returned is still JSON-formatted. You'll see how to add additional formatters in the next section.

Controller actions can return POCOs (Plain Old CLR Objects), in which case ASP.NET MVC will automatically create an `ObjectResult` for you that wraps the object. The client will get the formatted serialized object (JSON format is the default; you can configure XML or other formats). If the object being returned is `null`, then the framework will return a `204 No Content` response.

Returning an object type:

```
// GET api/authors/ardalis
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authorRepository.GetByAlias(alias);
}
```

In the sample, a request for a valid author alias will receive a 200 OK response with the author's data. A request for an invalid alias will receive a 204 No Content response. Screenshots showing the response in XML and JSON formats are shown below.

Content Negotiation Process

Content *negotiation* only takes place if an `Accept` header appears in the request. When a request contains an accept header, the framework will enumerate the media types in the accept header in preference order and will try to find a formatter that can produce a response in one of the formats specified by the accept header. In case no formatter is found that can satisfy the client's request, the framework will try to find the first formatter that can produce a response (unless the developer has configured the option on `MvcOptions` to return 406 Not Acceptable instead). If the request specifies XML, but the XML formatter has not been configured, then the JSON formatter will be used. More generally, if no formatter is configured that can provide the requested format, then the first formatter than can format the object is used. If no header is given, the first formatter that can handle the object to be returned will be used to serialize the response. In this case, there isn't any negotiation taking place - the server is determining what format it will use.

Note

If the Accept header contains `*/*`, the Header will be ignored unless `RespectBrowserAcceptHeader` is set to true on `MvcOptions`.

Browsers and Content Negotiation

Unlike typical API clients, web browsers tend to supply `Accept` headers that include a wide array of formats, including wildcards. By default, when the framework detects that the request is coming from a browser, it will ignore the `Accept` header and instead return the content in the application's configured default format (JSON unless otherwise configured). This provides a more consistent experience when using different browsers to consume APIs.

If you would prefer your application honor browser accept headers, you can configure this as part of MVC's configuration by setting `RespectBrowserAcceptHeader` to `true` in the `ConfigureServices` method in *Startup.cs*.

```
services.AddMvc(options =>
{
  options.RespectBrowserAcceptHeader = true; // false by default
}
```

Configuring Formatters

If your application needs to support additional formats beyond the default of JSON, you can add NuGet packages and configure MVC to support them. There are separate formatters for input and output. Input formatters are used by Model Binding; output formatters are used to format responses. You can also configure Custom Formatters.

Adding XML Format Support

To add support for XML formatting, install the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package.

Add the XmlSerializerFormatters to MVC's configuration in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        //set global format to xml
        options.Filters.Add(new ProducesAttribute("application/xml"));
    })
```

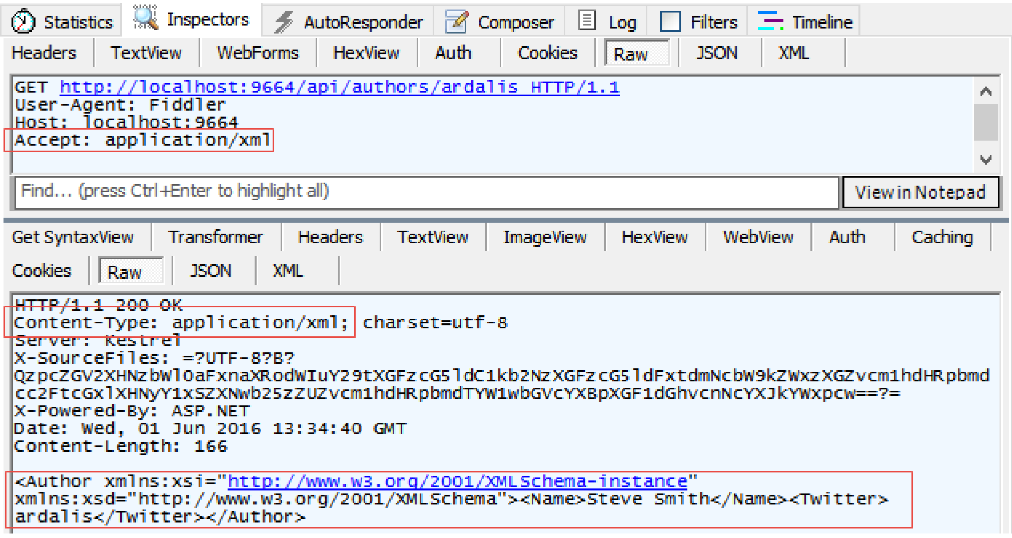
Alternately, you can add just the output formatter:

```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
});
```

These two approaches will serialize results using `System.Xml.Serialization.XmlSerializer`. If you prefer, you can use the `System.Runtime.Serialization.DataContractSerializer` by adding its associated formatter:

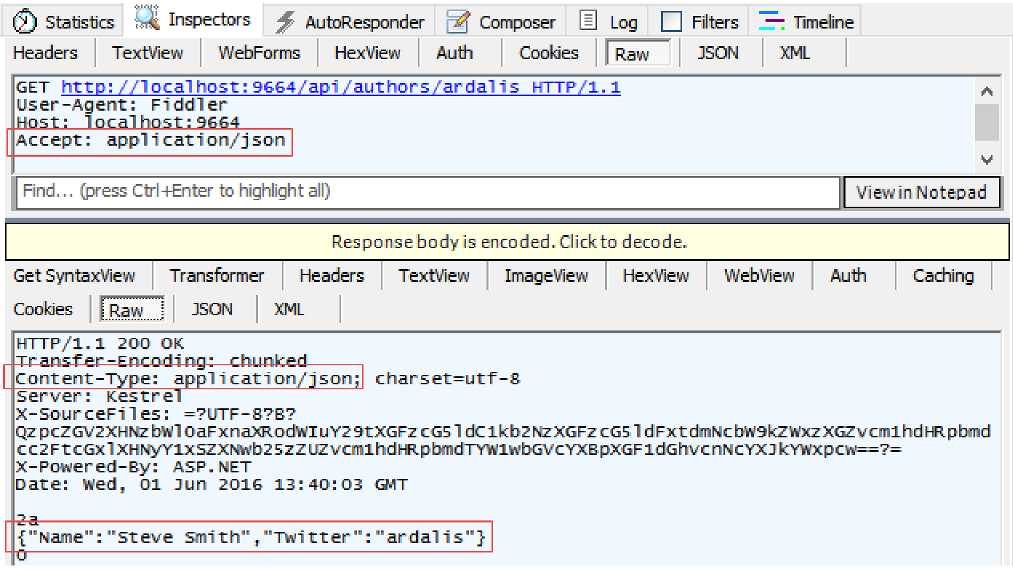
```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});
```

Once you've added support for XML formatting, your controller methods should return the appropriate format based on the request's `Accept` header, as this Fiddler example demonstrates:



You can see in the Inspectors tab that the Raw GET request was made with an `Accept: application/xml` header set. The response pane shows the `Content-Type: application/xml` header, and the `Author` object has been serialized to XML.

Use the Composer tab to modify the request to specify `application/json` in the `Accept` header. Execute the request, and the response will be formatted as JSON:



In this screenshot, you can see the request sets a header of `Accept: application/json` and the response specifies the same as its `Content-Type`. The `Author` object is shown in the body of the response, in JSON format.

Forcing a Particular Format

If you would like to restrict the response formats for a specific action you can, you can apply the `[Produces]` filter. The `[Produces]` filter specifies the response formats for a specific action (or controller). Like most Filters, this can be applied at the action, controller, or global scope.

```
[Produces("application/json")]
public class AuthorsController
```

The `[Produces]` filter will force all actions within the `AuthorsController` to return JSON-formatted responses, even if other formatters were configured for the application and the client provided an `Accept` header requesting a different, available format. See Filters to learn more, including how to apply filters globally.

Special Case Formatters

Some special cases are implemented using built-in formatters. By default, `string` return types will be formatted as *text/plain* (*text/html* if requested via `Accept` header). This behavior can be removed by removing the `TextOutputFormatter`. You remove formatters in the `Configure` method in *Startup.cs* (shown below). Actions that have a model object return type will return a 204 No Content response when returning `null`. This behavior can be removed by removing the `HttpNoContentOutputFormatter`. The following code removes the `TextOutputFormatter` and `HttpNoContentOutputFormatter`.

```
services.AddMvc(options =>
{
  options.OutputFormatters.RemoveType<TextOutputFormatter>();
  options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

Without the `TextOutputFormatter`, `string` return types return 406 Not Acceptable, for example. Note that if an XML formatter exists, it will format `string` return types if the `TextOutputFormatter` is removed.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example, the JSON formatter will simply return a response with a body of `null`, while the XML formatter will return an empty XML element with the attribute `xsi:nil="true"` set.

Response Format URL Mappings

Clients can request a particular format as part of the URL, such as in the query string or part of the path, or by using a format-specific file extension such as .xml or .json. The mapping from request path should be specified in the route the API is using. For example:

```
[FormatFilter]
public class ProductsController
{
  [Route("[controller]/[action]/{id}.{format?}")]
  public Product GetById(int id)
```

This route would allow the requested format to be specified as an optional file extension. The `[FormatFilter]` attribute checks for the existence of the format value in the `RouteData` and will map the response format to the appropriate formatter when the response is created.

ROUTE	FORMATTER
`/products/GetById/5`	The default output formatter
`/products/GetById/5.json`	The JSON formatter (if configured)
`/products/GetById/5.xml`	The XML formatter (if configured)

Custom formatters in ASP.NET Core MVC web APIs

By Tom Dykstra

ASP.NET Core MVC has built-in support for data exchange in web APIs by using JSON, XML, or plain text formats. This article shows how to add support for additional formats by creating custom formatters.

View or download sample from GitHub.

When to use custom formatters

Use a custom formatter when you want the content negotiation process to support a content type that isn't supported by the built-in formatters (JSON, XML, and plain text).

For example, if some of the clients for your web API can handle the Protobuf format, you might want to use Protobuf with those clients because it's more efficient. Or you might want your web API to send contact names and addresses in vCard format, a commonly used format for exchanging contact data. The sample app provided with this article implements a simple vCard formatter.

Overview of how to use a custom formatter

Here are the steps to create and use a custom formatter:

Create an output formatter class if you want to serialize data to send to the client.
Create an input formatter class if you want to deserialize data received from the client.
Add instances of your formatters to the `InputFormatters` and `OutputFormatters` collections in MvcOptions.

The following sections provide guidance and code examples for each of these steps.

How to create a custom formatter class

To create a formatter:

Derive the class from the appropriate base class.
Specify valid media types and encodings in the constructor.
Override `CanReadType`/`CanWriteType` methods
Override `ReadRequestBodyAsync`/`WriteResponseBodyAsync` methods

Derive from the appropriate base class

For text media types (for example, vCard), derive from the TextInputFormatter or TextOutputFormatter base class.

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the InputFormatter or OutputFormatter base class.

Specify valid media types and encodings

In the constructor, specify valid media types and encodings by adding to the `SupportedMediaTypes` and `SupportedEncodings` collections.

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

Note

You can't do constructor dependency injection in a formatter class. For example, you can't get a logger by adding a logger parameter to the constructor. To access services, you have to use the context object that gets passed in to your methods. A code example below shows how to do this.

Override CanReadType/CanWriteType

Specify the type you can deserialize into or serialize from by overriding the `CanReadType` or `CanWriteType` methods. For example, you might only be able to create vCard text from a `Contact` type and vice versa.

```
protected override bool CanWriteType(Type type)
{
    if (typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type))
    {
        return base.CanWriteType(type);
    }
    return false;
}
```

The CanWriteResult method

In some scenarios you have to override `CanWriteResult` instead of `CanWriteType`. Use `CanWriteResult` if the following conditions are true:

Your action method returns a model class.
There are derived classes which might be returned at runtime.
You need to know at runtime which derived class was returned by the action.

For example, suppose your action method signature returns a `Person` type, but it may return a `Student` or `Instructor` type that derives from `Person`. If you want your formatter to handle only `Student` objects, check the type of Object in the context object provided to the `CanWriteResult` method. Note that it's not necessary to use `CanWriteResult` when the action method returns `IActionResult`; in that case, the `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync/WriteResponseBodyAsync

You do the actual work of deserializing or serializing in `ReadRequestBodyAsync` or `WriteResponseBodyAsync`. The highlighted lines in the following example show how to get services from the dependency injection container (you can't get them from constructor parameters).

```
public override Task WriteResponseBodyAsync(OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    IServiceProvider serviceProvider = context.HttpContext.RequestServices;
    var logger = serviceProvider.GetService(typeof(ILogger<VcardOutputFormatter>)) as ILogger;

    var response = context.HttpContext.Response;

    var buffer = new StringBuilder();
    if (context.Object is IEnumerable<Contact>)
    {
        foreach (Contact contact in context.Object as IEnumerable<Contact>)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        var contact = context.Object as Contact;
        FormatVcard(buffer, contact, logger);
    }
    return response.WriteAsync(buffer.ToString());
}

private static void FormatVcard(StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendFormat($"N:{contact.LastName};{contact.FirstName}\r\n");
    buffer.AppendFormat($"FN:{contact.FirstName} {contact.LastName}\r\n");
    buffer.AppendFormat($"UID:{contact.ID}\r\n");
    buffer.AppendLine("END:VCARD");
    logger.LogInformation($"Writing {contact.FirstName} {contact.LastName}");
}
```

How to configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the `InputFormatters` or `OutputFormatters` collection.

```
services.AddMvc(options =>
{
    options.InputFormatters.Add(new VcardInputFormatter());
    options.OutputFormatters.Add(new VcardOutputFormatter());
});
```

Next steps

See the sample application, which implements simple vCard input and output formatters. The application reads and writes vCards that look like the following example:

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
UID:20293482-9240-4d68-b475-325df4a83728
END:VCARD
```

To see vCard output, run the application and send a Get request with Accept header "text/vcard" to `http://localhost:63313/api/contacts/` (when running from Visual Studio) or `http://localhost:5000/api/contacts/` (when running from the command line).

To add a vCard to the in-memory collection of contacts, send a Post request to the same URL, with Content-Type header "text/vcard" and with vCard text in the body, formatted like the example above.

Filters

By Tom Dykstra and Steve Smith

Filters in ASP.NET Core MVC allow you to run code before or after certain stages in the request processing pipeline.

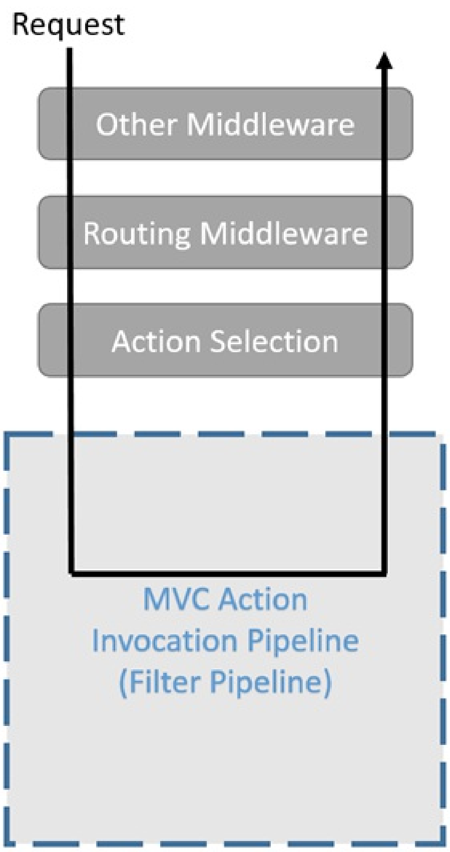
Built-in filters handle tasks such as authorization (preventing access to resources a user isn't authorized for), ensuring that all requests use HTTPS, and response caching (short-circuiting the request pipeline to return a cached response).

You can create custom filters to handle cross-cutting concerns for your application. Anytime you want to avoid duplicating code across actions, filters are the solution. For example, you can consolidate error handling code in a exception filter.

View or download sample from GitHub.

How do filters work?

Filters run within the *MVC action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after MVC selects the action to execute.



Filter types

Each filter type is executed at a different stage in the filter pipeline.

Authorization filters run first and are used to determine whether the current user is authorized for the current request. They can short-circuit the pipeline if a request is unauthorized.

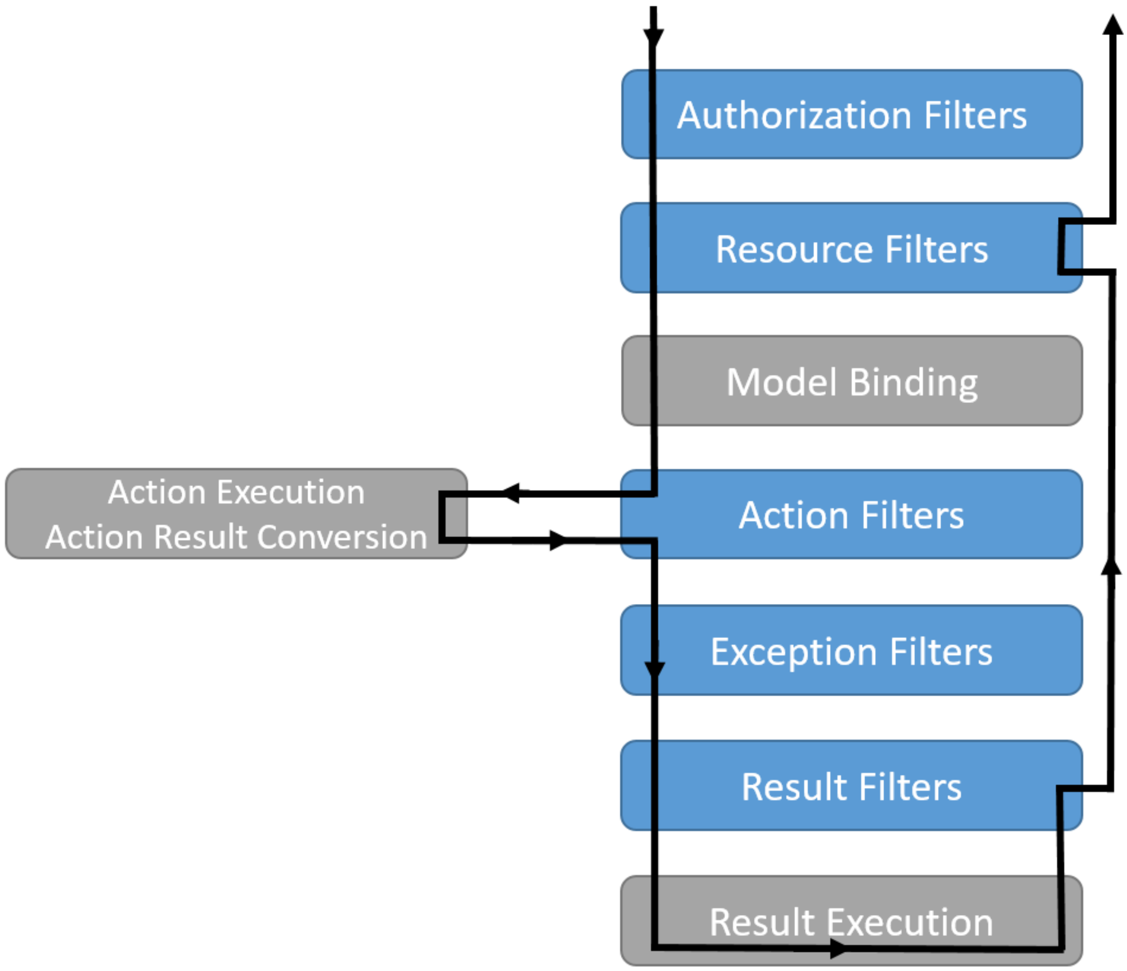
Resource filters are the first to handle a request after authorization. They can run code before the rest of the filter pipeline, and after the rest of the pipeline has completed. They're useful to implement caching or otherwise short-circuit the filter pipeline for performance reasons. Since they run before model binding, they're useful for anything that needs to influence model binding.

Action filters can run code immediately before and after an individual action method is called. They can be used to manipulate the arguments passed into an action and the result returned from the action.

Exception filters are used to apply global policies to unhandled exceptions that occur before anything has been written to the response body.

Result filters can run code immediately before and after the execution of individual action results. They run only when the action method has executed successfully and are useful for logic that must surround view or formatter execution.

The following diagram shows how these filter types interact in the filter pipeline.



Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions. Choose either the sync or async variant depending on the kind of task you need to perform.

Synchronous filters that can run code both before and after their pipeline stage define On*Stage*Executing and On*Stage*Executed methods. For example, `OnActionExecuting` is called before the action method is called, and `OnActionExecuted` is called after the action method returns.

```
using FiltersSample.Helper;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    #region snippet_ActionFilter
    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
        }
    }
    #endregion
}
```

Asynchronous filters define a single On*Stage*ExecutionAsync method. This method takes a *FilterType*ExecutionDelegate delegate which executes the filter's pipeline stage. For example, `ActionExecutionDelegate` calls the action method, and you can execute code before and after you call it.

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace FiltersSample.Filters
{
    public class SampleAsyncActionFilter : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(
            ActionExecutingContext context,
            ActionExecutionDelegate next)
        {
            // do something before the action executes
            await next();
            // do something after the action executes
        }
    }
}
```

You can implement interfaces for multiple filter stages in a single class. For example, the ActionFilterAttribute abstract class implements both `IActionFilter` and `IResultFilter`, as well as their async equivalents.

Note

Implement **either** the synchronous or the async version of a filter interface, not both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If you were to implement both interfaces on one class, only the async method would be called. When using abstract classes like ActionFilterAttribute you would override only the synchronous methods or the async method for each filter type.

IFilterFactory

`IFilterFactory` implements `IFilter`. Therefore, an `IFilterFactory` instance can be used as an `IFilter` instance anywhere in the filter pipeline. When the framework prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the `CreateInstance` method is called to create the `IFilter` instance that will be invoked. This provides a very flexible design, since the precise filter pipeline does not need to be set explicitly when the application starts.

You can implement `IFilterFactory` on your own attribute implementations as another approach to creating filters:

```
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;
        }
    }
}
```

Built-in filter attributes

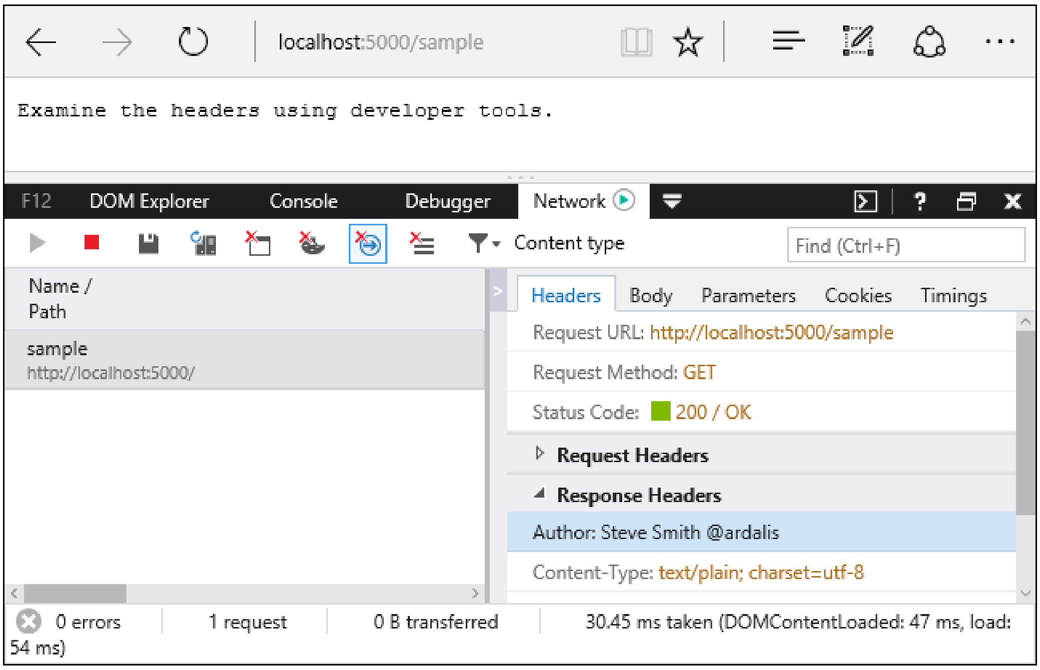
The framework includes built-in attribute-based filters that you can subclass and customize. For example, the following Result filter adds a header to the response.

```
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);
        }
    }
}
```

Attributes allow filters to accept arguments, as shown in the example above. You would add this attribute to a controller or action method and specify the name and value of the HTTP header:

```
[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.");
    }
```

The result of the `Index` action is shown below - the response headers are displayed on the bottom right.



Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

`ActionFilterAttribute`
`ExceptionFilterAttribute`
`ResultFilterAttribute`
`FormatFilterAttribute`
`ServiceFilterAttribute`
`TypeFilterAttribute`

`TypeFilterAttribute` and `ServiceFilterAttribute` are explained later in this article.

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*. You can add a filter to a particular action method or to a controller class by using an attribute. Or you can register a filter globally (for all controllers and actions) by adding it to the `MvcOptions.Filters` collection in the `ConfigureServices` method in the `Startup` class:

```
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>();
}
```

Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters. This is sometimes referred to as "Russian doll" nesting, as each increase in scope is wrapped around the previous scope, like a nesting doll. You generally get the desired overriding behavior without having to explicitly determine ordering.

Asa result of this nesting, the *after* code of filters runs in the reverse order of the *before* code. The sequence looks like this:

The *before* code of filters applied globally
The *before* code of filters applied to controllers
The *before* code of filters applied to action methods
The *after* code of filters applied to action methods

The *after* code of filters applied to controllers

The *after* code of filters applied globally

Here's an example that illustrates the order in which filter methods are called for synchronous Action filters.

SEQUENCE	FILTER SCOPE	FILTER METHOD
1	Global	`OnActionExecuting`
2	Controller	`OnActionExecuting`
3	Method	`OnActionExecuting`
4	Method	`OnActionExecuted`
5	Controller	`OnActionExecuted`
6	Global	`OnActionExecuted`

This sequence shows that the method filter is nested within the controller filter, and the controller filter is nested within the global filter. To put it another way, if you're inside an async filter's On*Stage*ExecutionAsync method, all of the filters with a tighter scope run while your code is on the stack.

Note

Every controller that inherits from the `Controller` base class includes `OnActionExecuting` and `OnActionExecuted` methods. These methods wrap the filters that run for a given action: `OnActionExecuting` is called before any of the filters, and `OnActionExecuted` is called after all of the filters.

Overriding the default order

You can override the default sequence of execution by implementing `IOrderedFilter`. This interface exposes an `Order` property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value will have its *before* code executed before that of a filter with a higher value of `Order`. A filter with a lower `Order` value will have its *after* code executed after that of a filter with a higher `Order` value. You can set the `Order` property by using a constructor parameter:

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

If you have the same 3 Action filters shown in the preceding example but set the `Order` property of the controller and global filters to 1 and 2 respectively, the order of execution would be reversed.

SEQUENCE	FILTER SCOPE	`ORDER` PROPERTY	FILTER METHOD
1	Method	0	`OnActionExecuting`
2	Controller	1	`OnActionExecuting`
3	Global	2	`OnActionExecuting`
4	Global	2	`OnActionExecuted`
5	Controller	1	`OnActionExecuted`
6	Method	0	`OnActionExecuted`

The `Order` property trumps scope when determining the order in which filters will run. Filters are sorted first by order, then scope is used to break ties. All of the built-in filters implement `IOrderedFilter` and set the default `Order` value to 0, so scope determines order unless you set `Order` to a non-zero value.

Cancellation and short circuiting

You can short-circuit the filter pipeline at any point by setting the `Result` property on the `context` parameter provided to the filter method. For instance, the following Resource filter prevents the rest of the pipeline from executing.

```
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)
        {
        }
    }
}
```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. However, because the `ShortCircuitingResourceFilter` runs first and short-circuits the rest of the pipeline, the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first.

```
[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.");
    }
```

Dependency injection

Filters can be added by type or by instance. If you add an instance, that instance will be used for every request. If you add a type, it will be type-activated, meaning an instance will be created for each request and any constructor dependencies will be populated by dependency injection (DI). Adding a filter by type is equivalent to `filters.Add(new TypeFilterAttribute(typeof(MyFilter)))`.

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by dependency injection (DI). This is because attributes must have their constructor parameters supplied where they are applied. This is a limitation of how attributes work.

If your filters have dependencies that you need to access from DI, there are several supported approaches. You can apply your filter to a class or action method using one of the following:

`ServiceFilterAttribute`
`TypeFilterAttribute`
`IFilterFactory` implemented on your attribute

Note

One dependency you might want to get from DI is a logger. However, avoid creating and using filters purely for logging purposes, since the built-in framework logging features may already provide what you need. If you're going to add logging to your filters, it should focus on business domain concerns or behavior specific to your filter, rather than MVC actions or other framework events.

ServiceFilterAttribute

A `ServiceFilter` retrieves an instance of the filter from DI. You add the filter to the container in `ConfigureServices`, and reference it in a `ServiceFilter` attribute

```
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();
}
```

Using `ServiceFilter` without registering the filter type results in an exception:

```
System.InvalidOperationException: No service for type
'FiltersSample.Filters.AddHeaderFilterWithDI' has been registered.
```

`ServiceFilterAttribute` implements `IFilterFactory`, which exposes a single method for creating an `IFilter` instance. In the case of `ServiceFilterAttribute`, the `IFilterFactory` interface's `CreateInstance` method is implemented to load the specified type from the services container (DI).

TypeFilterAttribute

`TypeFilterAttribute` is very similar to `ServiceFilterAttribute` (and also implements `IFilterFactory`), but its type is not resolved directly from the DI container. Instead, it instantiates the type by using `Microsoft.Extensions.DependencyInjection.ObjectFactory`.

Because of this difference, types that are referenced using the `TypeFilterAttribute` do not need to be registered with the container first (but they will still have their dependencies fulfilled by the container). Also, `TypeFilterAttribute` can optionally accept constructor arguments for the type in question. The following example demonstrates how to pass arguments to a type using `TypeFilterAttribute`:

```
[TypeFilter(typeof(AddHeaderAttribute),
    Arguments = new object[] { "Author", "Steve Smith (@ardalis)" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

If you have a filter that doesn't require any arguments, but which has constructor dependencies that need to be filled by DI, you can use your own named attribute on classes and methods instead of `[TypeFilter(typeof(FilterType))]`. The following filter shows how this can be implemented:

```
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.");
        }
    }
}
```

This filter can be applied to classes or methods using the `[SampleActionFilter]` syntax, instead of having to use `[TypeFilter]` or `[ServiceFilter]`.

Authorization filters

Authorization filters control access to action methods and are the first filters to be executed within the filter pipeline. They have only a before method, unlike most filters that support before and after methods. You should only write a custom authorization filter if you are writing your own authorization framework. Prefer configuring your authorization policies or writing a custom authorization policy over writing a custom filter. The built-in filter implementation is just responsible for calling the authorization system.

Note that you should not throw exceptions within authorization filters, since nothing will handle the exception (exception filters won't handle them). Instead, issue a challenge or find another way.

Learn more about Authorization.

Resource filters

Resource filters implement either the `IResourceFilter` or `IAsyncResourceFilter` interface, and their execution wraps most of the filter pipeline. (Only Authorization filters run before them.) Resource filters are especially useful if you need to short-circuit most of the work a request is doing. For example, a caching filter can avoid the rest of the pipeline if the response is already in the cache.

The short circuiting resource filter shown earlier is one example of a resource filter. Another example is DisableFormValueModelBindingAttribute, which prevents model binding from accessing the form data. It's useful for cases where you know that you're going to receive large file uploads and want to prevent the form from being read into memory.

Action filters

Action filters implement either the `IActionFilter` or `IAsyncActionFilter` interface, and their execution surrounds the execution of action methods.

Here's a sample action filter:

```
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
    }
}
```

The ActionExecutingContext provides the following properties:

`ActionArguments` - lets you manipulate the inputs to the action.
`Controller` - lets you manipulate the controller instance.
`Result` - setting this short-circuits execution of the action method and subsequent action filters. Throwing an exception also prevents execution of the action method and subsequent filters, but is treated as a failure instead of a successful result.

The ActionExecutedContext provides `Controller` and `Result` plus the following properties:

`Canceled` - will be true if the action execution was short-circuited by another filter.
`Exception` - will be non-null if the action or a subsequent action filter threw an exception. Setting this property to null effectively 'handles' an exception, and `Result` will be executed as if it were returned from the action method normally.

For an `IAsyncActionFilter`, a call to the `ActionExecutionDelegate` executes any subsequent action filters and the action method, returning an `ActionExecutedContext`. To short-circuit, assign `ActionExecutingContext.Result` to some result instance and do not call the `ActionExecutionDelegate`.

The framework provides an abstract `ActionFilterAttribute` that you can subclass.

Exception filters

Exception filters implement either the `IExceptionFilter` or `IAsyncExceptionFilter` interface. They can be used to implement common error handling policies for an app.

The following sample exception filter uses a custom developer error view to display details about exceptions that occur when the application is in development:

```
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;
    }
}
```

Exception filters do not have two events (for before and after) - they only implement `OnException` (or `OnExceptionAsync`).

Exception filters handle unhandled exceptions that occur in controller creation, model binding, action filters, or action methods. They won't catch exceptions that occur in Resource filters, Result filters, or MVC Result execution.

To handle an exception, set the `ExceptionContext.ExceptionHandled` property to true or write a response. This stops propagation of the exception. Note that an Exception filter can't turn an exception into a "success". Only an Action filter can do that.

 Note

In ASP.NET 1.1, the response is not sent if you set `ExceptionHandled` to true **and** write a response. In that scenario, ASP.NET Core 1.0 does send the response, and ASP.NET Core 1.1.2 will return to the 1.0 behavior. For more information, see issue #5594 in the GitHub repository.

Exception filters are good for trapping exceptions that occur within MVC actions, but they're not as flexible as error handling middleware. Prefer middleware for the general case, and use filters only where you need to do error handling *differently* based on which MVC action was chosen. For example, your app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

The framework provides an abstract `ExceptionFilterAttribute` that you can subclass.

You can use an action filter to automatically validate model state and return any errors if the state is invalid:

```
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);
            }
        }
    }
}
```

The `OnActionExecuted` method runs after the action method and can see and manipulate the results of the action through the `ActionExecutedContext.Result` property. `ActionExecutedContext.Canceled` will be set to true if the action execution was short-circuited by another filter. `ActionExecutedContext.Exception` will be set to a non-null value if the action or a subsequent action filter threw an exception. Setting `ActionExecutedContext.Exception` to null effectively 'handles' an exception, and `ActionExectedContext.Result` will then be executed as if it were returned from the action method normally.

Result filters

Result filters implement either the `IResultFilter` or `IAsyncResultFilter` interface, and their execution surrounds the execution of action results.

Here's an example of a Result filter that adds an HTTP header.

```
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.
    }
}
```

The kind of result being executed depends on the action in question. An MVC action returning a view would include all razor processing as part of the `ViewResult` being executed. An API method might perform some serialization as part of the execution of the result. Learn more about action results

Result filters are only executed for successful results - when the action or action filters produce an action result. Result filters are not executed when exception filters handle an exception.

The `OnResultExecuting` method can short-circuit execution of the action result and subsequent result filters by setting

`ResultExecutingContext.Cancel` to true. You should generally write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception will also prevent execution of the action result and subsequent filters, but will be treated as a failure instead of a successful result.

When the `OnResultExecuted` method runs, the response has likely been sent to the client and cannot be changed further (unless an exception was thrown). `ResultExecutedContext.Canceled` will be set to true if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` will be set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively 'handles' an exception and prevents the exception from being rethrown by MVC later in the pipeline. When you're handling an exception in a result filter, you might not be able to write any data to the response. If the action result throws partway through its execution, and the headers have already been flushed to the client, there's no reliable mechanism to send a failure code.

For an `IAsyncResultFilter` a call to `await next()` on the `ResultExecutionDelegate` executes any subsequent result filters and the action result. To short-circuit, set `ResultExecutingContext.Cancel` to true and do not call the `ResultExectionDelegate`.

The framework provides an abstract `ResultFilterAttribute` that you can subclass. The AddHeaderAttribute class shown earlier is an example of a result filter attribute.

Using middleware in the filter pipeline

Resource filters work like middleware in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they are part of MVC, which means that they have access to MVC context and constructs.

In ASP.NET Core 1.1, you can use middleware in the filter pipeline. You might want to do that if you have a middleware component that needs access to MVC route data, or one that should run only for certain controllers or actions.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware that you want to inject into the filter pipeline. Here's an example that uses the localization middleware to establish the current culture for a request:

```
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);
    }
}
```

You can then use the `MiddlewareFilterAttribute` to run the middleware for a selected controller or action or globally:

```
[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + "CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Next actions

To experiment with filters, download, test and modify the sample.

Areas

By Dhananjay Kumar and Rick Anderson

Areas are an ASP.NET MVC feature used to organize related functionality into a group as a separate namespace (for routing) and folder structure (for views). Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action`.

Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is effectively an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models. In this scenario, you can use Areas to physically partition the business components in the same project.

An area can be defined as smaller functional units in an ASP.NET Core MVC project with its own set of controllers, views, and models.

Consider using Areas in an MVC project when:

Your application is made of multiple high-level functional components that should be logically separated

You want to partition your MVC project so that each functional area can be worked on independently

Area features:

An ASP.NET Core MVC app can have any number of areas

Each area has its own controllers, models, and views

Allows you to organize large MVC projects into multiple high-level components that can be worked on independently

Supports multiple controllers with the same name - as long as they have different *areas*

Let's take a look at an example to illustrate how Areas are created and used. Let's say you have a store app that has two distinct groupings of controllers and views: Products and Services. A typical folder structure for that using MVC areas looks like below:

Project name

Areas

Products

Controllers

HomeController.cs

ManageController.cs

Views

Home

Index.cshtml

Manage

Index.cshtml

Services

Controllers

HomeController.cs

Views

Home

Index.cshtml

When MVC tries to render a view in an Area, by default, it tries to look in the following locations:

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
   /Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
   /Views/Shared/<Action-Name>.cshtml
```

These are the default locations which can be changed via the `AreaViewLocationFormats` on the `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions`.

For example, in the below code instead of having the folder name as 'Areas', it has been changed to '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");
   });
```

One thing to note is that the structure of the *Views* folder is the only one which is considered important here and the content of the rest of the folders like *Controllers* and *Models* does **not** matter. For example, you need not have a *Controllers* and *Models* folder at all. This works because the content of *Controllers* and *Models* is just code which gets compiled into a .dll where as the content of the *Views* is not until a request to that view has been made.

Once you've defined the folder hierarchy, you need to tell MVC that each controller is associated with an area. You do that by decorating the controller name with the `[Area]` attribute.

```
...
   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();
           }
       }
   }
```

Set up a route definition that works with your newly created areas. The Routing to Controller Actions article goes into detail about how to create route definitions, including using conventional routes versus attribute routes. In this example, we'll use a conventional route. To do so, open the *Startup.cs* file and modify it by adding the `areaRoute` named route definition below.

```
...
   app.UseMvc(routes =>
   {
     routes.MapRoute(name: "areaRoute",
       template: "{area:exists}/{controller=Home}/{action=Index}");

     routes.MapRoute(
         name: "default",
         template: "{controller=Home}/{action=Index}/{id?}");
   });
```

Browsing to `http://<yourApp>/products`, the `Index` action method of the `HomeController` in the `Products` area will be invoked.

Link Generation

Generating links from an action within an area based controller to another action within the same controller.

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax: `@Html.ActionLink("Go to Product's Home Page", "Index")`

TagHelper syntax: `<a asp-action="Index">Go to Product's Home Page</a>`

Note that we need not supply the 'area' and 'controller' values here as they are already available in the context of the current request. These kind of values are called `ambient` values.

Generating links from an action within an area based controller to another action on a different controller

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax: `@Html.ActionLink("Go to Manage Products’ Home Page", "Index", "Manage")`

TagHelper syntax: `<a asp-controller="Manage" asp-action="Index">Go to Manage Products’ Home Page</a>`

Note that here the ambient value of an 'area' is used but the 'controller' value is specified explicitly above.

Generating links from an action within an area based controller to another action on a different controller and a different area.

Let's say the current request's path is like `/Products/Home/Create`

HtmlHelper syntax: `@Html.ActionLink("Go to Services’ Home Page", "Index", "Home", new { area = "Services" })`

TagHelper syntax: `<a asp-area="Services" asp-controller="Home" asp-action="Index">Go to Services’ Home Page</a>`

Note that here no ambient values are used.

Generating links from an action within an area based controller to another action on a different controller and **not** in an area.

HtmlHelper syntax: `@Html.ActionLink("Go to Manage Products’ Home Page", "Index", "Home", new { area = "" })`

TagHelper syntax: `<a asp-area="" asp-controller="Manage" asp-action="Index">Go to Manage Products’ Home Page</a>`

Since we want to generate links to a non-area based controller action, we empty the ambient value for 'area' here.

Publishing Areas

All `*.cshtml` and `wwwroot/**` files are published to output when `<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the *.csproj* file.

Advanced topics for ASP.NET Core MVC

Working with the Application Model
Application parts
Custom Model Binding
Custom formatters

Working with the Application Model

By Steve Smith

ASP.NET Core MVC defines an *application model* representing the components of an MVC app. You can read and manipulate this model to modify how MVC elements behave. By default, MVC follows certain conventions to determine which classes are considered to be controllers, which methods on those classes are actions, and how parameters and routing behave. You can customize this behavior to suit your app's needs by creating your own conventions and applying them globally or as attributes.

Models and Providers

The ASP.NET Core MVC application model include both abstract interfaces and concrete implementation classes that describe an MVC application. This model is the result of MVC discovering the app's controllers, actions, action parameters, routes, and filters according to default conventions. By working with the application model, you can modify your app to follow different conventions from the default MVC behavior. The parameters, names, routes, and filters are all used as configuration data for actions and controllers.

The ASP.NET Core MVC Application Model has the following structure:

ApplicationModel
Controllers (ControllerModel)
Actions (ActionModel)
Parameters (ParameterModel)

Each level of the model has access to a common `Properties` collection, and lower levels can access and overwrite property values set by higher levels in the hierarchy. The properties are persisted to the `ActionDescriptor.Properties` when the actions are created. Then when a request is being handled, any properties a convention added or modified can be accessed through `ActionContext.ActionDescriptor.Properties`. Using properties is a great way to configure your filters, model binders, etc. on a per-action basis.

▯ Note

The `ActionDescriptor.Properties` collection is not thread safe (for writes) once app startup has finished. Conventions are the best way to safely add data to this collection.

IApplicationModelProvider

ASP.NET Core MVC loads the application model using a provider pattern, defined by the IApplicationModelProvider interface. This section covers some of the internal implementation details of how this provider functions. This is an advanced topic - most apps that leverage the application model should do so by working with conventions.

Implementations of the `IApplicationModelProvider` interface "wrap" one another, with each implementation calling `OnProvidersExecuting` in ascending order based on its `Order` property. The `OnProvidersExecuted` method is then called in reverse order. The framework defines several providers:

First (`Order=-1000`):

`DefaultApplicationModelProvider`

Then (`Order=-990`):

`AuthorizationApplicationModelProvider`
`CorsApplicationModelProvider`

▯ Note

The order in which two providers with the same value for `Order` are called is undefined, and therefore should not be relied upon.

Note

`IApplicationModelProvider` is an advanced concept for framework authors to extend. In general, apps should use conventions and frameworks should use providers. The key distinction is that providers always run before conventions.

The `DefaultApplicationModelProvider` establishes many of the default behaviors used by ASP.NET Core MVC. Its responsibilities include:

- Adding global filters to the context
- Adding controllers to the context
- Adding public controller methods as actions
- Adding action method parameters to the context
- Applying route and other attributes

Some built-in behaviors are implemented by the `DefaultApplicationModelProvider`. This provider is responsible for constructing the `ControllerModel`, which in turn references `ActionModel`, `PropertyModel`, and `ParameterModel` instances. The `DefaultApplicationModelProvider` class is an internal framework implementation detail that can and will change in the future.

The `AuthorizationApplicationModelProvider` is responsible for applying the behavior associated with the `AuthorizeFilter` and `AllowAnonymousFilter` attributes. Learn more about these attributes.

The `CorsApplicationModelProvider` implements behavior associated with the `IEnableCorsAttribute` and `IDisableCorsAttribute`, and the `DisableCorsAuthorizationFilter`. Learn more about CORS.

Conventions

The application model defines convention abstractions that provide a simpler way to customize the behavior of the models than overriding the entire model or provider. These abstractions are the recommended way to modify your app's behavior. Conventions provide a way for you to write code that will dynamically apply customizations. While filters provide a means of modifying the framework's behavior, customizations let you control how the whole app is wired together.

The following conventions are available:

- `IApplicationModelConvention`
- `IControllerModelConvention`
- `IActionModelConvention`
- `IParameterModelConvention`

Conventions are applied by adding them to MVC options or by implementing `Attribute`s and applying them to controllers, actions, or action parameters (similar to `Filters`). Unlike filters, conventions are only executed when the app is starting, not as part of each request.

Sample: Modifying the ApplicationModel

The following convention is used to add a property to the application model.

```
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;
        }
    }
}
```

Application model conventions are applied as options when MVC is added in `ConfigureServices` in `Startup`.

```
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());
    });
}
```

Properties are accessible from the `ActionDescriptor` properties collection within controller actions:

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Sample: Modifying the ControllerModel Description

As in the previous example, the controller model can also be modified to include custom properties. These will override existing properties with the same name specified in the application model. The following convention attribute adds a description at the controller level:

```
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;
        }
    }
}
```

This convention is applied as an attribute on a controller.

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
```

The "description" property is accessed in the same manner as in previous examples.

Sample: Modifying the ActionModel Description

A separate attribute convention can be applied to individual actions, overriding behavior already applied at the application or controller level.

```
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;
        }
    }
}
```

Applying this to an action within the previous example's controller demonstrates how it overrides the controller-level convention:

```
[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"];
    }
}
```

Sample: Modifying the ParameterModel

The following convention can be applied to action parameters to modify their `BindingInfo`. The following convention requires that the parameter be a route parameter; other potential binding sources (such as query string values) are ignored.

```
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;
        }
    }
}
```

The attribute may be applied to any action parameter:

```
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}";
    }
}
```

Sample: Modifying the ActionModel Name

The following convention modifies the `ActionModel` to update the *name* of the action to which it is applied. The new name is provided as a parameter to the attribute. This new name is used by routing, so it will affect the route used to reach this action method.

```
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;
        }
    }
}
```

This attribute is applied to an action method in the `HomeController`:

```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

Even though the method name is `SomeName`, the attribute overrides the MVC convention of using the method name and replaces the action name with `MyCoolAction`. Thus, the route used to reach this action is `/Home/MyCoolAction`.

Note

This example is essentially the same as using the built-in ActionName attribute.

Sample: Custom Routing Convention

You can use an `IApplicationModelConvention` to customize how routing works. For example, the following convention will incorporate Controllers' namespaces into their routes, replacing `.` in the namespace with `/` in the route:

```
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
        }
    }
}
```

The convention is added as an option in Startup.

```
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());
    });
}
```

 Tip

You can add conventions to your middleware by accessing `MvcOptions` using `services.Configure<MvcOptions>(c => c.Conventions.Add(YOURCONVENTION));`

This sample applies this convention to routes that are not using attribute routing where the controller has "Namespace" in its name. The following controller demonstrates this convention:

```
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.";
        }
    }
}
```

Application Model Usage in WebApiCompatShim

ASP.NET MVC Core uses a different set of conventions from ASP.NET Web API 2. Using custom conventions, you can modify an ASP.NET MVC Core app's behavior to be consistent with that of a Web API app. Microsoft ships the WebApiCompatShim specifically for this purpose.

Note

Learn more about migrating from ASP.NET Web API).

To use the Web API Compatibility Shim, you need to add the package to your project and then add the conventions to MVC by calling `AddWebApiConventions` in `Startup`:

```
services.AddMvc().AddWebApiConventions();
```

The conventions provided by the shim are only applied to parts of the app that have had certain attributes applied to them. The following four attributes are used to control which controllers should have their conventions modified by the shim's conventions:

UseWebApiActionConventionsAttribute
UseWebApiOverloadingAttribute
UseWebApiParameterConventionsAttribute
UseWebApiRoutesAttribute

Action Conventions

The `UseWebApiActionConventionsAttribute` is used to map the HTTP method to actions based on their name (for instance, `Get` would map to `HttpGet`). It only applies to actions that do not use attribute routing.

Overloading

The `UseWebApiOverloadingAttribute` is used to apply the `WebApiOverloadingApplicationModelConvention` convention. This convention adds an `OverloadActionConstraint` to the action selection process, which limits candidate actions to those for which the request satisfies all non-optional parameters.

Parameter Conventions

The `UseWebApiParameterConventionsAttribute` is used to apply the `WebApiParameterConventionsApplicationModelConvention` action convention. This convention specifies that simple types used as action parameters are bound from the URI by default, while complex types are bound from the request body.

Routes

The `UseWebApiRoutesAttribute` controls whether the `WebApiApplicationModelConvention` controller convention is applied. When enabled, this convention is used to add support for areas to the route.

In addition to a set of conventions, the compatibility package includes a `System.Web.Http.ApiController` base class that replaces the one provided by Web API. This allows your controllers written for Web API and inheriting from its `ApiController` to work as they were designed, while running on ASP.NET Core MVC. This base controller class is decorated with all of the `UseWebApi*` attributes listed above. The `ApiController` exposes properties, methods, and result types that are compatible with those found in Web API.

Using ApiExplorer to Document Your App

The application model exposes an `ApiExplorer` property at each level that can be used to traverse the app's structure. This can be used to generate help pages for your Web APIs using tools like Swagger. The `ApiExplorer` property exposes an `IsVisible` property that can be set to specify which parts of your app's model should be exposed. You can configure this setting using a convention:

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

Using this approach (and additional conventions if required), you can enable or disable API visibility at any level within your app.

Application Parts

By Steve Smith

View or download sample code

An *Application Part* is an abstraction over the resources of an application, from which MVC features like controllers, view components, or tag helpers may be discovered. One example of an application part is an AssemblyPart, which encapsulates an assembly reference and exposes types and compilation references. *Feature providers* work with application parts to populate the features of an ASP.NET Core MVC app. The main use case for application parts is to allow you to configure your app to discover (or avoid loading) MVC features from an assembly.

Introducing Application Parts

MVC apps load their features from application parts. In particular, the AssemblyPart class represents an application part that is backed by an assembly. You can use these classes to discover and load MVC features, such as controllers, view components, tag helpers, and razor compilation sources. The ApplicationPartManager is responsible for tracking the application parts and feature providers available to the MVC app. You can interact with the `ApplicationPartManager` in `Startup` when you configure MVC:

```
// 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 => p.ApplicationParts.Add(part));
```

By default MVC will search the dependency tree and find controllers (even in other assemblies). To load an arbitrary assembly (for instance, from a plugin that isn't referenced at compile time), you can use an application part.

You can use application parts to *avoid* looking for controllers in a particular assembly or location. You can control which parts (or assemblies) are available to the app by modifying the `ApplicationParts` collection of the `ApplicationPartManager`. The order of the entries in the `ApplicationParts` collection is not important. It is important to fully configure the `ApplicationPartManager` before using it to configure services in the container. For example, you should fully configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Failing to do so, will mean that controllers in application parts added after that method call will not be affected (will not get registered as services) which might result in incorrect bevavior of your application.

If you have an assembly that contains controllers you do not want to be used, remove it from the `ApplicationPartManager`:

```
services.AddMvc()
    .ConfigureApplicationPartManager(p =>
    {
        var dependentLibrary = p.ApplicationParts
            .FirstOrDefault(part => part.Name == "DependentLibrary");
        if (dependentLibrary != null)
        {
           p.ApplicationParts.Remove(dependentLibrary);
        }
    })
```

In addition to your project's assembly and its dependent assemblies, the `ApplicationPartManager` will include parts for `Microsoft.AspNetCore.Mvc.TagHelpers` and `Microsoft.AspNetCore.Mvc.Razor` by default.

Application Feature Providers

Application Feature Providers examine application parts and provide features for those parts. There are built-in feature providers for the following MVC features:

Controllers
Metadata Reference
Tag Helpers
View Components

Feature providers inherit from `IApplicationFeatureProvider<T>`, where `T` is the type of the feature. You can implement your own feature providers for any of MVC's feature types listed above. The order of feature providers in the `ApplicationPartManager.FeatureProviders` collection can be important, since later providers can react to actions taken by previous providers.

Sample: Generic Controller Feature

By default, ASP.NET Core MVC ignores generic controllers (for example, `SomeController<T>`). This sample uses a controller feature provider that runs after the default provider and adds generic controller instances for a specified list of types (defined in `EntityTypes.Types`):

```
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);
            }
        }
    }
}
```

The entity types:

```
public static class EntityTypes
{
    public static IReadOnlyList<TypeInfo> Types => new List<TypeInfo>()
    {
        typeof(Sprocket).GetTypeInfo(),
        typeof(Widget).GetTypeInfo(),
    };

    public class Sprocket { }
    public class Widget { }
}
```

The feature provider is added in `Startup`:

```
services.AddMvc()
        .ConfigureApplicationPartManager(p =>
            p.FeatureProviders.Add(new GenericControllerFeatureProvider()));
```

By default, the generic controller names used for routing would be of the form *GenericController`1[Widget]* instead of *Widget*. The following attribute is used to modify the name to correspond to the generic type used by the controller:

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System;

namespace AppPartSample
{
    // 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;
        }
    }
}
```

The `GenericController` class:

```
using Microsoft.AspNetCore.Mvc;

namespace AppPartSample
{
    [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.");
        }
    }
}
```

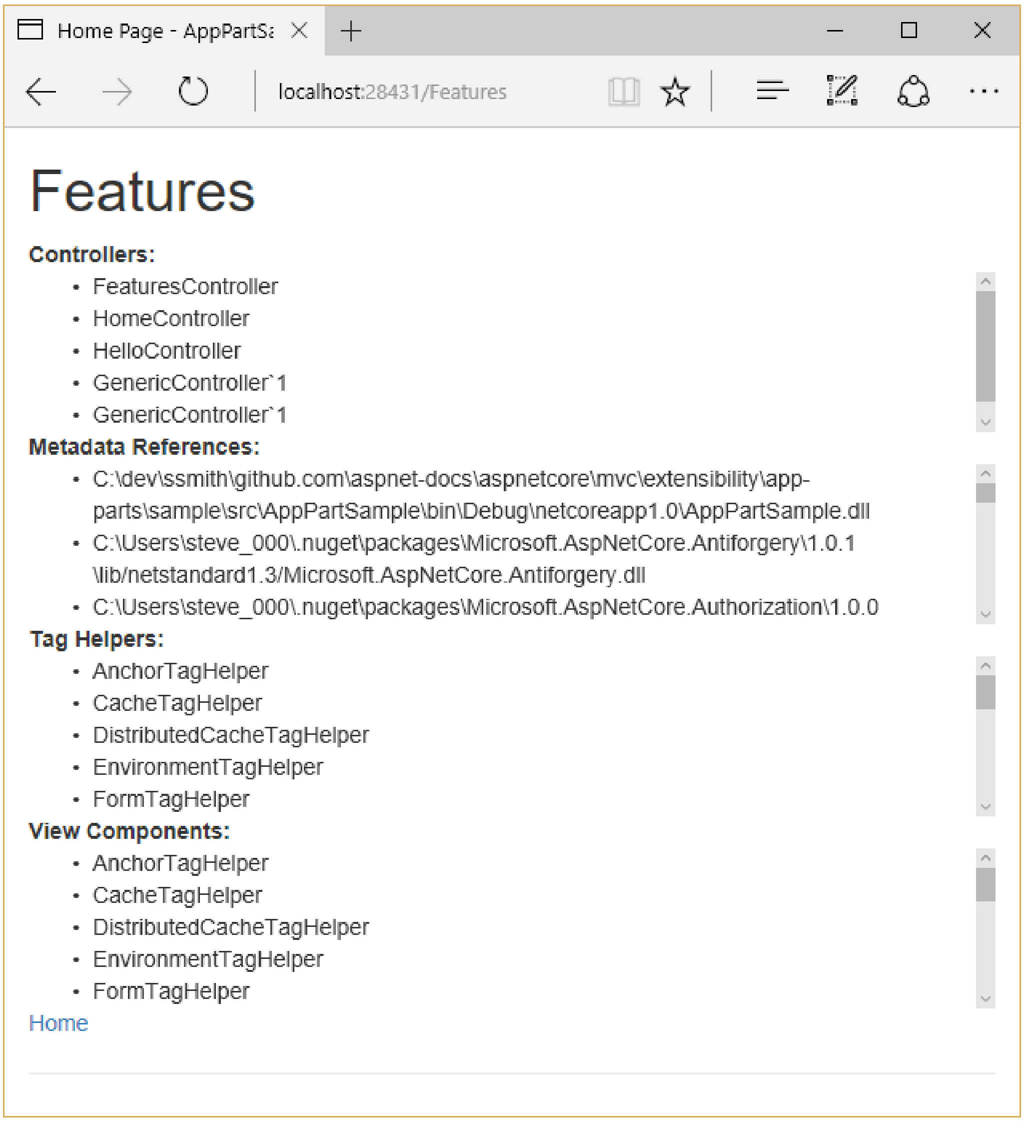
The result, when a matching route is requested:

Sample: Display Available Features

You can iterate through the populated features available to your app by requesting an `ApplicationPartManager` through dependency injection and using it to populate instances of the appropriate features:

```
using AppPartSample.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 AppPartSample.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);
        }
    }
}
```

Example output:



Home Page - AppPartSa
localhost:28431/Features
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

Custom Model Binding

By Steve Smith

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

View or download sample from GitHub

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers` needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

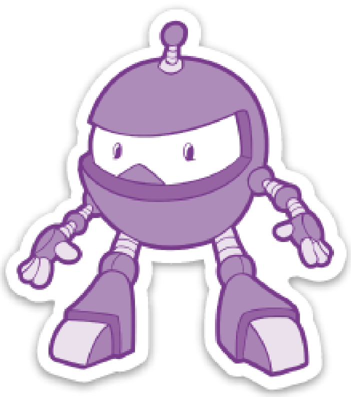
Model binding review

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string in the input. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter`. We recommended you create a type converter if you have a simple `string` -> `SomeType` mapping that doesn't require external resources.

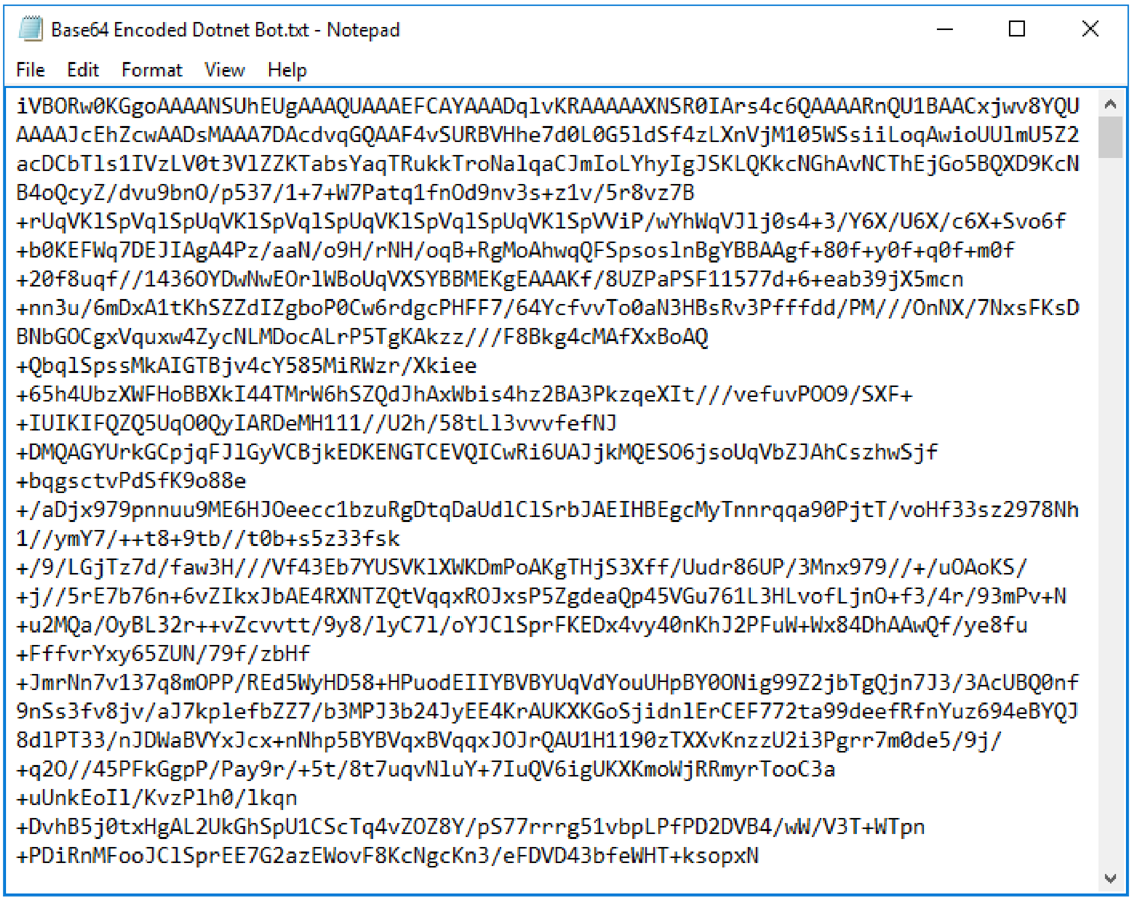
Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the ByteArrayModelBinder which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, the following image can be encoded as a string.



A small portion of the encoded string is shown in the following image:



Follow the instructions in the sample's README to convert the base64-encoded string into a file.

ASP.NET Core MVC can take a base64-encoded strings and use a `ByteArrayModelBinder` to convert it into a byte array. The ByteArrayModelBinderProvider which implements IModelBinderProvider maps `byte[]` arguments to `ByteArrayModelBinder`:

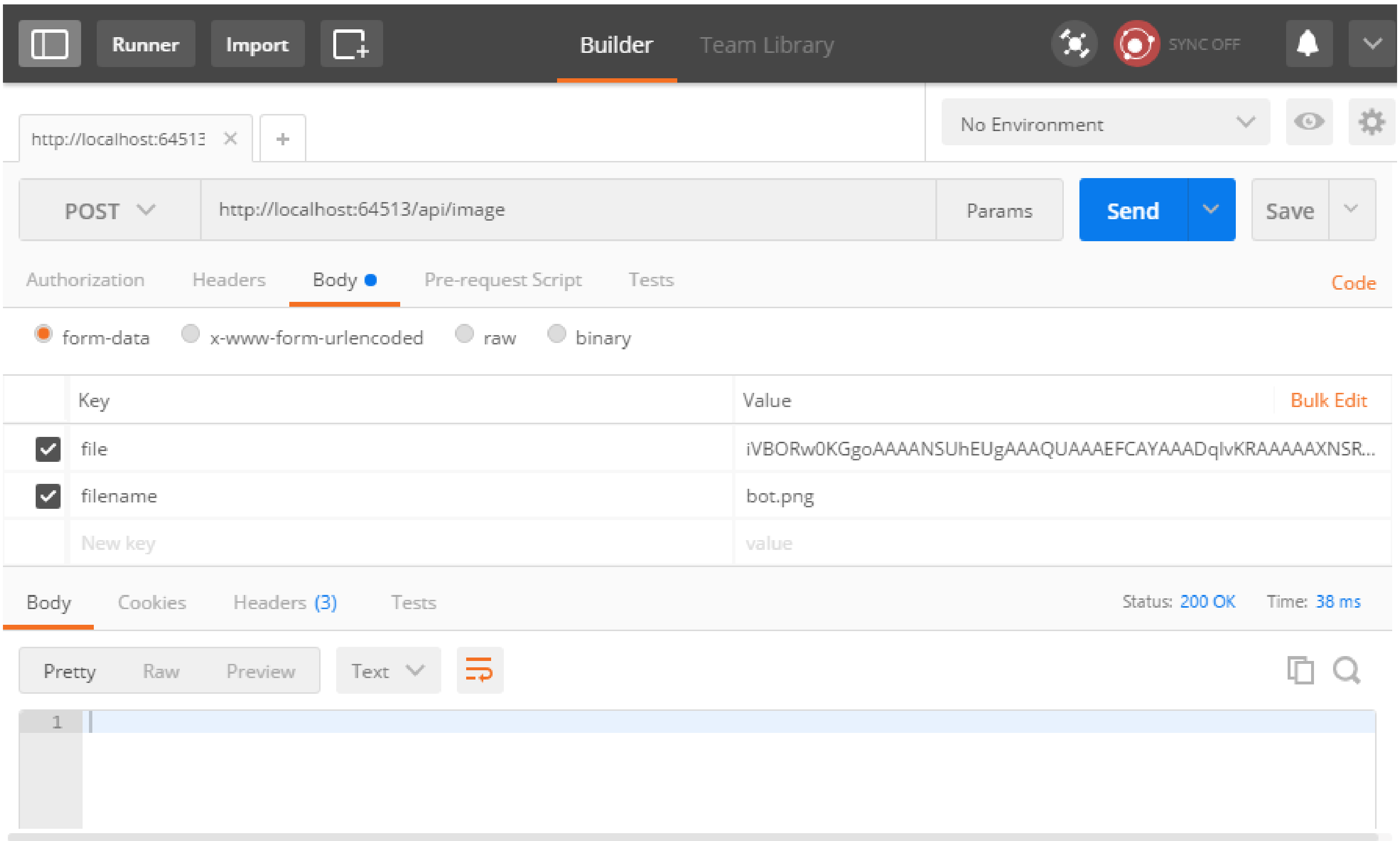
```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        return new ByteArrayModelBinder();
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the ModelBinderAttribute.

The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

```
// 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);
}
```

You can POST a base64-encoded string to this api method using a tool like Postman:



As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

```
[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; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

Converts incoming request data into strongly typed key arguments.
Uses Entity Framework Core to fetch the associated entity.
Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

```
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; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinderProvider` that should be used to bind `Author` action parameters.

The `AuthorEntityBinder` is used to bind an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

```
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 TaskCache.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
            valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return TaskCache.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 TaskCache.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 TaskCache.CompletedTask;
    }
}
```

The following code shows how to use the `AuthorEntityBinder` in an action method:

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that do not use default conventions:

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")]Author author)
{
    if (author == null)
    {
        return NotFound();
    }
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    return Ok(author);
}
```

In this example, since the name of the argument is not the default `authorId`, it's specified on the parameter using `ModelBinder` attribute. Note that both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be considerable simplification when you have several methods that bind to the author model, and can help you to follow the DRY principle.

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author` typed parameters.

```
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;
        }
    }
}
```

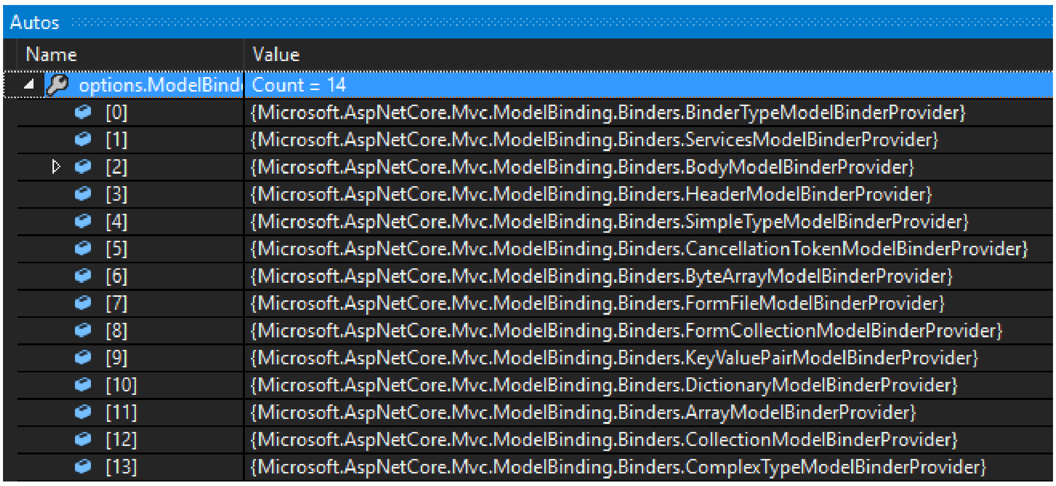
> Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

```
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());
    });
}
```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder is used.

The following image shows the default model binders from the debugger.



Adding your provider to the end of the collection may result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it is used for `Author` action arguments.

```
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());
    });
}
```

Recommendations and best practices

Custom model binders:

Should not attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an action filter or logic within the action method itself should handle the failure.
Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
Typically should not be used to convert a string into a custom type, a `TypeConverter` is usually a better option.

Custom formatters in ASP.NET Core MVC web APIs

By Tom Dykstra

ASP.NET Core MVC has built-in support for data exchange in web APIs by using JSON, XML, or plain text formats. This article shows how to add support for additional formats by creating custom formatters.

View or download sample from GitHub.

When to use custom formatters

Use a custom formatter when you want the content negotiation process to support a content type that isn't supported by the built-in formatters (JSON, XML, and plain text).

For example, if some of the clients for your web API can handle the Protobuf format, you might want to use Protobuf with those clients because it's more efficient. Or you might want your web API to send contact names and addresses in vCard format, a commonly used format for exchanging contact data. The sample app provided with this article implements a simple vCard formatter.

Overview of how to use a custom formatter

Here are the steps to create and use a custom formatter:

Create an output formatter class if you want to serialize data to send to the client.
Create an input formatter class if you want to deserialize data received from the client.
Add instances of your formatters to the `InputFormatters` and `OutputFormatters` collections in MvcOptions.

The following sections provide guidance and code examples for each of these steps.

How to create a custom formatter class

To create a formatter:

Derive the class from the appropriate base class.
Specify valid media types and encodings in the constructor.
Override `CanReadType`/`CanWriteType` methods
Override `ReadRequestBodyAsync`/`WriteResponseBodyAsync` methods

Derive from the appropriate base class

For text media types (for example, vCard), derive from the TextInputFormatter or TextOutputFormatter base class.

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the InputFormatter or OutputFormatter base class.

Specify valid media types and encodings

In the constructor, specify valid media types and encodings by adding to the `SupportedMediaTypes` and `SupportedEncodings` collections.

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

 Note

You can't do constructor dependency injection in a formatter class. For example, you can't get a logger by adding a logger parameter to the constructor. To access services, you have to use the context object that gets passed in to your methods. A code example below shows how to do this.

Override CanReadType/CanWriteType

Specify the type you can deserialize into or serialize from by overriding the `CanReadType` or `CanWriteType` methods. For example, you might only be able to create vCard text from a `Contact` type and vice versa.

```
protected override bool CanWriteType(Type type)
{
    if (typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type))
    {
        return base.CanWriteType(type);
    }
    return false;
}
```

The CanWriteResult method

In some scenarios you have to override `CanWriteResult` instead of `CanWriteType`. Use `CanWriteResult` if the following conditions are true:

Your action method returns a model class.
There are derived classes which might be returned at runtime.
You need to know at runtime which derived class was returned by the action.

For example, suppose your action method signature returns a `Person` type, but it may return a `Student` or `Instructor` type that derives from `Person`. If you want your formatter to handle only `Student` objects, check the type of Object in the context object provided to the `CanWriteResult` method. Note that it's not necessary to use `CanWriteResult` when the action method returns `IActionResult`; in that case, the `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync/WriteResponseBodyAsync

You do the actual work of deserializing or serializing in `ReadRequestBodyAsync` or `WriteResponseBodyAsync`. The highlighted lines in the following example show how to get services from the dependency injection container (you can't get them from constructor parameters).

```
public override Task WriteResponseBodyAsync(OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    IServiceProvider serviceProvider = context.HttpContext.RequestServices;
    var logger = serviceProvider.GetService(typeof(ILogger<VcardOutputFormatter>)) as ILogger;

    var response = context.HttpContext.Response;

    var buffer = new StringBuilder();
    if (context.Object is IEnumerable<Contact>)
    {
        foreach (Contact contact in context.Object as IEnumerable<Contact>)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        var contact = context.Object as Contact;
        FormatVcard(buffer, contact, logger);
    }
    return response.WriteAsync(buffer.ToString());
}

private static void FormatVcard(StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendFormat($"N:{contact.LastName};{contact.FirstName}\r\n");
    buffer.AppendFormat($"FN:{contact.FirstName} {contact.LastName}\r\n");
    buffer.AppendFormat($"UID:{contact.ID}\r\n");
    buffer.AppendLine("END:VCARD");
    logger.LogInformation($"Writing {contact.FirstName} {contact.LastName}");
}
```

How to configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the `InputFormatters` or `OutputFormatters` collection.

```
services.AddMvc(options =>
{
    options.InputFormatters.Add(new VcardInputFormatter());
    options.OutputFormatters.Add(new VcardOutputFormatter());
});
```

Next steps

See the sample application, which implements simple vCard input and output formatters. The application reads and writes vCards that look like the following example:

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
UID:20293482-9240-4d68-b475-325df4a83728
END:VCARD
```

To see vCard output, run the application and send a Get request with Accept header "text/vcard" to `http://localhost:63313/api/contacts/` (when running from Visual Studio) or `http://localhost:5000/api/contacts/` (when running from the command line).

To add a vCard to the in-memory collection of contacts, send a Post request to the same URL, with Content-Type header "text/vcard" and with vCard text in the body, formatted like the example above.

Introduction to Razor Pages in ASP.NET Core

By Ryan Nowak and Rick Anderson

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

Razor Pages requires ASP.NET Core 2.0.0 or later. Tooling support for Razor Pages ships in Visual Studio 2017 Update 3 or later.

Getting started

Razor Pages is on by default in MVC. If you are using a typical *Startup.cs* like the following code, Razor Pages is enabled:

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

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

All the new Razor Pages types and features are included in the `Microsoft.AspNetCore.Mvc.RazorPages` assembly. If you are referencing the `Microsoft.AspNetCore.Mvc` package, then a reference to the Razor Pages assembly is already included.

Consider a basic page:

```
@page

@{
    var message = "Hello, World!";
}

<html>
<body>
    <p>@message</p>
</body>
</html>
```

The preceeding code looks a lot like a regular Razor view file. What makes it different is the new `@page` directive. Using `@page` makes this file into an MVC action - which means that it can handle requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Pages path and the matching URL:

FILE NAME AND PATH	MATCHING URL
/Pages/Index.cshtml	`/` or `/Index`
/Pages/Contact.cshtml	`/Contact`
/Pages/Store/Contact.cshtml	`/Store/Contact`

The runtime looks for Razor Pages files in the *Pages* folder by default.

Writing a basic form

The new Razor Pages features are designed to make common patterns used with web browsers easy. Consider a page that implements a basic 'contact us' form for the `Contact` model:

For the examples on this page, the `DbContext` is initialized in the *Startup.cs** file.

The *MyApp/Contact.cs* file:

```
using System.ComponentModel.DataAnnotations;

namespace MyApp
{
    public class Contact
    {
        [Required]
        public string Name { get; set; }

        [Required]
        public string Email { get; set; }
    }
}
```

The *MyApp/Pages/Contact.cshtml* file:

```
@page
@using MyApp
@using Microsoft.AspNetCore.Mvc.RazorPages
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers"
@inject ApplicationDbContext Db

@functions {
    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (ModelState.IsValid)
        {
            Db.Contacts.Add(Contact);
            await Db.SaveChangesAsync();
            return RedirectToPage();
        }

        return Page();
    }
}

<html>
<body>
    <p>Enter your contact info here and we will email you about our fine products!</p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
      <div>Name: <input asp-for="Contact.Name" /></div>
      <div>Email: <input asp-for="Contact.Email" /></div>
      <input type="submit" />
    </form>
</body>
</html>
```

The page has a `OnPostAsync` *handler method* which runs on `POST` requests (when a user posts the form). You can add handler methods for any HTTP verb. You most frequently use an `OnGet` handler to initialize any state a needed to show the HTML and `OnPost` to handle form submissions. The `Async` naming suffix is optional but is often used by convention. The code that's in

`OnPostAsync` in the preceding example looks similar to what you would normally write in a controller. This is typical for pages. Most of the MVC primitives like model binding, validation, and action results are shared.

The basic flow of `OnPostAsync` is:

Check for validation errors.
If there are no errors, save the data and redirect -
Else, show the page again with the validation errors.

When the data is entered successfully, the `OnPostAsync` handler method calls the `RedirctToPage` helper method to return an instance of `RedirectToPageResult`. This is a new action result similar to `RedirectToAction` or `RedirectToRoute` but customized for pages. In the preceding sample it redirects back to the same URL as the current page (`/Contact`). Later I'll show how to redirect to a different page.

When the submitted form has validation errors, the`OnPostAsync` handler method calls the `Page` helper method. `Page` returns an instance of `PageResult`. This is similar to how actions in controllers return `View`. `PageResult` is the default for a handler method. A handler method that returns `void` will render the page.

The `Contact` property is using the new `[BindProperty]` attribute to opt-in to model binding. Pages, by default, bind properties only with non-GET verbs. Binding to properties can reduce the amount of code you have to write by using the same property to render form fields (`<input asp-for="Contacts.Name" />`) and accept the input.

Rather than using `@model` here, we're taking advantage of a special new feature for pages. By default, the generated `Page`-derived class *is* the model. This means that features like model binding, tag helpers, and HTML helpers all *just work* with the properties defined in `@functions`. Using a *view model* with Razor views is a best practice. With pages, you get a view model automatically.

Notice that this Page also uses `@inject` for dependency injection, which is the same as traditional Razor views - this generates the `Db` property that is used in `OnPostAsync`. Injected (`@inject`) properties are set before handler methods run.

You don't have to write any code for antiforgery validation. Antiforgery token generation and validation is automatic for pages. No additional code or attributes are needed to get this security feature.

Introducing PageModel

You could write this form by partitioning the view code and the handler method into separate files. The view code:

MyApp/Pages/Contact.cshtml

```
@page
@using MyApp
@using MyApp.Pages
@using Microsoft.AspNetCore.Mvc.RazorPages
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers"
@model ContactModel

<html>
<body>
    <p>Enter your contact info here and we will email you about our fine products!</p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
      <div>Name: <input asp-for="Contact.Name" /></div>
      <div>Email: <input asp-for="Contact.Email" /></div>
      <input type="submit" />
    </form>
</body>
</html>
```

The `PageModel` class, a 'code-behind' file for the view code:

MyApp/Pages/Contact.cshtml.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace MyApp.Pages
{
    public class ContactModel : PageModel
    {
        public ContactModel(ApplicationDbContext db)
        {
            Db = db;
        }

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

        private ApplicationDbContext Db { get; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (ModelState.IsValid)
            {
                Db.Contacts.Add(Contact);
                await Db.SaveChangesAsync();
                return RedirectToPage();
            }

            return Page();
        }
    }
}
```

By convention the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page. Not much change is needed to convert from a page using `@functions` to define handlers and a page using a `PageModel` class. The main change is to add constructor injection for all your injected (`@inject`) properties.

Using a `PageModel` supports unit testing, but requires you to write an explicit constructor and class. Pages without `PageModel` files support runtime compilation, which can be an advantage in development.

Using the view engine

Pages work with all the features of the Razor view engine. Layouts, partials, templates, tag helpers, *_ViewStart.cshtml*, *_ViewImports.cshtml* all work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those features.

Add a layout page for the HTML skeleton, and set the `Layout` property from `_ViewStart.cshtml`:

MyApp/Pages/_Layout.cshtml

```
<html>
    ...
</html>
```

MyApp/Pages/_ViewStart.cshtml

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

Note that we placed the layout in the *MyApp/Pages* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. This means that a layout in the *MyApp/Pages* folder can be used from any page.

View search from a page will include the *MyApp/Views/Shared* folder. The layouts, templates, and partials you're using with MVC controllers and conventional Razor views 'just work'.

Add a *_ViewImports.cshtml* file:

MyApp/Pages/_ViewImports.cshtml

```
@namespace MyApp.Pages
@using Microsoft.AspNetCore.Mvc.RazorPages
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers"
```

The `@namespace` directive is a new feature that controls the namespace of the generated code. The `@namespace` directive allows us to get rid of `@using` directives from the page. The `@namespace` directive works by computing the difference in folders between your view code and the `_ViewImports.cshtml` where it appears. Because the *Customer.cshtml* file is also in the *MyApp/Pages* folder, it will have the namespace `MyApp.Pages`. If the path was *MyApp/Pages/Store/Customer.cshtml*, the namespace of the generated code would be *MyApp.Pages.Store*. This is intended so that the C# classes you add and pages generated code *just work* without having to add extra usings.

`@namespace` also works for conventional Razor views.

Here's what the page looks like after simplification:

MyApp/Pages/Contact.cshtml

```
@page
@inject ApplicationDbContext Db

@functions {

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

    public async Task<IActionResult> OnPostAsync()
    {
        if (ModelState.IsValid)
        {
            Db.Contacts.Add(Contact);
            await Db.SaveChangesAsync();
            return RedirectToPage();
        }

        return Page();
    }
}

<div class="row">
    <div class="col-md-3">
        <p>Enter your contact info here and we will email you about our fine products!</p>
        <div asp-validation-summary="All"></div>
        <form method="POST">
            <div>Name: <input asp-for="Contact.Name" /></div>
            <div>Email: <input asp-for="Contact.Email" /></div>
            <input type="submit" />
        </form>
    </div>
</div>
```

URL generation for Pages

Let's suppose we want to do something more useful than showing the same page again when the visitor submits their contact information. We can use `RedirectToPage("/Index")` to redirect to the `Index` page.

This example adds a confirmation message and redirects back to the home page:

MyApp/Pages/Contact.cshtml

```
@page
@inject ApplicationDbContext Db

@functions {

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

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

    public async Task<IActionResult> OnPostAsync()
    {
        if (ModelState.IsValid)
        {
            Db.Contacts.Add(Contact);
            await Db.SaveChangesAsync();

            Message = "Thanks, we'll be in touch shortly.";
            return RedirectToPage("/Index");
        }

        return Page();
    }
}


<div class="row">
    <div class="col-md-3">
        <p>Enter your contact info here and we will email you about our fine products!</p>
        <div asp-validation-summary="All"></div>
        <form method="POST">
            <div>Name: <input asp-for="Contact.Name" /></div>
            <div>Email: <input asp-for="Contact.Email" /></div>
            <input type="submit" />
        </form>
    </div>
</div>
```

MyApp/Pages/Index.cshtml

```
@page

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


<div class="row">
    <div class="col-md-3">
        @if (Message != null)
        {
            <h3>@Message</h3>
        }

        <p>Hi, welcome to our website!</p>
    </div>
</div>
```

We've added another page (*MyApp/Pages/Index.cshtml*), and are redirecting to it using `RedirectToPage("/Index")`. The string `/Index` is the name of the page we just added, and can be used with `Url.Page(...)`, `<a asp-page="..." />` or `RedirectToPage`.

The page name is just the path to the page from the root *MyApp/Pages* folder (including a leading `/`). It seems simple, but this is much more feature rich than just hardcoding a URL. This is URL generation using routing, and can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. From *MyApp/Pages/Contact.cshtml*, you could also redirect to *MyApp/Pages/Index.cshtml* using `RedirectToPage("Index")` or `RedirectToPage("./Index")`. These are both *relative names*. The provided string is *combined* with the page name of the current page to compute the name of the destination page. You can also use the directory traversal `..` operator.

Relative name linking is useful when building sites with a complex structure. If you use relative names to link between pages in a folder, you can rename that folder. All the links still work (because they didn't include the folder name).

Since we have another page here, we're also taking advantage of the `[TempData]` attribute to pass data across pages. `[TempData]` is a more convenient way to use the existing MVC temp data features. The `[TempData]` attribute is new in 2.0.0 and is supported on controllers and pages. In 2.0.0, the default storage for temp data is now cookies. A session provider is no longer required by default.

Using multiple handlers

Let's update this form to support multiple operations. A visitor to the site can either join the mailing list or ask for a free quote.

If you want one page to handle multiple logical actions, you can use *named handler methods*. Any text in the name after `On<Verb>` and before `Async` (if present) in the method name is considered a handler name. The handler methods in the following example have the handler names `JoinMailingList` and `RequestQuote`:

MyApp/Pages/Contact.cshtml

```
@page
@inject ApplicationDbContext Db

@functions {

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

    public async Task<IActionResult> OnPostJoinMailingListAsync()
    {
        ...
    }

    public async Task<IActionResult> OnPostRequestQuoteAsync()
    {
        ...
    }
}
<div class="row">
    <div class="col-md-3">
        <p>Enter your contact info here we will email you about our fine products! Or get a free quote!</p>
        <div asp-validation-summary="All"></div>
        <form method="POST">
            <div>Name: <input asp-for="Contact.Name" /></div>
            <div>Email: <input asp-for="Contact.Email" /></div>
            <input type="submit" asp-page-handler="JoinMailingList" value="Join our mailing list"/>
            <input type="submit" asp-page-handler="RequestQuote" value="Get a free quote"/>
        </form>
    </div>
</div>
```

The form in this example has two submit buttons, each using the new `FormActionTagHelper` in conjunction to submit to a different URL. The `asp-handler` attribute is a companion to `asp-page` and generates URLs that submit to each of the handler methods defined by the page. We don't need to specify `asp-page` because we're linking to the current page.

In this case, the URL path that submits to `OnPostJoinMailingListAsync` is `/Contact?handler=JoinMailingList` and the URL path that submits to `OnPostRequestQuoteAsync` is `/Contact?handler=RequestQuote`.

Customizing Routing

If you don't like seeing `?handler=RequestQuote` in the URL, you can change the route to put the handler name in the path portion of the URL. You can customize the route by adding a route template enclosed in quotes after the `@page` directive.

```
@page "{handler?}"
@inject ApplicationDbContext Db

...
```

This route will now put the handler name in the URL path instead of the query string.

You can use `@page` to add additional segments and parameters to a page's route, whatever's there is **appended** to the default route of the page. Using an absolute or virtual path to change the page's route (like `"~/Some/Other/Path"`) is not supported.

Configuration and settings

Use the extension method `AddRazorPagesOptions` on the MVC builder to configure advanced options such as the following example:

```
public class Startup
{
    public void ConfigureServices(IServiceCollections services)
    {
        services.AddMvc().AddRazorPagesOptions(options =>
        {
           ...
        });
    }

    ...
}
```

Currently you can use the `RazorPagesOptions` to set the root directory for pages, or add application model conventions for pages. We hope to enable more extensibility this way in the future.

See Razor view compilation to precompile views.

Testing and debugging ASP.NET Core

Integration testing

By Steve Smith

Integration testing ensures that an application's components function correctly when assembled together. ASP.NET Core supports integration testing using unit test frameworks and a built-in test web host that can be used to handle requests without network overhead.

View or download sample code

Introduction to integration testing

Integration tests verify that different parts of an application work correctly together. Unlike Unit testing, integration tests frequently involve application infrastructure concerns, such as a database, file system, network resources, or web requests and responses. Unit tests use fakes or mock objects in place of these concerns, but the purpose of integration tests is to confirm that the system works as expected with these systems.

Integration tests, because they exercise larger segments of code and because they rely on infrastructure elements, tend to be orders of magnitude slower than unit tests. Thus, it's a good idea to limit how many integration tests you write, especially if you can test the same behavior with a unit test.

Note

If some behavior can be tested using either a unit test or an integration test, prefer the unit test, since it will be almost always be faster. You might have dozens or hundreds of unit tests with many different inputs but just a handful of integration tests covering the most important scenarios.

Testing the logic within your own methods is usually the domain of unit tests. Testing how your application works within its framework, for example with ASP.NET Core, or with a database is where integration tests come into play. It doesn't take too many integration tests to confirm that you're able to write a row to the database and read it back. You don't need to test every possible permutation of your data access code - you only need to test enough to give you confidence that your application is working properly.

Integration testing ASP.NET Core

To get set up to run integration tests, you'll need to create a test project, add a reference to your ASP.NET Core web project, and install a test runner. This process is described in the Unit testing documentation, along with more detailed instructions on running tests and recommendations for naming your tests and test classes.

Note

Separate your unit tests and your integration tests using different projects. This helps ensure you don't accidentally introduce infrastructure concerns into your unit tests and lets you easily choose which set of tests to run.

The Test Host

ASP.NET Core includes a test host that can be added to integration test projects and used to host ASP.NET Core applications, serving test requests without the need for a real web host. The provided sample includes an integration test project which has been configured to use xUnit and the Test Host. It uses the `Microsoft.AspNetCore.TestHost` NuGet package.

Once the `Microsoft.AspNetCore.TestHost` package is included in the project, you'll be able to create and configure a `TestServer` in your tests. The following test shows how to verify that a request made to the root of a site returns "Hello World!" and should run successfully against the default ASP.NET Core Empty Web template created by Visual Studio.

```
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);
    }
}
```

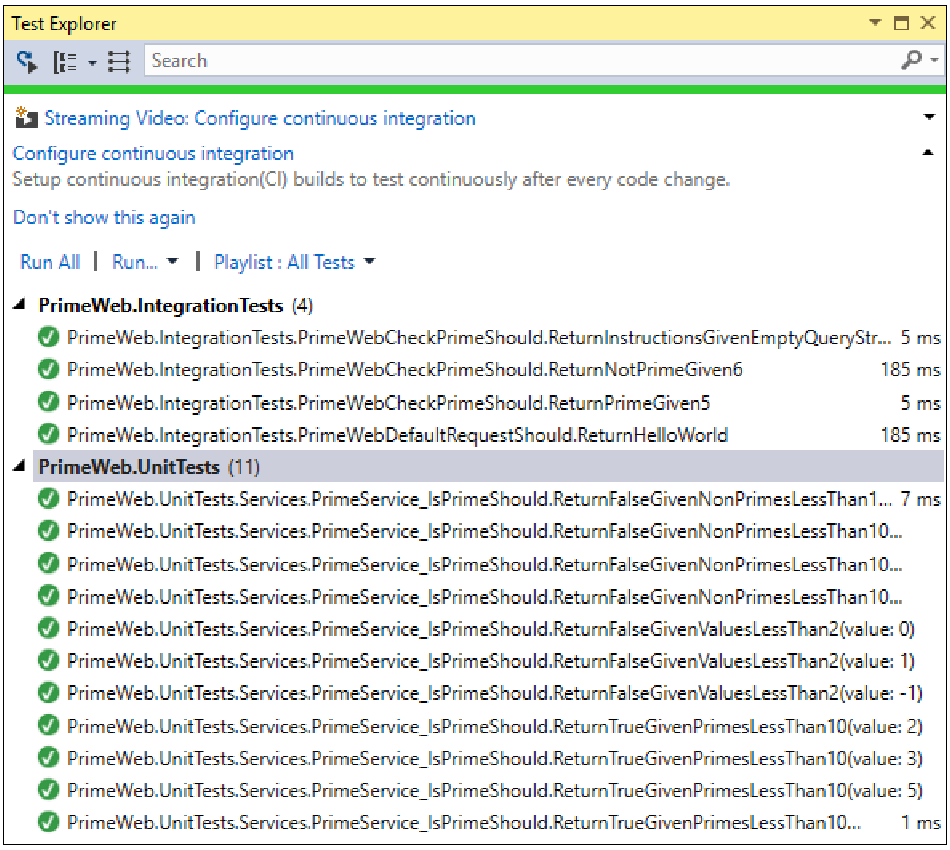
This test is using the Arrange-Act-Assert pattern. The Arrange step is done in the constructor, which creates an instance of `TestServer`. A configured `WebHostBuilder` will be used to create a `TestHost`; in this example, the `Configure` method from the system under test (SUT)'s `Startup` class is passed to the `WebHostBuilder`. This method will be used to configure the request pipeline of the `TestServer` identically to how the SUT server would be configured.

In the Act portion of the test, a request is made to the `TestServer` instance for the "/" path, and the response is read back into a string. This string is compared with the expected string of "Hello World!". If they match, the test passes; otherwise, it fails.

Now you can add a few additional integration tests to confirm that the prime checking functionality works via the web application:

```
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);
    }
}
```

Note that you're not really trying to test the correctness of the prime number checker with these tests but rather that the web application is doing what you expect. You already have unit test coverage that gives you confidence in `PrimeService`, as you can see here:



You can learn more about the unit tests in the Unit testing article.

Refactoring to use middleware

Refactoring is the process of changing an application's code to improve its design without changing its behavior. It should ideally be done when there is a suite of passing tests, since these help ensure the system's behavior remains the same before and after the changes. Looking at the way in which the prime checking logic is implemented in the web application's `Configure` method, you see:

```
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!");
        }
    });
}
```

This code works, but it's far from how you would like to implement this kind of functionality in an ASP.NET Core application, even as simple as this is. Imagine what the `Configure` method would look like if you needed to add this much code to it every time you add another URL endpoint!

One option to consider is adding MVC to the application and creating a controller to handle the prime checking. However, assuming you don't currently need any other MVC functionality, that's a bit overkill.

You can, however, take advantage of ASP.NET Core middleware, which will help us encapsulate the prime checking logic in its own class and achieve better separation of concerns in the `Configure` method.

You want to allow the path the middleware uses to be specified as a parameter, so the middleware class expects a `RequestDelegate` and a `PrimeCheckerOptions` instance in its constructor. If the path of the request doesn't match what this middleware is configured to expect, you simply call the next middleware in the chain and do nothing further. The rest of the implementation code that was in `Configure` is now in the `Invoke` method.

Note

Since the middleware depends on the `PrimeService` service, you're also requesting an instance of this service with the constructor. The framework will provide this service via Dependency Injection, assuming it has been configured, for example in `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");
                }
            }
        }
    }
}
```

Since this middleware acts as an endpoint in the request delegate chain when its path matches, there is no call to `_next.Invoke` when this middleware handles the request.

With this middleware in place and some helpful extension methods created to make configuring it easier, the refactored `Configure` method looks like this:

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

    app.UsePrimeChecker();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Following this refactoring, you're confident that the web application still works as before, since your integration tests are all passing.

 Note

It's a good idea to commit your changes to source control after you complete a refactoring and your tests pass. If you're practicing Test Driven Development, consider adding Commit to your Red-Green-Refactor cycle.

Resources

Unit testing

Middleware

Testing controllers

Testing controllers

By Steve Smith

Controllers in ASP.NET MVC apps should be small and focused on user-interface concerns. Large controllers that deal with non-UI concerns are more difficult to test and maintain.

View or download sample from GitHub

Why Test Controllers

Controllers are a central part of any ASP.NET Core MVC application. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production. It's important to avoid placing unnecessary responsibilities within your controllers and ensure your tests focus only on controller responsibilities.

Controller logic should be minimal and not be focused on business logic or infrastructure concerns (for example, data access). Test controller logic, not the framework. Test how the controller *behaves* based on valid or invalid inputs. Test controller responses based on the result of the business operation it performs.

Typical controller responsibilities:

Verify `ModelState.IsValid`

Return an error response if `ModelState` is invalid

Retrieve a business entity from persistence

Perform an action on the business entity

Save the business entity to persistence

Return an appropriate `IActionResult`

Unit Testing

Unit testing involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like filters, routing, or model binding. By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of integration testing.

If you're writing custom filters, routes, etc, you should unit test them, but not as part of your tests on a particular controller action. They should be tested in isolation.

 Tip

Create and run unit tests with Visual Studio.

To demonstrate unit testing, review the following controller. It displays a list of brainstorming sessions and allows new brainstorming sessions to be created with a POST:

```
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);
            }

            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });

            return RedirectToAction("Index");
        }
    }
}
```

The controller is following the explicit dependencies principle, expecting dependency injection to provide it with an instance of `IBrainstormSessionRepository`. This makes it fairly easy to test using a mock object framework, like Moq. The `HTTP GET Index` method has no looping or branching and only calls one method. To test this `Index` method, we need to verify that a `ViewResult` is returned, with a `ViewModel` from the repository's `List` method.

```
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;
        }
    }
}
```

The `HomeController` `HTTP POST Index` method (shown above) should verify:

The action method returns a Bad Request `ViewResult` with the appropriate data when `ModelState.IsValid` is `false`

The `Add` method on the repository is called and a `RedirectToActionResult` is returned with the correct arguments when `ModelState.IsValid` is true.

Invalid model state can be tested by adding errors using `AddModelError` as shown in the first test below.

```
[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();
}
```

The first test confirms when `ModelState` is not valid, the same `ViewResult` is returned as for a `GET` request. Note that the test doesn't attempt to pass in an invalid model. That wouldn't work anyway since model binding isn't running (though an integration test would use exercise model binding). In this case, model binding is not being tested. These unit tests are only testing what the code in the action method does.

The second test verifies that when `ModelState` is valid, a new `BrainstormSession` is added (via the repository), and the method returns a `RedirectToActionResult` with the expected properties. Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows it to be verified in the test. This is done with the call to `mockRepo.Verify`, which will fail the test if the expected method was not called.

Note

The Moq library used in this sample makes it easy to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

Another controller in the app displays information related to a particular brainstorming session. It includes some logic to deal with invalid id values:

```
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("Index", "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);
        }
    }
}
```

The controller action has three cases to test, one for each `return` statement:

```
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;
    }
```

```
    }
}
```

The app exposes functionality as a web API (a list of ideas associated with a brainstorming session and a method for adding new ideas to a session):

```
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);
        }
    }
}
```

The `ForSession` method returns a list of `IdeaDTO` types. Avoid returning your business domain entities directly via API calls, since frequently they include more data than the API client requires, and they unnecessarily couple your app's internal domain model with the API you expose externally. Mapping between domain entities and the types you will return over the wire can be done manually (using a LINQ `Select` as shown here) or using a library like AutoMapper

The unit tests for the `Create` and `ForSession` API methods:

```
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;
        }
    }
}
```

As stated previously, to test the behavior of the method when `ModelState` is invalid, add a model error to the controller as part of the test. Don't try to test model validation or model binding in your unit tests - just test your action method's behavior when confronted with a particular `ModelState` value.

The second test depends on the repository returning null, so the mock repository is configured to return null. There's no need to create a test database (in memory or otherwise) and construct a query that will return this result - it can be done in a single statement as shown.

The last test verifies that the repository's `Update` method is called. As we did previously, the mock is called with `Verifiable` and then the mocked repository's `Verify` method is called to confirm the verifiable method was executed. It's not a unit test responsibility to ensure that the `Update` method saved the data; that can be done with an integration test.

Integration Testing

Integration testing is done to ensure separate modules within your app work correctly together. Generally, anything you can test with a unit test, you can also test with an integration test, but the reverse isn't true. However, integration tests tend to be much slower than unit tests. Thus, it's best to test whatever you can with unit tests, and use integration tests for scenarios that involve multiple collaborators.

Although they may still be useful, mock objects are rarely used in integration tests. In unit testing, mock objects are an effective way to control how collaborators outside of the unit being tested should behave for the purposes of the test. In an integration test, real collaborators are used to confirm the whole subsystem works together correctly.

Application State

One important consideration when performing integration testing is how to set your app's state. Tests need to run independent of one another, and so each test should start with the app in a known state. If your app doesn't use a database or have any persistence, this may not be an issue. However, most real-world apps persist their state to some kind of data store, so any modifications made by one test could impact another test unless the data store is reset. Using the built-in `TestServer`, it's very straightforward to host ASP.NET Core apps within our integration tests, but that doesn't necessarily grant access to the data it will use. If you're using an actual database, one approach is to have the app connect to a test database, which your tests can access and ensure is reset to a known state before each test executes.

In this sample application, I'm using Entity Framework Core's InMemoryDatabase support, so I can't just connect to it from my test project. Instead, I expose an `InitializeDatabase` method from the app's `Startup` class, which I call when the app starts up if it's in the `Development` environment. My integration tests automatically benefit from this as long as they set the environment to `Development`. I don't have to worry about resetting the database, since the InMemoryDatabase is reset each time the app restarts.

The `Startup` class:

```
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());

            services.AddMvc();

            services.AddScoped<IBrainstormSessionRepository,
                EFStormSessionRepository>();
        }

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

```
        {
            loggerFactory.AddConsole(LogLevel.Warning);

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();

                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;
        }
    }
}
```

You'll see the `GetTestSession` method used frequently in the integration tests below.

Accessing Views

Each integration test class configures the `TestServer` that will run the ASP.NET Core app. By default, `TestServer` hosts the web app in the folder where it's running - in this case, the test project folder. Thus, when you attempt to test controller actions that return `ViewResult`, you may see this error:

```
The view 'Index' was not found. The following locations were searched:
(list of locations)
```

To correct this issue, you need to configure the server's content root, so it can locate the views for the project being tested. This is done by a call to `UseContentRoot` in the `TestFixture` class, shown below:

```
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;
using Microsoft.Extensions.PlatformAbstractions;

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 const string SolutionName = "TestingControllersSample.sln";
        private readonly TestServer _server;

        public TestFixture()
            : this(Path.Combine("src"))
        {
        }

        protected TestFixture(string solutionRelativeTargetProjectParentDir)
        {
            var startupAssembly = typeof(TStartup).GetTypeInfo().Assembly;
            var contentRoot = GetProjectPath(solutionRelativeTargetProjectParentDir, 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 that 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 path that we wish to test
        /// </summary>
        /// <param name="solutionRelativePath">
        /// The parent directory of the target project
```

```
        /// 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 solutionRelativePath, 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 = PlatformServices.Default.Application.ApplicationBasePath;

            // Find the folder which contains the solution file. We then use this information to find the
target
            // project which we want to test.
            var directoryInfo = new DirectoryInfo(applicationBasePath);
            do
            {
                var solutionFileInfo = new FileInfo(Path.Combine(directoryInfo.FullName, SolutionName));
                if (solutionFileInfo.Exists)
                {
                    return Path.GetFullPath(Path.Combine(directoryInfo.FullName, solutionRelativePath,
projectName));
                }

                directoryInfo = directoryInfo.Parent;
            }
            while (directoryInfo.Parent != null);

            throw new Exception($"Solution root could not be located using application root
{applicationBasePath}.");
        }
    }
}
```

The `TestFixture` class is responsible for configuring and creating the `TestServer`, setting up an `HttpClient` to communicate with the `TestServer`. Each of the integration tests uses the `Client` property to connect to the test server and make a request.

```
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.True(responseString.Contains(testSession.Name));
        }

        [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());
        }
    }
}
```

In the first test above, the `responseString` holds the actual rendered HTML from the View, which can be inspected to confirm it contains expected results.

The second test constructs a form POST with a unique session name and POSTs it to the app, then verifies that the expected redirect is returned.

API Methods

If your app exposes web APIs, it's a good idea to have automated tests confirm they execute as expected. The built-in `TestServer` makes it easy to test web APIs. If your API methods are using model binding, you should always check `ModelState.IsValid`, and integration tests are the right place to confirm that your model validation is working properly.

The following set of tests target the `Create` method in the IdeasController class shown above:

```
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.True(returnedSession.Ideas.Any(i => i.Name == testIdeaName));
    }

    [Fact]
    public async Task ForSessionReturnsNotFoundForBadSessionId()
    {
        // Arrange & Act
        var response = await _client.GetAsync("/api/ideas/forsession/500");

        // Assert
        Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
    }

    [Fact]
```

```
        [Fact]
        public async Task ForSessionReturnsIdeasForValidSessionId()
        {
            // 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);
        }
    }
}
```

Unlike integration tests of actions that returns HTML views, web API methods that return results can usually be deserialized as strongly typed objects, as the last test above shows. In this case, the test deserializes the result to a `BrainstormSession` instance, and confirms that the idea was correctly added to its collection of ideas.

You'll find additional examples of integration tests in this article's sample project.

Working with Data in ASP.NET Core

Getting started with ASP.NET Core and Entity Framework Core using Visual Studio

This series of tutorials teaches you how to create ASP.NET Core MVC web applications that use Entity Framework Core for data access. The tutorials require Visual Studio 2017.

Getting started with ASP.NET Core and Entity Framework 6

By Paweł Grudzień, Damien Pontifex, and Tom Dykstra

This article shows how to use Entity Framework 6 in an ASP.NET Core application.

Overview

To use Entity Framework 6, your project has to compile against the full .NET Framework, as Entity Framework 6 does not support .NET Core. If you need cross-platform features you will need to upgrade to Entity Framework Core.

The recommended way to use Entity Framework 6 in an ASP.NET Core application is to put the EF6 context and model classes in a class library project that targets the full framework. Add a reference to the class library from the ASP.NET Core project. See the sample Visual Studio solution with EF6 and ASP.NET Core projects.

You can't put an EF6 context in an ASP.NET Core project because .NET Core projects don't support all of the functionality that EF6 commands such as *Enable-Migrations* require.

Regardless of project type in which you locate your EF6 context, only EF6 command-line tools work with an EF6 context. For example, `Scaffold-DbContext` is only available in Entity Framework Core. If you need to do reverse engineering of a database into an EF6 model, see Code First to an Existing Database.

Reference full framework and EF6 in the ASP.NET Core project

Your ASP.NET Core project needs to reference the full .NET framework and EF6. For example, the *.csproj* file of your ASP.NET Core project will look similar to the following example (only relevant parts of the file are shown).

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>MVCCore</AssemblyName>
  <OutputType>Exe</OutputType>
  <PackageId>MVCCore</PackageId>
</PropertyGroup>
```

If you're creating a new project, use the **ASP.NET Core Web Application (.NET Framework)** template.

Handle connection strings

The EF6 command-line tools that you'll use in the EF6 class library project require a default constructor so they can instantiate the context. But you'll probably want to specify the connection string to use in the ASP.NET Core project, in which case your context constructor must have a parameter that lets you pass in the connection string. Here's an example.

```
public class SchoolContext : DbContext
{
    public SchoolContext(string connString) : base(connString)
    {
    }
```

Since your EF6 context doesn't have a parameterless constructor, your EF6 project has to provide an implementation of IDbContextFactory. The EF6 command-line tools will find and use that implementation so they can instantiate the context. Here's an example.

```
public class SchoolContextFactory : IDbContextFactory<SchoolContext>
{
    public SchoolContext Create()
    {
        return new EF6.SchoolContext("Server=
(localdb)\\mssqllocaldb;Database=EF6MVCCore;Trusted_Connection=True;MultipleActiveResultSets=true");
    }
}
```

In this sample code, the `IDbContextFactory` implementation passes in a hard-coded connection string. This is the connection string that the command-line tools will use. You'll want to implement a strategy to ensure that the class library uses the same connection string that the calling application uses. For example, you could get the value from an environment variable in both projects.

Set up dependency injection in the ASP.NET Core project

In the Core project's *Startup.cs* file, set up the EF6 context for dependency injection (DI) in `ConfigureServices`. EF context objects should be scoped for a per-request lifetime.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddScoped<SchoolContext>(_ => new
SchoolContext(Configuration.GetConnectionString("DefaultConnection")));
}
```

You can then get an instance of the context in your controllers by using DI. The code is similar to what you'd write for an EF Core context:

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context)
    {
        _context = context;
    }
```

Sample application

For a working sample application, see the sample Visual Studio solution that accompanies this article.

This sample can be created from scratch by the following steps in Visual Studio:

Create a solution.

Add New Project > Web > ASP.NET Core Web Application (.NET Framework)

Add New Project > Windows Classic Desktop > Class Library (.NET Framework)

In **Package Manager Console** (PMC) for both projects, run the command `Install-Package Entityframework`.

In the class library project, create data model classes and a context class, and an implementation of `IDbContextFactory`.

In PMC for the class library project, run the commands `Enable-Migrations` and `Add-Migration Initial`. If you have set the ASP.NET Core project as the startup project, add `-StartupProjectName EF6` to these commands.

In the Core project, add a project reference to the class library project.

In the Core project, in *Startup.cs*, register the context for DI.

In the Core project, in *appsettings.json*, add the connection string.

In the Core project, add a controller and view(s) to verify that you can read and write data. (Note that ASP.NET Core MVC scaffolding won't work with the EF6 context referenced from the class library.)

Summary

This article has provided basic guidance for using Entity Framework 6 in an ASP.NET Core application.

Additional Resources

[Entity Framework - Code-Based Configuration]

Azure Storage in ASP.NET Core

Introduction to using Gulp in ASP.NET Core

By Erik Reitan, Scott Addie, Daniel Roth, and Shayne Boyer

In a typical modern web application, the build process might:

Bundle and minify JavaScript and CSS files.

Run tools to call the bundling and minification tasks before each build.

Compile LESS or SASS files to CSS.

Compile CoffeeScript or TypeScript files to JavaScript.

A *task runner* is a tool which automates these routine development tasks and more. Visual Studio provides built-in support for two popular JavaScript-based task runners: Gulp and Grunt.

Introducing Gulp

Gulp is a JavaScript-based streaming build toolkit for client-side code. It is commonly used to stream client-side files through a series of processes when a specific event is triggered in a build environment. For instance, Gulp can be used to automate bundling and minification or the cleansing of a development environment before a new build.

A set of Gulp tasks is defined in *gulpfile.js*. The following JavaScript includes Gulp modules and specifies file paths to be referenced within the forthcoming tasks:

The above code specifies which Node modules are required. The `require` function imports each module so that the dependent tasks can utilize their features. Each of the imported modules is assigned to a variable. The modules can be located either by name or path. In this example, the modules named `gulp`, `rimraf`, `gulp-concat`, `gulp-cssmin`, and `gulp-uglify` are retrieved by name. Additionally, a series of paths are created so that the locations of CSS and JavaScript files can be reused and referenced within the tasks. The following table provides descriptions of the modules included in *gulpfile.js*.

MODULE NAME	DESCRIPTION
gulp	The Gulp streaming build system. For more information, see gulp.
rimraf	A Node deletion module. For more information, see rimraf.
gulp-concat	A module that concatenates files based on the operating system's newline character. For more information, see gulp-concat.
gulp-cssmin	A module that minifies CSS files. For more information, see gulp-cssmin.
gulp-uglify	A module that minifies *.js* files. For more information, see gulp-uglify.

Once the requisite modules are imported, the tasks can be specified. Here there are six tasks registered, represented by the following code:

The following table provides an explanation of the tasks specified in the code above:

TASK NAME	DESCRIPTION
clean:js	A task that uses the rimraf Node deletion module to remove the minified version of the site.js file.

TASK NAME	DESCRIPTION
clean:css	A task that uses the rimraf Node deletion module to remove the minified version of the site.css file.
clean	A task that calls the `clean:js` task, followed by the `clean:css` task.
min:js	A task that minifies and concatenates all .js files within the js folder. The .min.js files are excluded.
min:css	A task that minifies and concatenates all .css files within the css folder. The .min.css files are excluded.
min	A task that calls the `min:js` task, followed by the `min:css` task.

Running default tasks

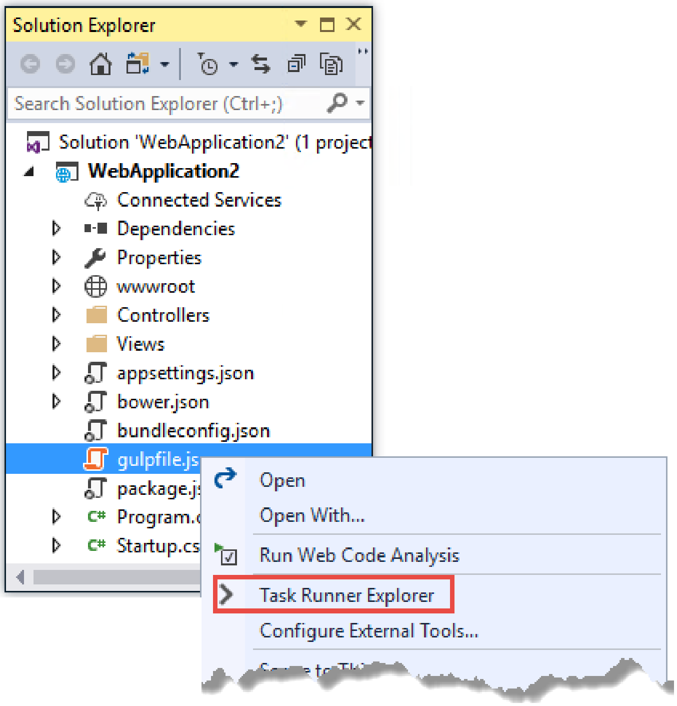
If you haven't already created a new Web app, create a new ASP.NET Web Application project in Visual Studio.

Add a new JavaScript file to your project and name it *gulpfile.js*, then copy the following code.

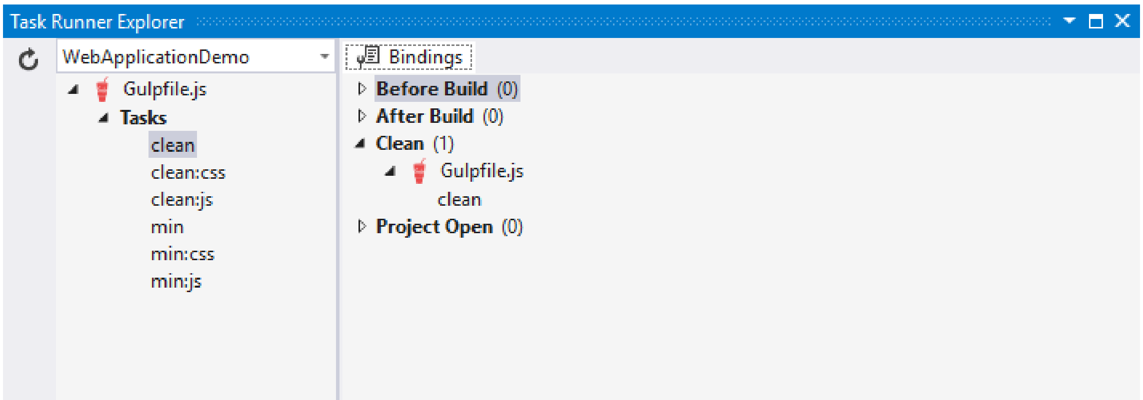
Open the *package.json* file (add if not there) and add the following.

```
{
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "2.0.1",
    "rimraf": "2.6.1"
  }
}
```

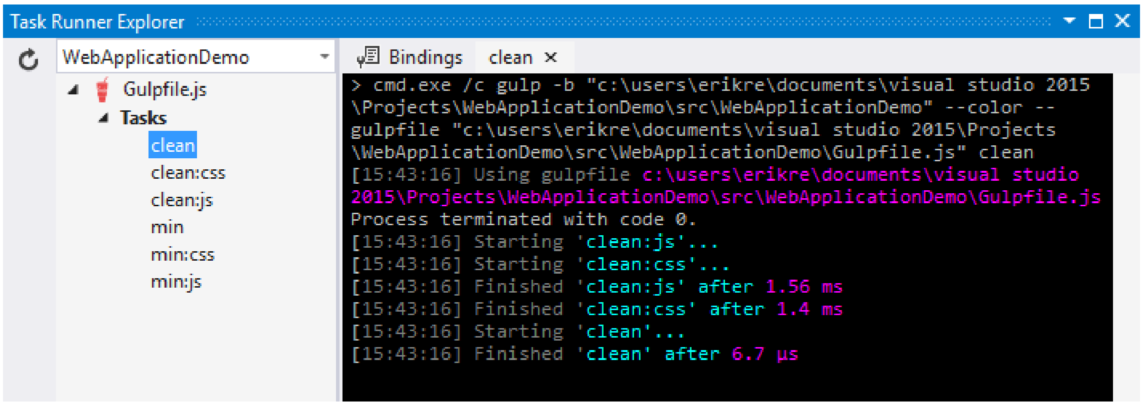
In **Solution Explorer**, right-click *gulpfile.js*, and select **Task Runner Explorer**.



Task Runner Explorer shows the list of Gulp tasks. (You might have to click the **Refresh** button that appears to the left of the project name.)

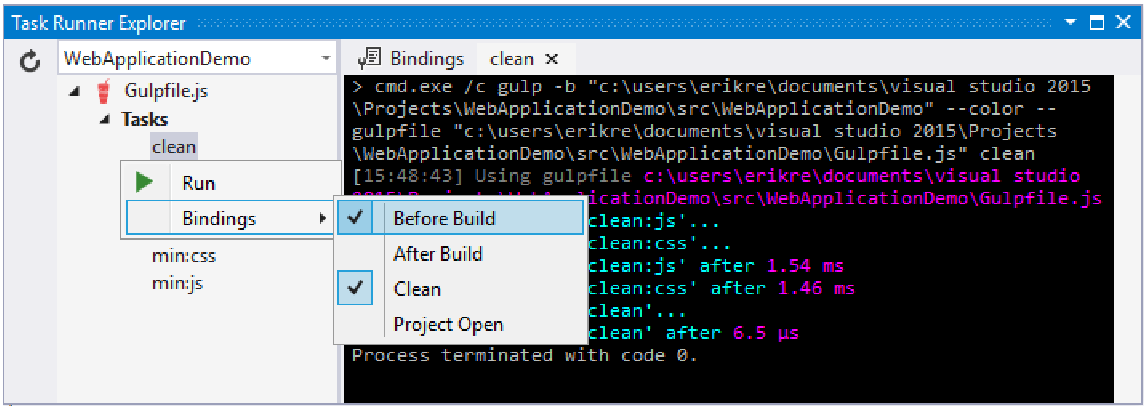


Underneath **Tasks** in **Task Runner Explorer**, right-click **clean**, and select **Run** from the pop-up menu.



Task Runner Explorer will create a new tab named **clean** and execute the clean task as it is defined in *gulpfile.js*.

Right-click the **clean** task, then select **Bindings** > **Before Build**.



The **Before Build** binding configures the clean task to run automatically before each build of the project.

The bindings you set up with **Task Runner Explorer** are stored in the form of a comment at the top of your *gulpfile.js* and are effective only in Visual Studio. An alternative that doesn't require Visual Studio is to configure automatic execution of gulp tasks in your *.csproj* file. For example put this in your *.csproj* file:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp clean" />
</Target>
```

Now the clean task is executed when you run the project in Visual Studio or from a command prompt using the `dotnet run` command (run `npm install` first).

Defining and running a new task

To define a new Gulp task, modify *gulpfile.js*.

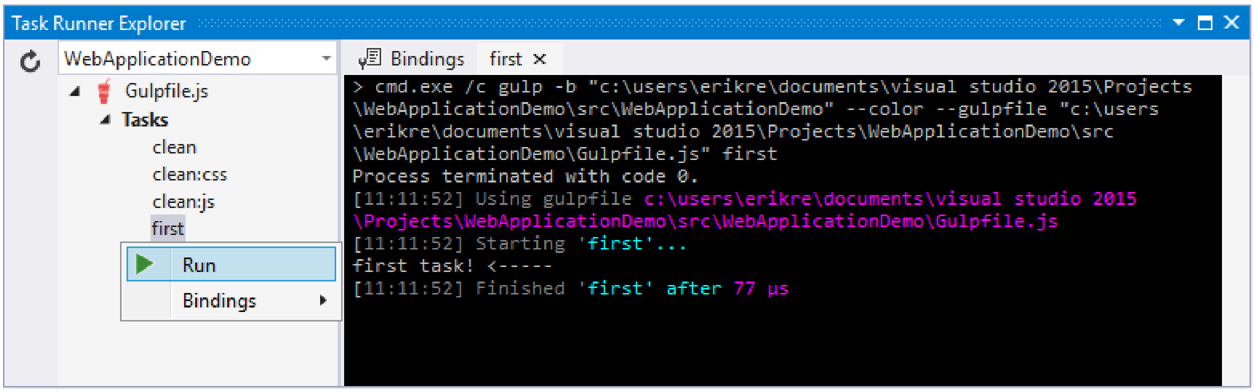
Add the following JavaScript to the end of *gulpfile.js*:

This task is named `first`, and it simply displays a string.

Save *gulpfile.js*.

In **Solution Explorer**, right-click *gulpfile.js*, and select *Task Runner Explorer*.

In **Task Runner Explorer**, right-click **first**, and select **Run**.



You'll see that the output text is displayed. If you are interested in examples based on a common scenario, see Gulp Recipes.

Defining and running tasks in a series

When you run multiple tasks, the tasks run concurrently by default. However, if you need to run tasks in a specific order, you must specify when each task is complete, as well as which tasks depend on the completion of another task.

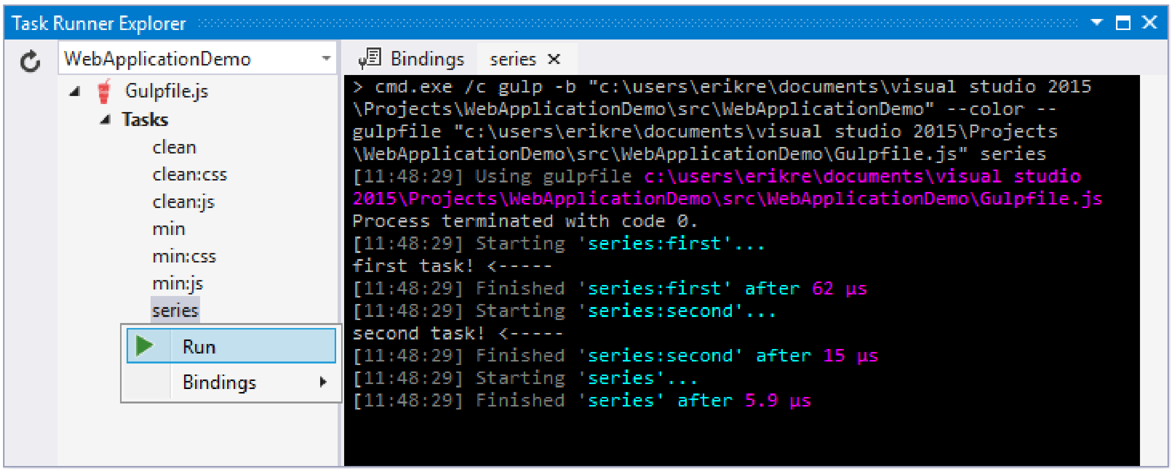
To define a series of tasks to run in order, replace the `first` task that you added above in *gulpfile.js* with the following:

You now have three tasks: `series:first`, `series:second`, and `series`. The `series:second` task includes a second parameter which specifies an array of tasks to be run and completed before the `series:second` task will run. As specified in the code above, only the `series:first` task must be completed before the `series:second` task will run.

Save *gulpfile.js*.

In **Solution Explorer**, right-click *gulpfile.js* and select **Task Runner Explorer** if it isn't already open.

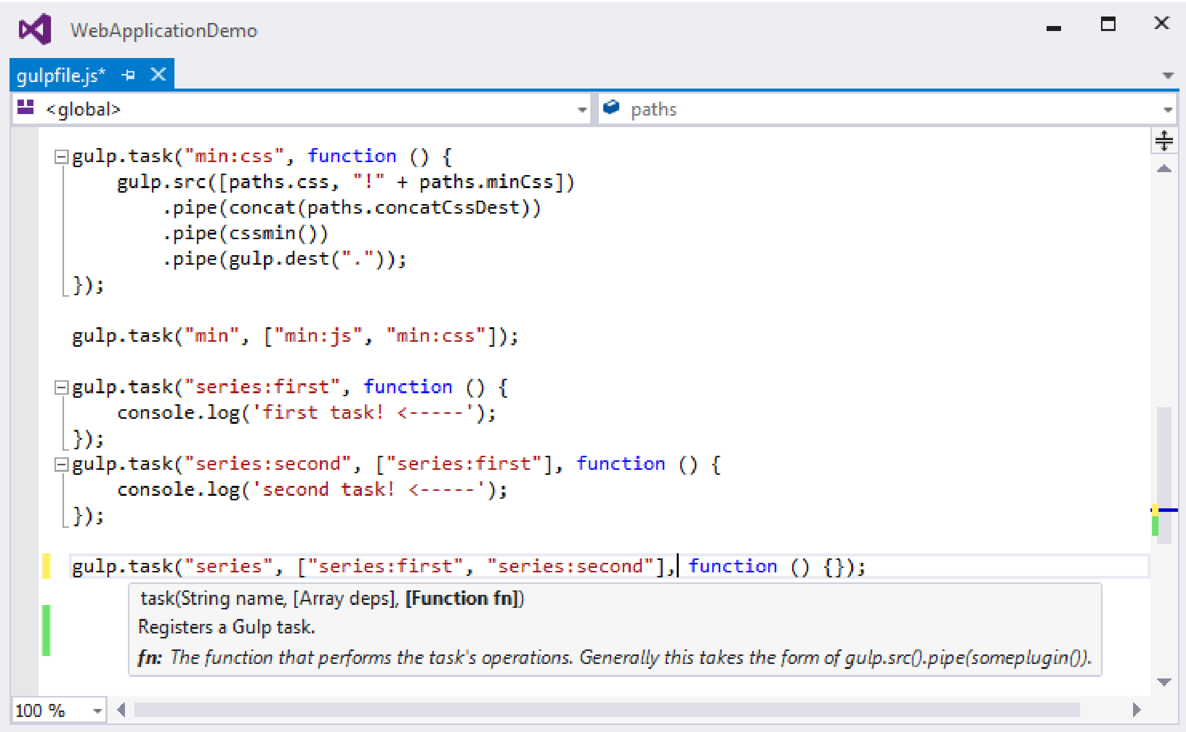
In **Task Runner Explorer**, right-click **series** and select **Run**.



IntelliSense

IntelliSense provides code completion, parameter descriptions, and other features to boost productivity and to decrease errors.

Gulp tasks are written in JavaScript; therefore, IntelliSense can provide assistance while developing. As you work with JavaScript, IntelliSense lists the objects, functions, properties, and parameters that are available based on your current context. Select a coding option from the pop-up list provided by IntelliSense to complete the code.



For more information about IntelliSense, see JavaScript IntelliSense.

Development, staging, and production environments

When Gulp is used to optimize client-side files for staging and production, the processed files are saved to a local staging and production location. The *_Layout.cshtml* file uses the **environment** tag helper to provide two different versions of CSS files. One version of CSS files is for development and the other version is optimized for both staging and production. In Visual Studio 2017, when you change the **ASPNETCORE_ENVIRONMENT** environment variable to `Production`, Visual Studio will build the Web app and link to the minimized CSS files. The following markup shows the **environment** tag helpers containing link tags to the `Development` CSS files and the minified `Staging, Production` CSS files.

```
<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-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>
```

Switching between environments

To switch between compiling for different environments, modify the **ASPNETCORE_ENVIRONMENT** environment variable's value.

In **Task Runner Explorer**, verify that the **min** task has been set to run **Before Build**.

In **Solution Explorer**, right-click the project name and select **Properties**.

The property sheet for the Web app is displayed.

Click the **Debug** tab.

Set the value of the **Hosting:Environment** environment variable to `Production`.

Press **F5** to run the application in a browser.

In the browser window, right-click the page and select **View Source** to view the HTML for the page.

Notice that the stylesheet links point to the minified CSS files.

Close the browser to stop the Web app.

In Visual Studio, return to the property sheet for the Web app and change the **Hosting:Environment** environment variable back to `Development`.

Press **F5** to run the application in a browser again.

In the browser window, right-click the page and select **View Source** to see the HTML for the page.

Notice that the stylesheet links point to the unminified versions of the CSS files.

For more information related to environments in ASP.NET Core, see Working with Multiple Environments.

Task and module details

A Gulp task is registered with a function name. You can specify dependencies if other tasks must run before the current task. Additional functions allow you to run and watch the Gulp tasks, as well as set the source (*src*) and destination (*dest*) of the files being modified. The following are the primary Gulp API functions:

GULP FUNCTION	SYNTAX	DESCRIPTION
task	`gulp.task(name[, deps], fn) { }`	The `task` function creates a task. The `name` parameter defines the name of the task. The `deps` parameter contains an array of tasks to be completed before this task runs. The `fn` parameter represents a callback function which performs the operations of the task.
watch	`gulp.watch(glob [, opts], tasks) { }`	The `watch` function monitors files and runs tasks when a file change occurs. The `glob` parameter is a `string` or `array` that determines which files to watch. The `opts` parameter provides additional file watching options.
src	`gulp.src(globs[, options]) { }`	The `src` function provides files that match the glob value(s). The `glob` parameter is a `string` or `array` that determines which files to read. The `options` parameter provides additional file options.
dest	`gulp.dest(path[, options]) { }`	The `dest` function defines a location to which files can be written. The `path` parameter is a string or function that determines the destination folder. The `options` parameter is an object that specifies output folder options.

For additional Gulp API reference information, see Gulp Docs API.

Gulp recipes

The Gulp community provides Gulp recipes. These recipes consist of Gulp tasks to address common scenarios.

Additional resources

Gulp documentation
Bundling and minification in ASP.NET Core
Using Grunt in ASP.NET Core

Using Grunt in ASP.NET Core

By Noel Rice

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio, though the ASP.NET project templates use Gulp by default (see Using Gulp).

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

grunt: The Grunt task runner package.

grunt-contrib-clean: A plugin that removes files or directories.

grunt-contrib-jshint: A plugin that reviews JavaScript code quality.

grunt-contrib-concat: A plugin that joins files into a single file.

grunt-contrib-uglify: A plugin that minifies JavaScript to reduce size.

grunt-contrib-watch: A plugin that watches file activity.

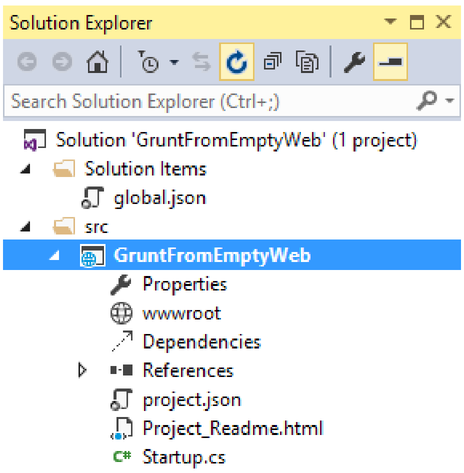
Preparing the application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

In Visual Studio, create a new `ASP.NET Web Application`.

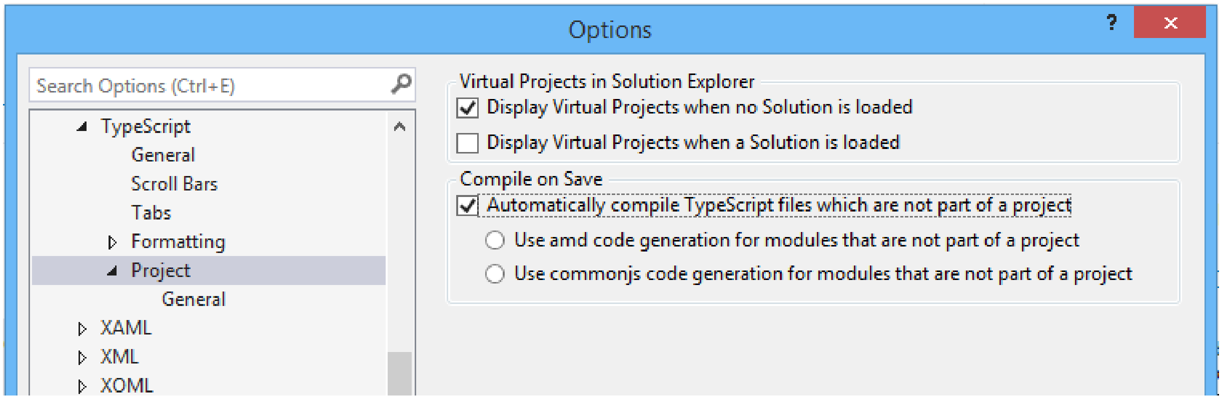
In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.

In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



Add a new folder named `TypeScript` to your project directory.

Before adding any files, let's make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. *Tools* > *Options* > *Text Editor* > *Typescript* > *Project*



Right-click the `TypeScript` directory and select **Add > New Item** from the context menu. Select the **JavaScript file** item and name the file *Tastes.ts* (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new *Tastes.js* file will appear with the JavaScript source).

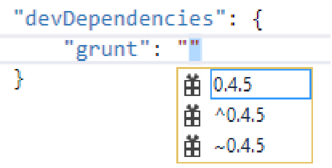
Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, *package.json*, and click the **Add** button.

In the *package.json* file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl-Space` if Intellisense does not appear).



Note

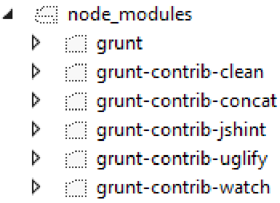
NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme ... Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

Add more dependencies to load grunt-contrib-* packages for *clean*, *jshint*, *concat*, *uglify*, and *watch* as shown in the example below. The versions do not need to match the example.

```
"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"
}
```

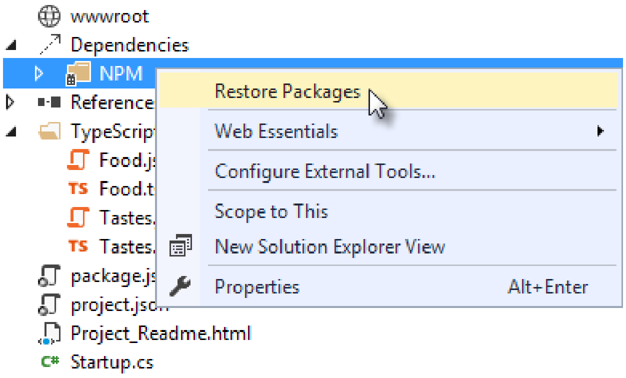
Save the *package.json* file.

The packages for each devDependencies item will download, along with any files that each package requires. You can find the package files in the `node_modules` directory by enabling the **Show All Files** button in the Solution Explorer.



Note

If you need to, you can manually restore dependencies in Solution Explorer by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named *Gruntfile.js* that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

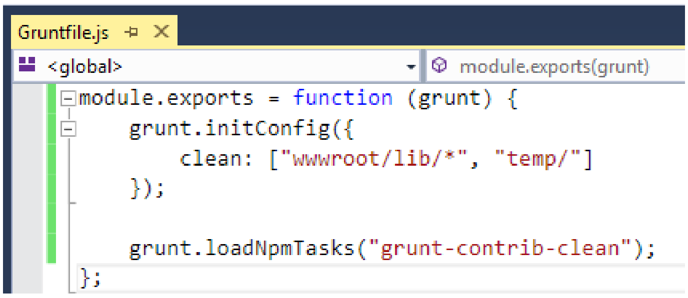
Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, *Gruntfile.js*, and click the **Add** button.

The initial code includes a module definition and the `grunt.initConfig()` method. The `initConfig()` is used to set options for each package, and the remainder of the module will load and register tasks.

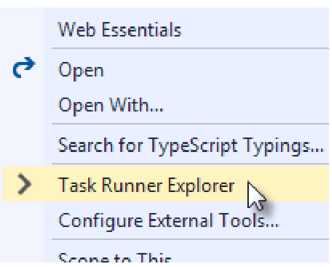
Inside the `initConfig()` method, add options for the `clean` task as shown in the example *Gruntfile.js* below. The clean task accepts an array of directory strings. This task removes files from wwwroot/lib and removes the entire /temp directory.

Below the initConfig() method, add a call to `grunt.loadNpmTasks()`. This will make the task runnable from Visual Studio.

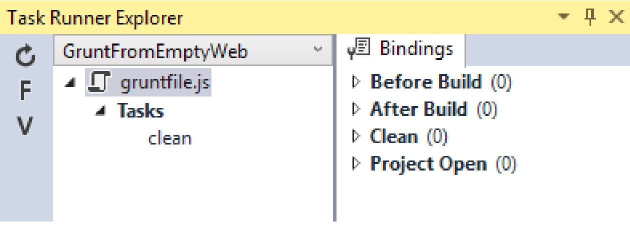
Save *Gruntfile.js*. The file should look something like the screenshot below.



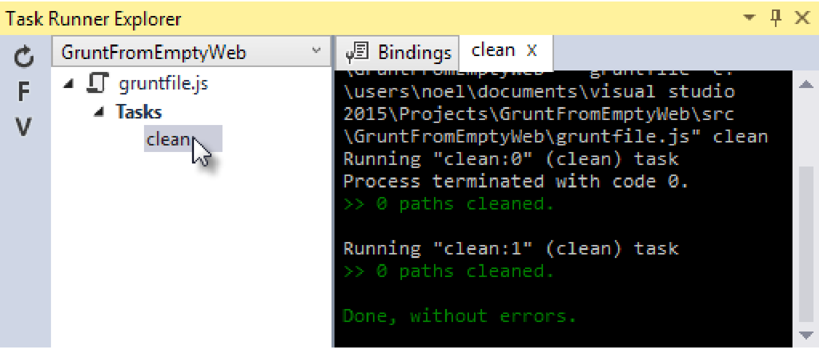
Right-click *Gruntfile.js* and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



Verify that `clean` shows under **Tasks** in the Task Runner Explorer.



Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.



Note

There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

In the initConfig() method, add an entry for `concat` using the code below.

The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that is produced.

Note

The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using Intellisense or assign your own.

Add the `jshint` task using the code below.

The jshint code-quality utility is run against every JavaScript file found in the temp directory.

Note

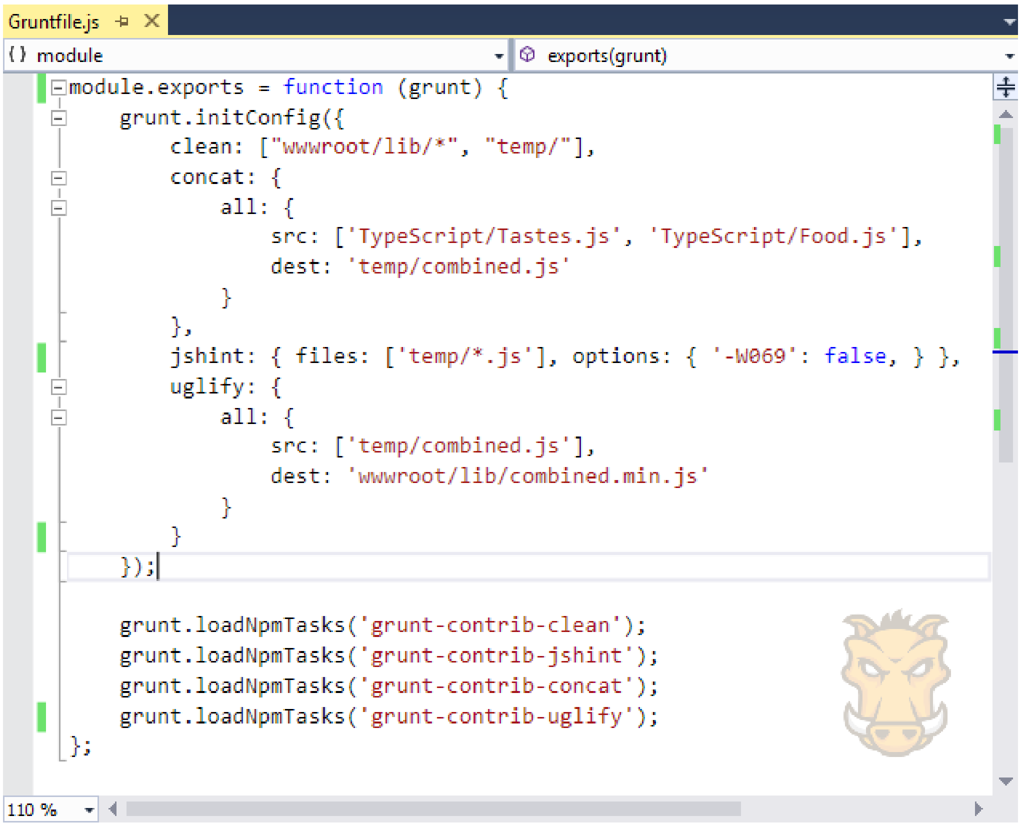
The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

Add the `uglify` task using the code below.

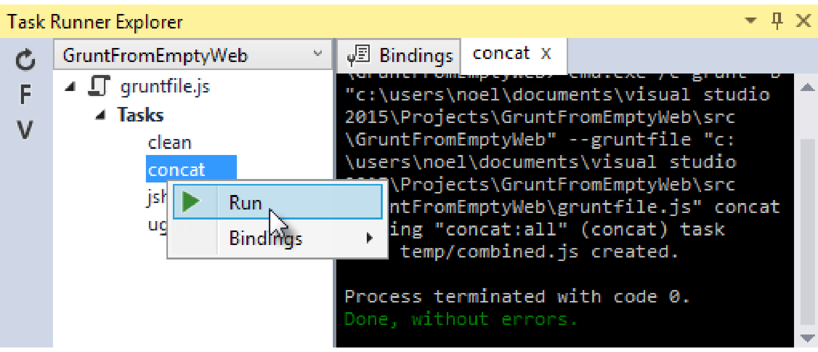
The task minifies the *combined.js* file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention *<file name>.min.js*.

Under the call grunt.loadNpmTasks() that loads grunt-contrib-clean, include the same call for jshint, concat and uglify using the code below.

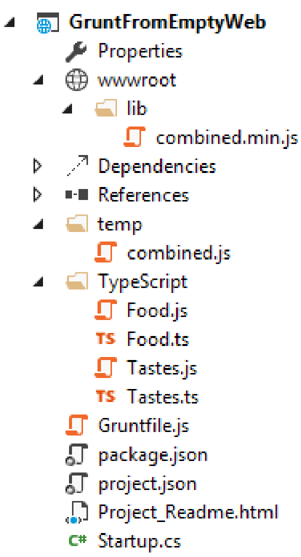
Save *Gruntfile.js*. The file should look something like the example below.



Notice that the Task Runner Explorer Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.



The concat task creates a new *combined.js* file and places it into the temp directory. The jshint task simply runs and doesn't produce output. The uglify task creates a new *combined.min.js* file and places it into wwwroot/lib. On completion, the solution should look something like the screenshot below:



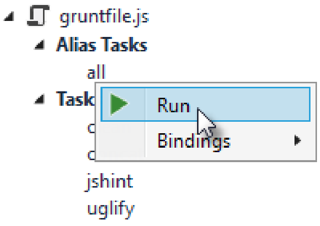
Note

For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

All together now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.

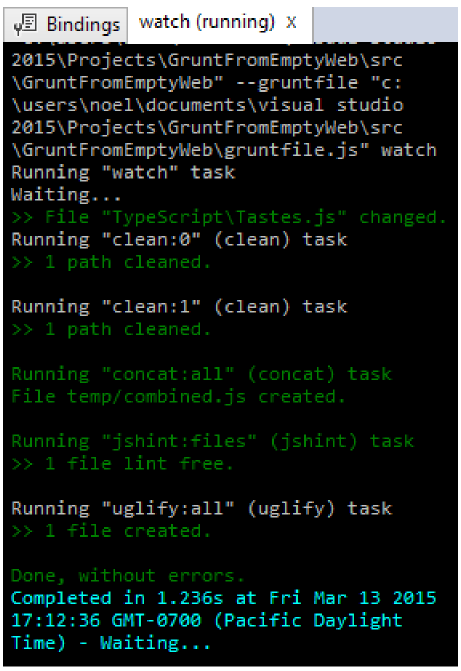


Watching for changes

A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

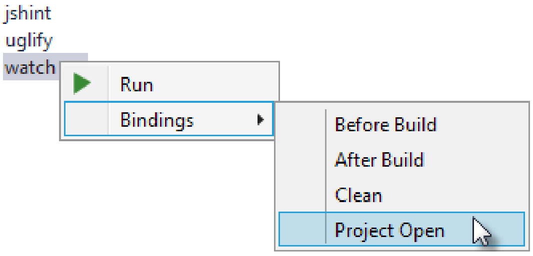
Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting…" message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.



Binding to Visual Studio events

Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let's bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings > Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task will start running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Additional resources

Using Gulp

Manage client-side packages with Bower in ASP.NET Core

By Rick Anderson, Noel Rice, and Scott Addie

Bower calls itself "A package manager for the web." Within the .NET ecosystem, it fills the void left by NuGet's inability to deliver static content files. For ASP.NET Core projects, these static files are inherent to client-side libraries like jQuery and Bootstrap. For .NET libraries, you still use NuGet package manager.

New projects created with the ASP.NET Core project templates set up the client-side build process. jQuery and Bootstrap are installed, and Bower is supported.

Client-side packages are listed in the *bower.json* file. The ASP.NET Core project templates configures *bower.json* with jQuery, jQuery validation, and Bootstrap.

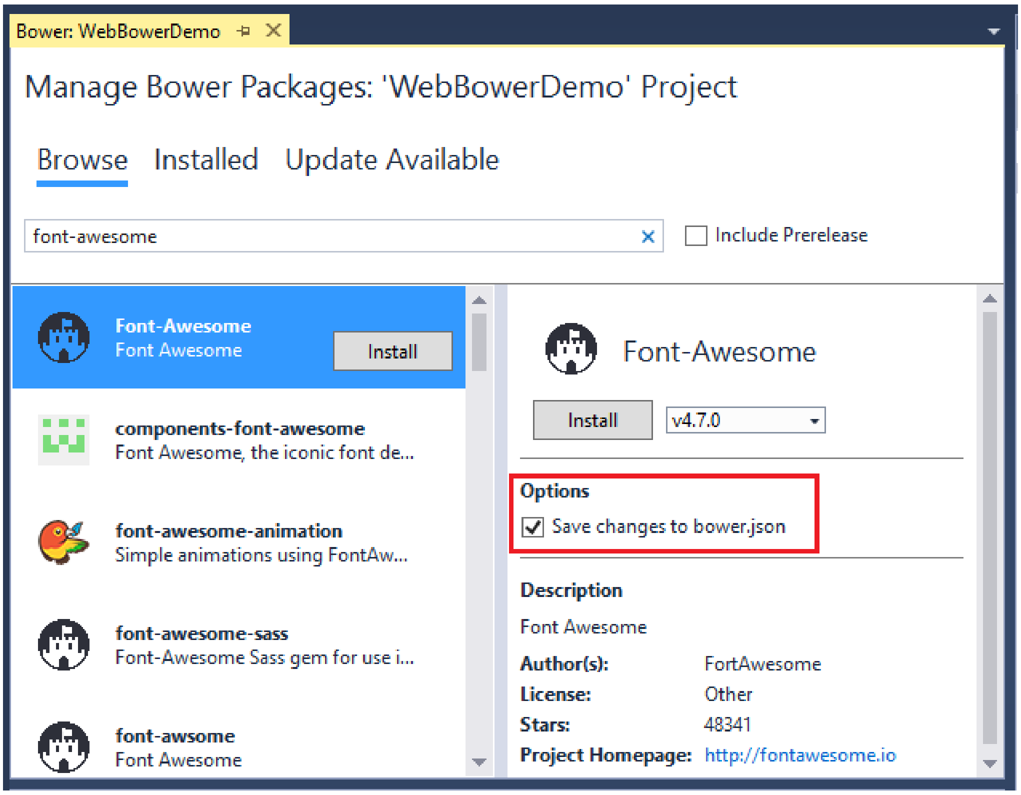
In this tutorial, we'll add support for Font Awesome. Bower packages can be installed with the **Manage Bower Packages** UI or manually in the *bower.json* file.

Installation via Manage Bower Packages UI

Create a new ASP.NET Core Web app with the **ASP.NET Core Web Application (.NET Core)** template. Select **Web Application** and **No Authentication**.

Right-click the project in Solution Explorer and select **Manage Bower Packages** (alternatively from the main menu, **Project** > **Manage Bower Packages**).

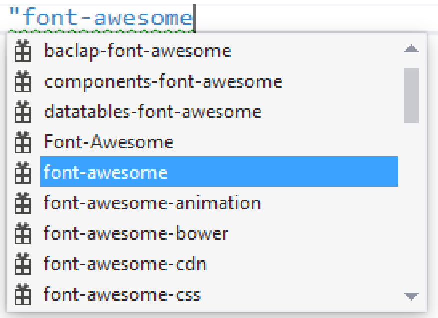
In the **Bower: <project name>** window, click the "Browse" tab, and then filter the packages list by entering `font-awesome` in the search box:

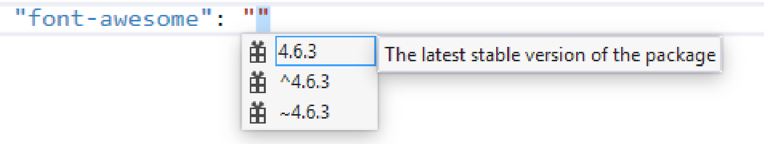


Confirm that the "Save changes to *bower.json*" checkbox is checked. Select a version from the drop-down list and click the **Install** button. The **Output** window shows the installation details.

Manual installation in bower.json

Open the *bower.json* file and add "font-awesome" to the dependencies. IntelliSense shows the available packages. When a package is selected, the available versions are displayed. The images below are older and will not match what you see.





Bower uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme <major>.<minor>.<patch>. IntelliSense simplifies semantic versioning by showing only a few common choices. The top item in the IntelliSense list (4.6.3 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version.

Save the *bower.json* file. Visual Studio watches the *bower.json* file for changes. Upon saving, the *bower install* command is executed. See the Output window's **Bower/npm** view for the exact command executed.

Open the *.bowerrc* file under *bower.json*. The `directory` property is set to *wwwroot/lib* which indicates the location Bower will install the package assets.

```
{
 "directory": "wwwroot/lib"
}
```

You can use the search box in Solution Explorer to find and display the font-awesome package.

Open the *Views\Shared_Layout.cshtml* file and add the font-awesome CSS file to the environment Tag Helper for `Development`. From Solution Explorer, drag and drop *font-awesome.css* inside the `<environment names="Development">` element.

```
<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>
```

In a production app you would add *font-awesome.min.css* to the environment tag helper for `Staging,Production`.

Replace the contents of the *Views\Home\About.cshtml* Razor file with the following markup:

```
@{
    ViewData["Title"] = "About";
}

<div class="list-group">
    <a class="list-group-item" href="#"><i class="fa fa-home fa-fw" aria-hidden="true"></i>  Home</a>
    <a class="list-group-item" href="#"><i class="fa fa-book fa-fw" aria-hidden="true"></i>  Library</a>
    <a class="list-group-item" href="#"><i class="fa fa-pencil fa-fw" aria-hidden="true"></i> 
Applications</a>
    <a class="list-group-item" href="#"><i class="fa fa-cog fa-fw" aria-hidden="true"></i>  Settings</a>
</div>
```

Run the app and navigate to the About view to verify the font-awesome package works.

Exploring the client-side build process

Most ASP.NET Core project templates are already configured to use Bower. This next walkthrough starts with an empty ASP.NET Core project and adds each piece manually, so you can get a feel for how Bower is used in a project. You see can what happens to the project structure and the runtime output as each configuration change is made.

The general steps to use the client-side build process with Bower are:

Define packages used in your project.
Reference packages from your web pages.

Define packages

Once you list packages in the *bower.json* file, Visual Studio will download them. The following example uses Bower to load jQuery and Bootstrap to the *wwwroot* folder.

Create a new ASP.NET Core Web app with the **ASP.NET Core Web Application (.NET Core)** template. Select the **Empty** project template and click **OK**.

In Solution Explorer, right-click the project > **Add New Item** and select **Bower Configuration File**. Note: A *.bowerrc* file is also added.

Open *bower.json*, and add jquery and bootstrap to the `dependencies` section. The resulting *bower.json* file will look like the following example. The versions will change over time and may not match the image below.

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "jquery": "3.1.1",
    "bootstrap": "3.3.7"
  }
}
```

Save the *bower.json* file.

Verify the project includes the *bootstrap* and *jQuery* directories in *wwwroot/lib*. Bower uses the *.bowerrc* file to install the assets in *wwwroot/lib*.

Note: The "Manage Bower Packages" UI provides an alternative to manual file editing.

Enable static files

Add the `Microsoft.AspNetCore.StaticFiles` NuGet package to the project.
Enable static files to be served with the Static file middleware middleware. Add a call to UseStaticFiles to the `Configure` method of `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!");
        });
    }
}
```

Reference packages

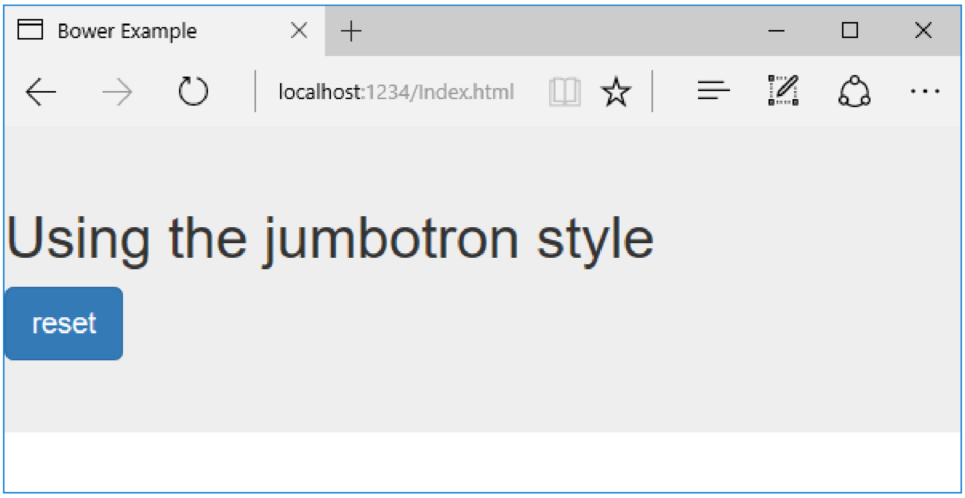
In this section, you will create an HTML page to verify it can access the deployed packages.

Add a new HTML page named *Index.html* to the *wwwroot* folder. Note: You must add the HTML file to the *wwwroot* folder. By default, static content cannot be served outside *wwwroot*. See Working with static files for more information.

Replace the contents of *Index.html* with the following markup:

```
<!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>
```

Run the app and navigate to `http://localhost:<port>/Index.html`. Alternatively, with *Index.html* opened, press `Ctrl+Shift+W`. Verify that the jumbotron styling is applied, the jQuery code responds when the button is clicked, and that the Bootstrap button changes state.


Bower Example
localhost:1234/Index.html
Using the jumbotron style
reset

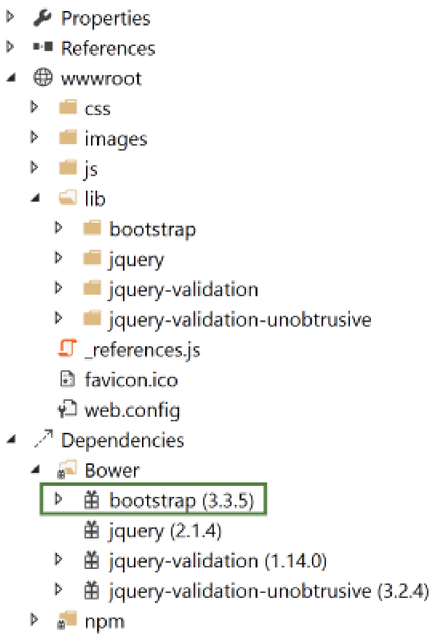
Building beautiful, responsive sites with Bootstrap

By Steve Smith

Bootstrap is currently the most popular web framework for developing responsive web applications. It offers a number of features and benefits that can improve your users' experience with your web site, whether you're a novice at front-end design and development or an expert. Bootstrap is deployed as a set of CSS and JavaScript files, and is designed to help your website or application scale efficiently from phones to tablets to desktops.

Getting started

There are several ways to get started with Bootstrap. If you're starting a new web application in Visual Studio, you can choose the default starter template for ASP.NET Core, in which case Bootstrap will come pre-installed:



Adding Bootstrap to an ASP.NET Core project is simply a matter of adding it to *bower.json* as a dependency:

```
{
  "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"
  }
}
```

This is the recommended way to add Bootstrap to an ASP.NET Core project.

You can also install bootstrap using one of several package managers, such as bower, npm, or NuGet. In each case, the process is essentially the same:

Bower

```
bower install bootstrap
```

npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

 Note

The recommended way to install client-side dependencies like Bootstrap in ASP.NET Core is via Bower (using *bower.json*, as shown above). The use of npm/NuGet are shown to demonstrate how easily Bootstrap can be added to other kinds of web applications, including earlier versions of ASP.NET.

If you're referencing your own local versions of Bootstrap, you'll need to reference them in any pages that will use it. In production you should reference bootstrap using a CDN. In the default ASP.NET site template, the *_Layout.cshtml* file does so like this:

```
<!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>
```

Note

If you're going to be using any of Bootstrap's jQuery plugins, you will also need to reference jQuery.

Basic templates and features

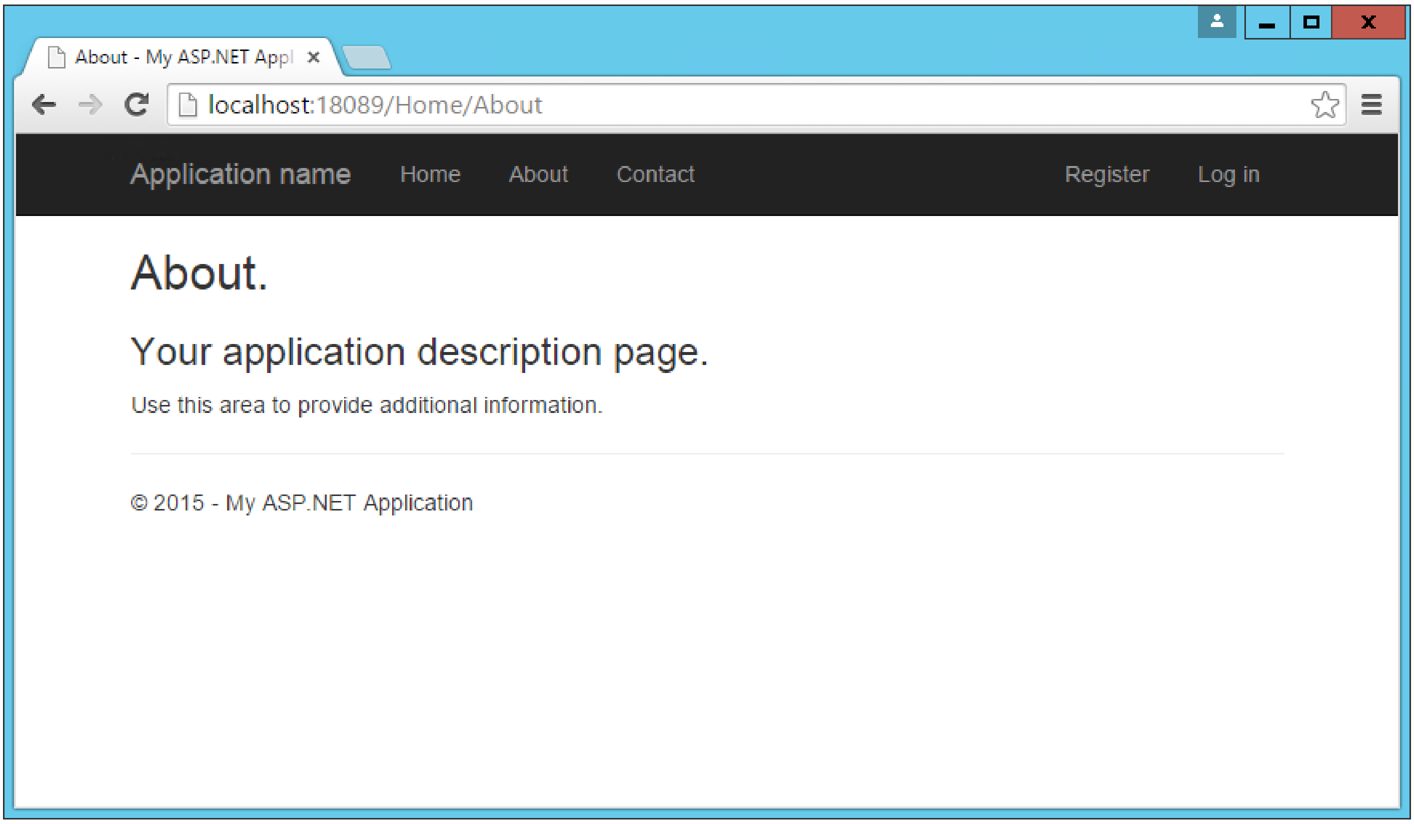
The most basic Bootstrap template looks very much like the *_Layout.cshtml* file shown above, and simply includes a basic menu for navigation and a place to render the rest of the page.

Basic navigation

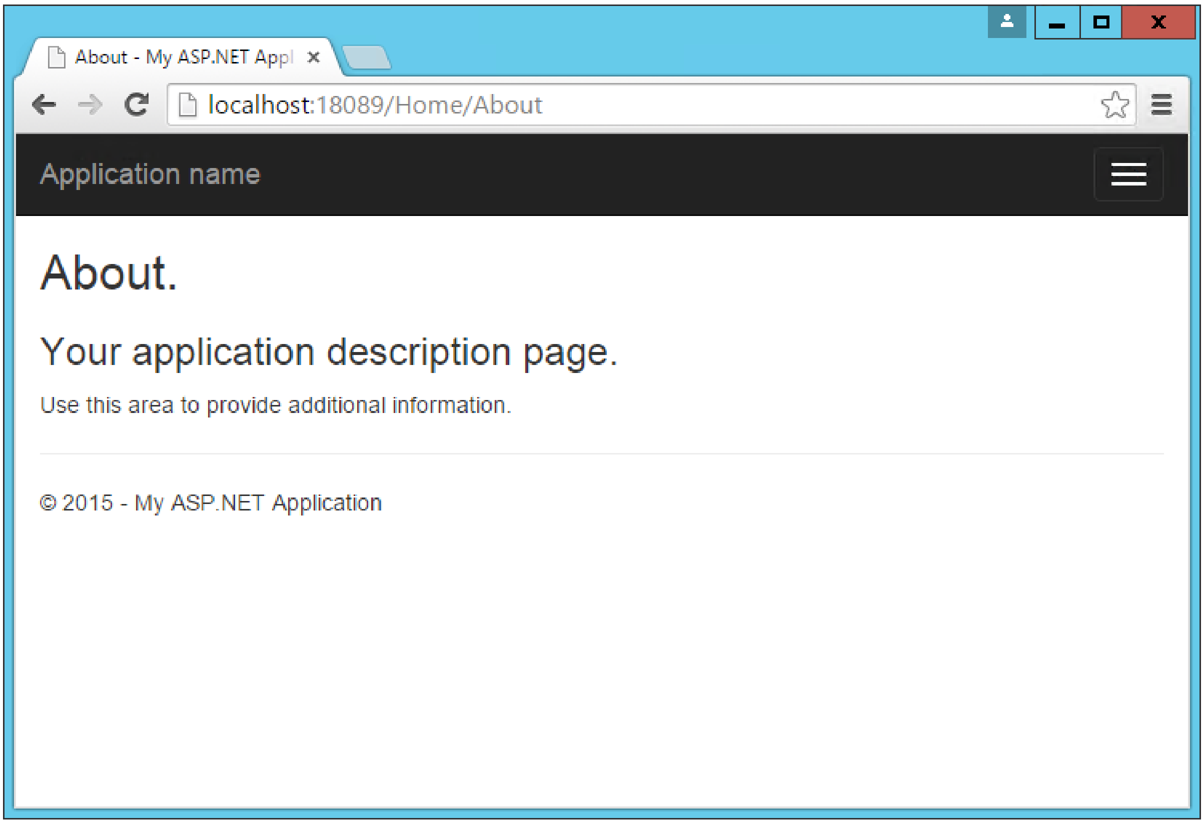
The default template uses a set of `<div>` elements to render a top navbar and the main body of the page. If you're using HTML5, you can replace the first `<div>` tag with a `<nav>` tag to get the same effect, but with more precise semantics. Within this first `<div>` you can see there are several others. First, a `<div>` with a class of "container", and then within that, two more `<div>` elements: "navbar-header" and "navbar-collapse". The navbar-header div includes a button that will appear when the screen is below a certain minimum width, showing 3 horizontal lines (a so-called "hamburger icon"). The icon is rendered using pure HTML and CSS; no image is required. This is the code that displays the icon, with each of the tags rendering one of the white bars:

```
<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>
```

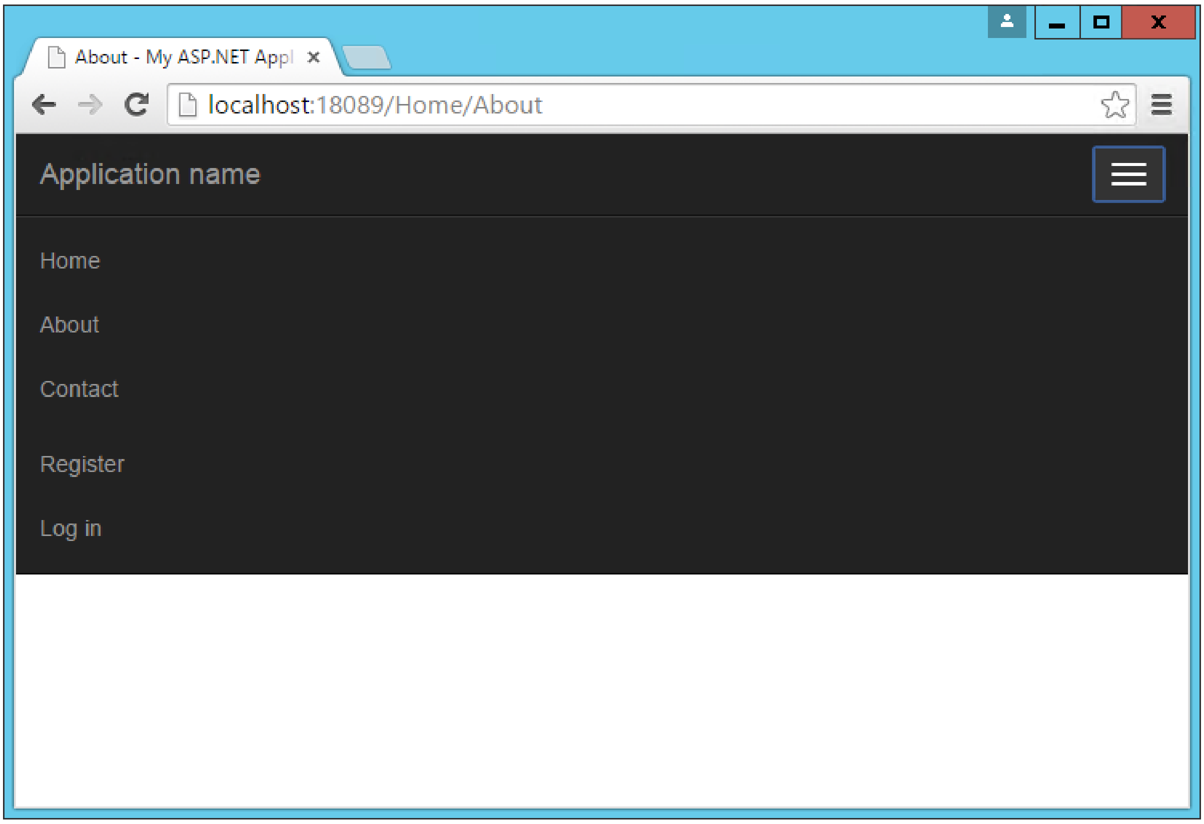
It also includes the application name, which appears in the top left. The main navigation menu is rendered by the `<ul>` element within the second div, and includes links to Home, About, and Contact. Additional links for Register and Login are added by the _LoginPartial line on line 29. Below the navigation, the main body of each page is rendered in another `<div>`, marked with the "container" and "body-content" classes. In the simple default _Layout file shown here, the contents of the page are rendered by the specific View associated with the page, and then a simple `<footer>` is added to the end of the `<div>` element. You can see how the built-in About page appears using this template:



The collapsed navbar, with "hamburger" button in the top right, appears when the window drops below a certain width:



Clicking the icon reveals the menu items in a vertical drawer that slides down from the top of the page:



Typography and links

Bootstrap sets up the site's basic typography, colors, and link formatting in its CSS file. This CSS file includes default styles for tables, buttons, form elements, images, and more (learn more). One particularly useful feature is the grid layout system, covered next.

Grids

One of the most popular features of Bootstrap is its grid layout system. Modern web applications should avoid using the `<table>` tag for layout, instead restricting the use of this element to actual tabular data. Instead, columns and rows can be laid out using a series of `<div>` elements and the appropriate CSS classes. There are several advantages to this approach, including the ability to adjust the layout of grids to display vertically on narrow screens, such as on phones.

Bootstrap's grid layout system is based on twelve columns. This number was chosen because it can be divided evenly into 1, 2, 3, or 4 columns, and column widths can vary to within 1/12th of the vertical width of the screen. To start using the grid layout system, you should begin with a container `<div>` and then add a row `<div>`, as shown here:

```
<div class="container">
    <div class="row">
        ...
    </div>
</div>
```

Next, add additional `<div>` elements for each column, and specify the number of columns that `<div>` should occupy (out of 12) as part of a CSS class starting with "col-md-". For instance, if you want to simply have two columns of equal size, you would use a class of "col-md-6" for each one. In this case "md" is short for "medium" and refers to standard-sized desktop computer display sizes. There are four different options you can choose from, and each will be used for higher widths unless overridden (so if you want the layout to be fixed regardless of screen width, you can just specify xs classes).

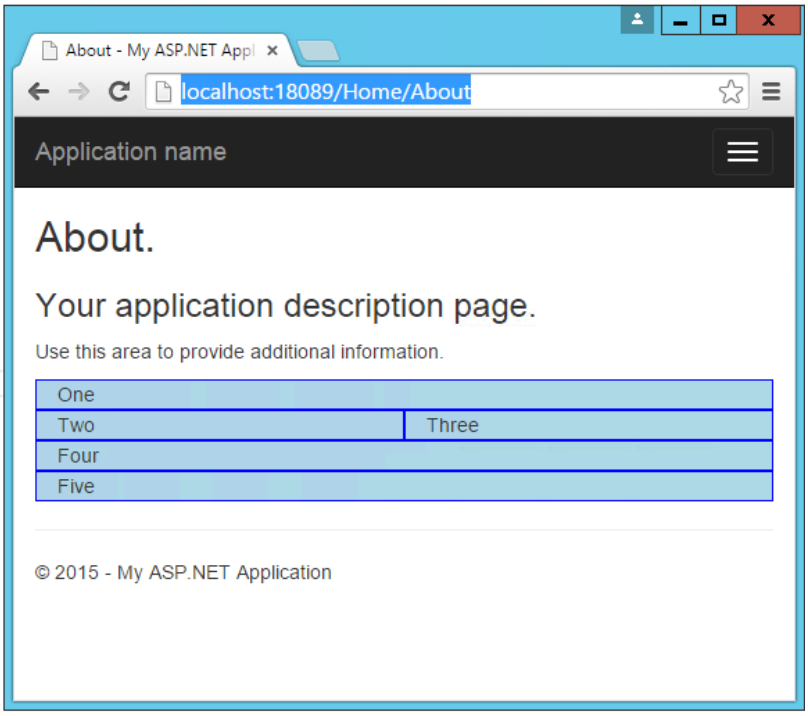
CSS CLASS PREFIX	DEVICE TIER	WIDTH
col-xs-	Phones	< 768px

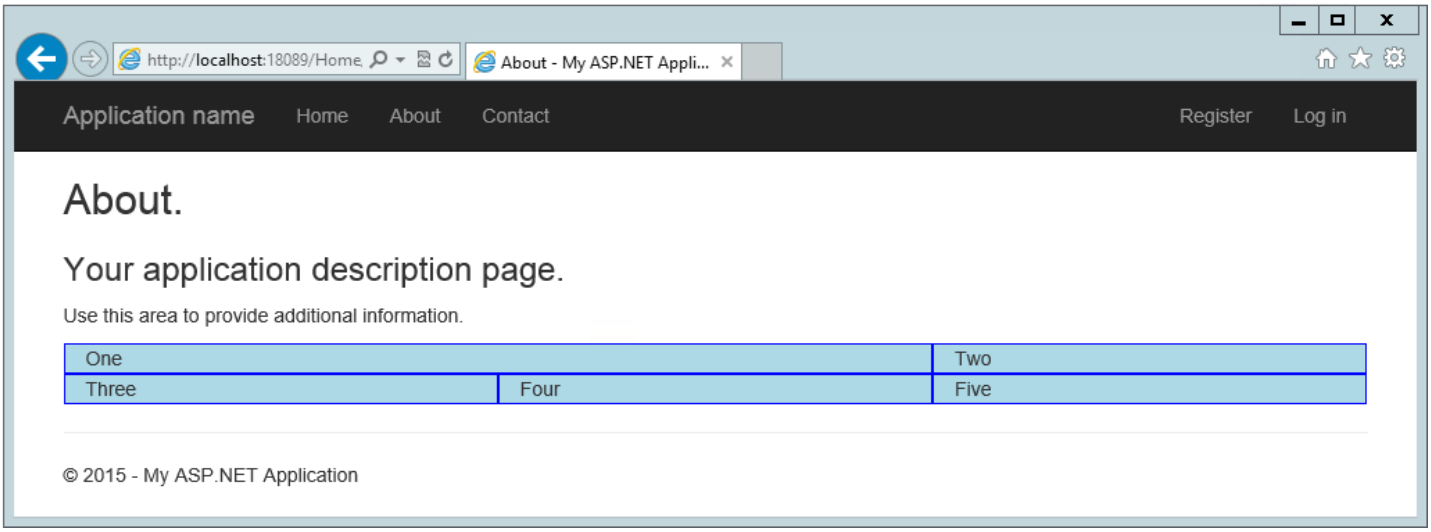
CSS CLASS PREFIX	DEVICE TIER	WIDTH
col-sm-	Tablets	>= 768px
col-md-	Desktops	>= 992px
col-lg-	Larger Desktop Displays	>= 1200px

When specifying two columns both with "col-md-6" the resulting layout will be two columns at desktop resolutions, but these two columns will stack vertically when rendered on smaller devices (or a narrower browser window on a desktop), allowing users to easily view content without the need to scroll horizontally.

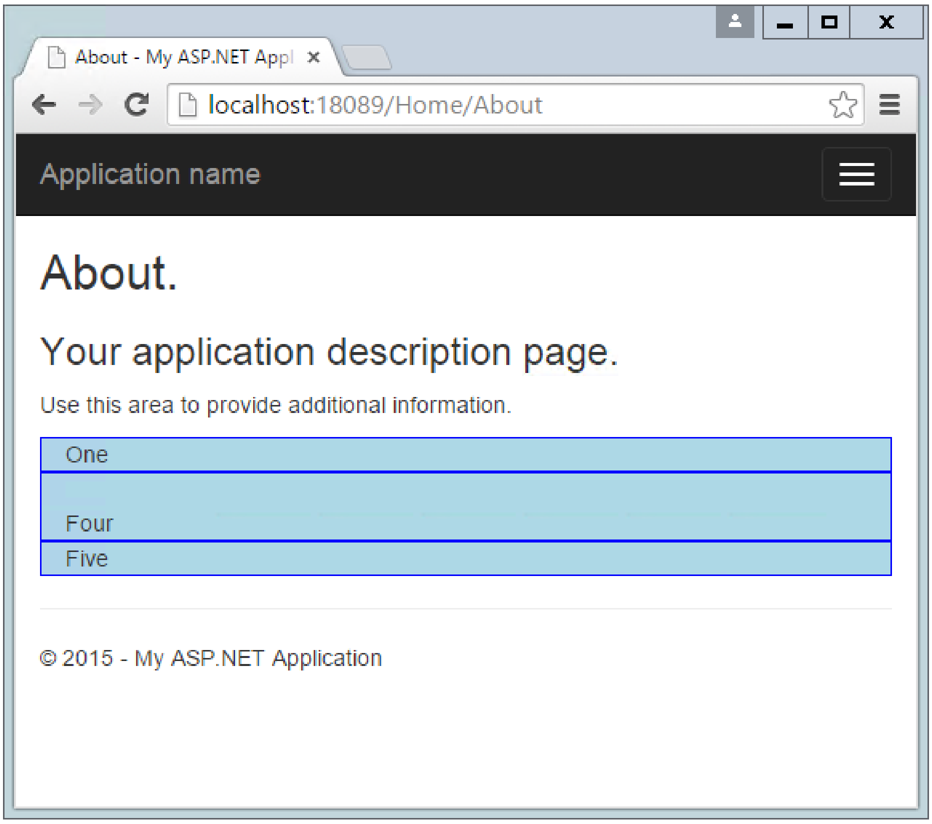
Bootstrap will always default to a single-column layout, so you only need to specify columns when you want more than one column. The only time you would want to explicitly specify that a `<div>` take up all 12 columns would be to override the behavior of a larger device tier. When specifying multiple device tier classes, you may need to reset the column rendering at certain points. Adding a clearfix div that is only visible within a certain viewport can achieve this, as shown here:

```html
<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>
```





In the above example, One and Two share a row in the "md" layout, while Two and Three share a row in the "xs" layout. Without the clearfix `<div>`, Two and Three are not shown correctly in the "xs" view (note that only One, Four, and Five are shown):

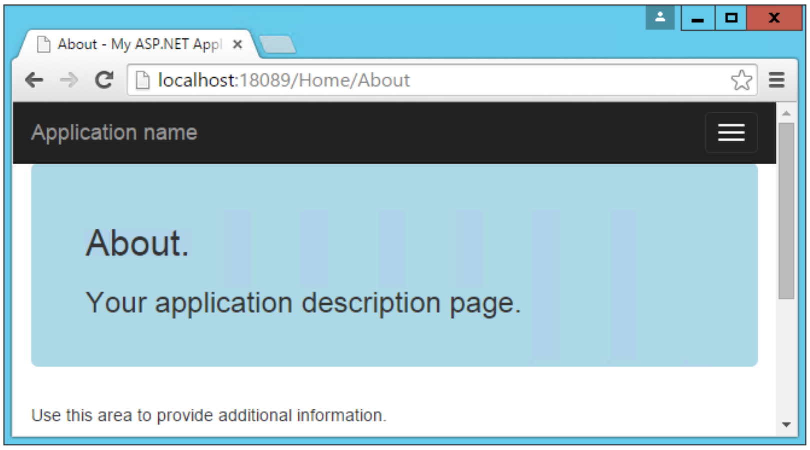


In this example, only a single row `<div>` was used, and Bootstrap still mostly did the right thing with regard to the layout and stacking of the columns. Typically, you should specify a row `<div>` for each horizontal row your layout requires, and of course you can nest Bootstrap grids within one another. When you do, each nested grid will occupy 100% of the width of the element in which it is placed, which can then be subdivided using column classes.

Jumbotron

If you've used the default ASP.NET MVC templates in Visual Studio 2012 or 2013, you've probably seen the Jumbotron in action. It refers to a large full-width section of a page that can be used to display a large background image, a call to action, a rotator, or similar elements. To add a jumbotron to a page, simply add a `<div>` and give it a class of "jumbotron", then place a container `<div>` inside and add your content. We can easily adjust the standard About page to use a jumbotron for the main headings it displays:

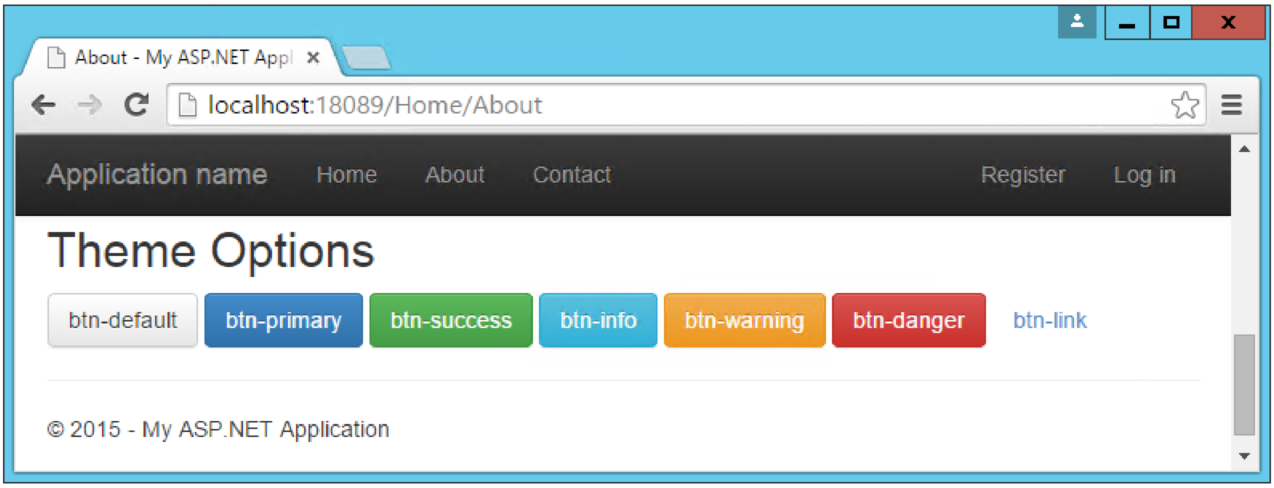
```
<style>
    .jumbotron {
        background-color: lightblue;
    }
</style>

<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>
```



Buttons

The default button classes and their colors are shown in the figure below.

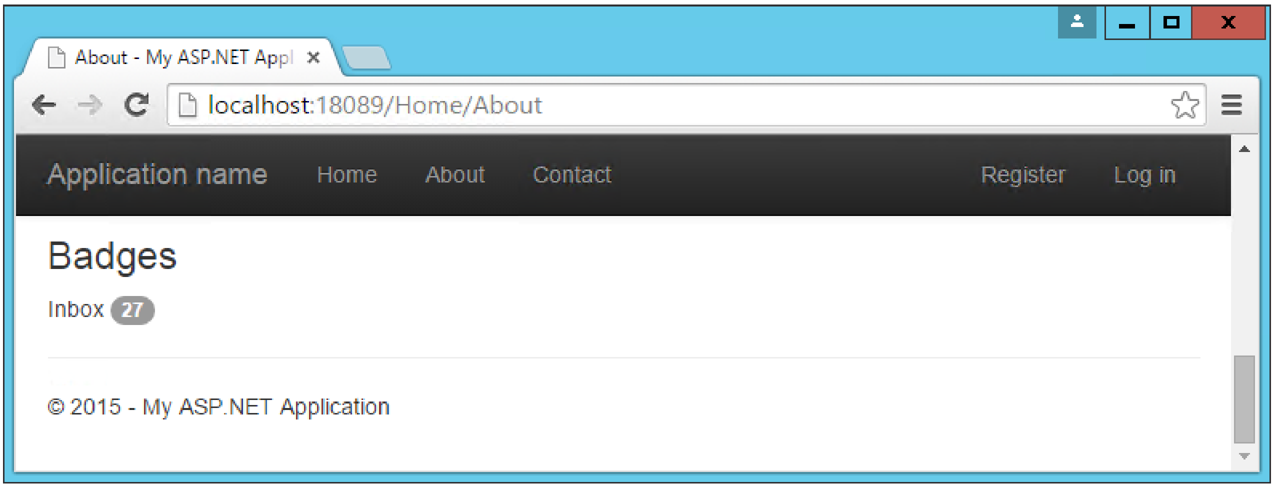
```
<h2>Theme Options</h2>
<p>
    <button type="button" class="btn btn-default">btn-default</button>
    <button type="button" class="btn btn-primary">btn-primary</button>
    <button type="button" class="btn btn-success">btn-success</button>
    <button type="button" class="btn btn-info">btn-info</button>
    <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>
```



Badges

Badges refer to small, usually numeric callouts next to a navigation item. They can indicate a number of messages or notifications waiting, or the presence of updates. Specifying such badges is as simple as adding a containing the text, with a class of "badge":

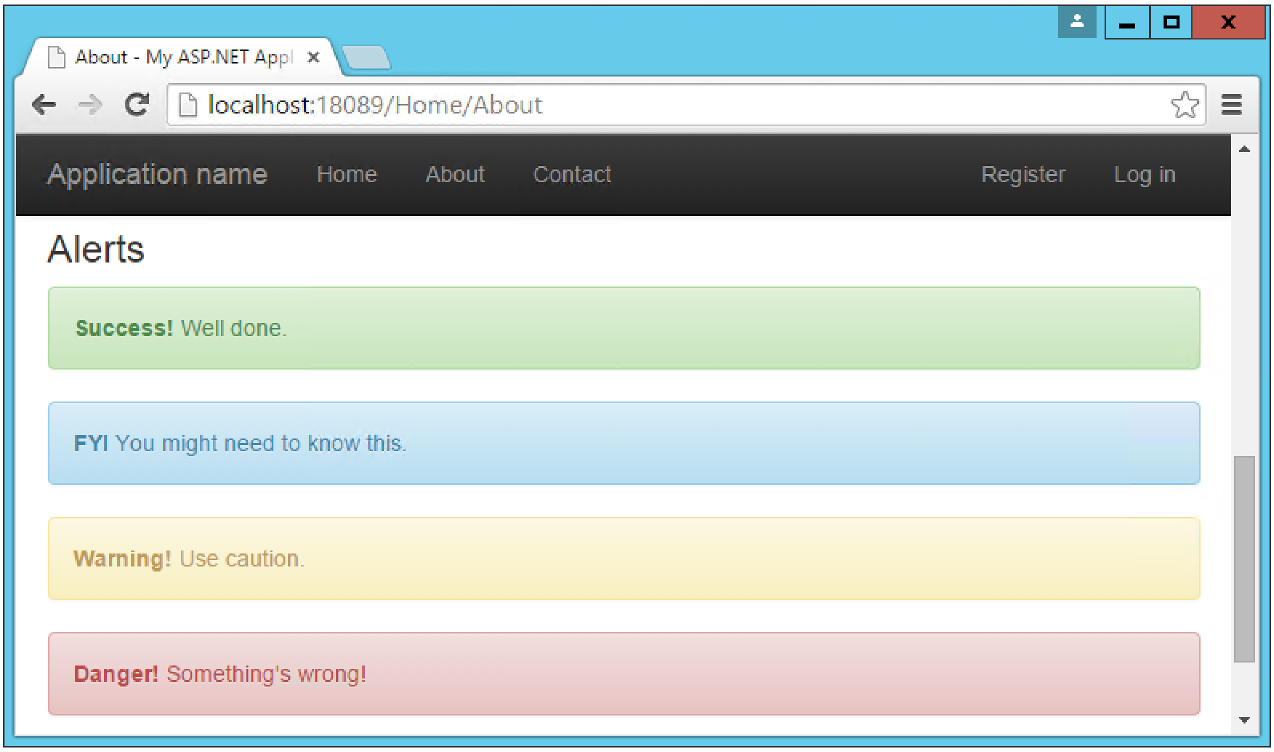
```
<h3>Badges</h3>
<p>
    Inbox <span class="badge">27</span>
</p>
```



Alerts

You may need to display some kind of notification, alert, or error message to your application's users. That's where the standard alert classes are useful. There are four different severity levels with associated color schemes:

```
<h3>Alerts</h3>
<div class="alert alert-success">
    <strong>Success!</strong> Well done.
</div>
<div class="alert alert-info">
    <strong>FYI</strong> You might need to know this.
</div>
<div class="alert alert-warning">
    <strong>Warning!</strong> Use caution.
</div>
<div class="alert alert-danger">
    <strong>Danger!</strong> Something's wrong!
</div>
```

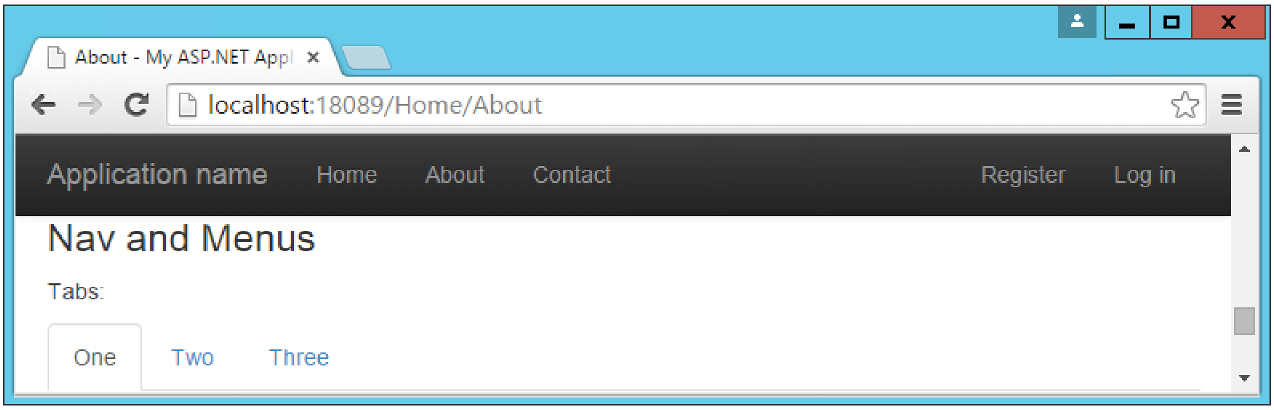


Navbars and menus

Our layout already includes a standard navbar, but the Bootstrap theme supports additional styling options. We can also easily opt to display the navbar vertically rather than horizontally if that's preferred, as well as adding sub-navigation items in flyout menus. Simple navigation menus, like tab strips, are built on top of

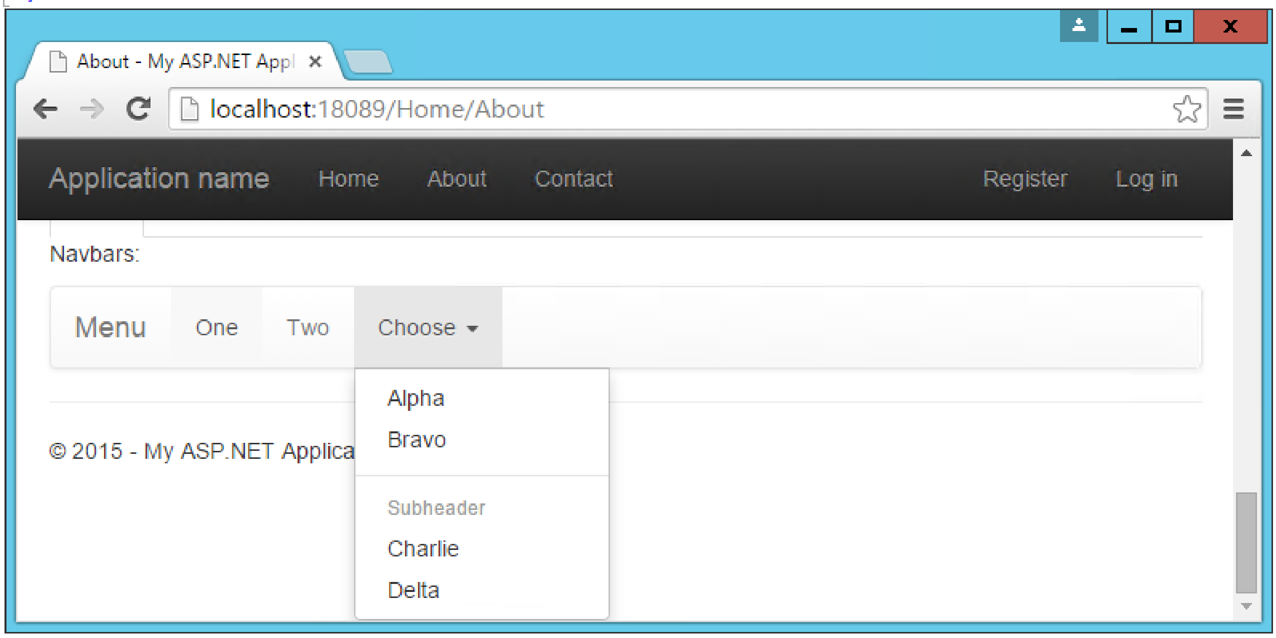
elements. These can be created very simply by just providing them with the CSS classes "nav" and "nav-tabs":

```
<h3>Nav and Menus</h3>
<p>Tabs:</p>
<ul class="nav nav-tabs">
    <li class="active"><a href="#">One</a></li>
    <li><a href="#">Two</a></li>
    <li><a href="#">Three</a></li>
</ul>
```



Navbars are built similarly, but are a bit more complex. They start with a `<nav>` or `<div>` with a class of "navbar", within which a container div holds the rest of the elements. Our page includes a navbar in its header already – the one shown below simply expands on this, adding support for a dropdown menu:

```
<p>Navbars:</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">Two</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown"
                       role="button" aria-expanded="false">Choose <span class="caret"></
                    <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>
```



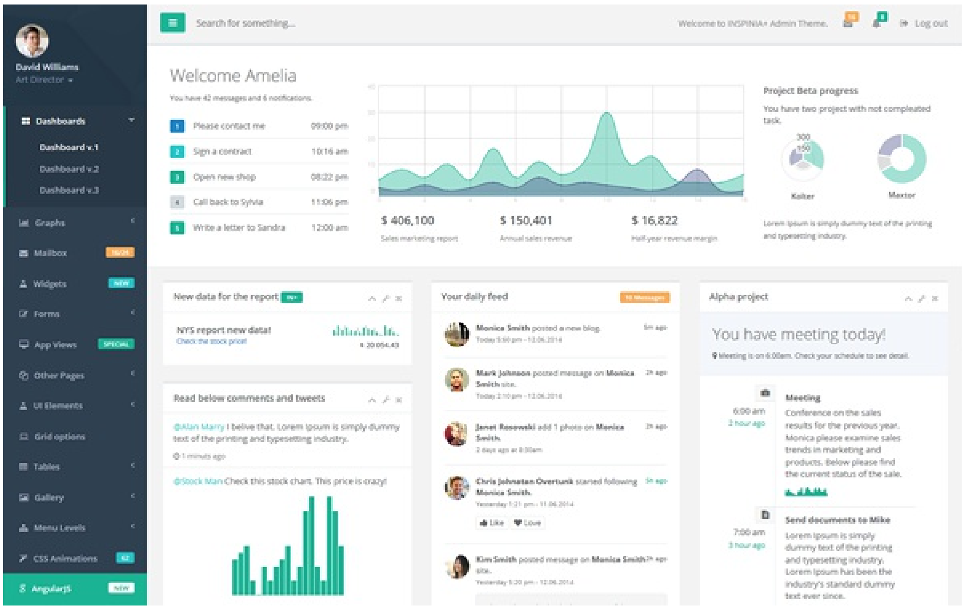
Additional elements

The default theme can also be used to present HTML tables in a nicely formatted style, including support for striped views. There are labels with styles that are similar to those of the buttons. You can create custom Dropdown menus that support additional styling options beyond the standard HTML `<select>` element, along with Navbars like the one our default starter site is already using. If you need a progress bar, there are several styles to choose from, as well as List Groups and panels that include a title and content. Explore additional options within the standard Bootstrap Theme here:

http://getbootstrap.com/examples/theme/

More themes

You can extend the standard Bootstrap theme by overriding some or all of its CSS, adjusting the colors and styles to suit your own application's needs. If you'd like to start from a ready-made theme, there are several theme galleries available online that specialize in Bootstrap themes, such as WrapBootstrap.com (which has a variety of commercial themes) and Bootswatch.com (which offers free themes). Some of the paid templates available provide a great deal of functionality on top of the basic Bootstrap theme, such as rich support for administrative menus, and dashboards with rich charts and gauges. An example of a popular paid template is Inspinia, currently for sale for $18, which includes an ASP.NET MVC5 template in addition to AngularJS and static HTML versions. A sample screenshot is shown below.



If you want to change your Bootstrap theme, put the *bootstrap.css* file for the theme you want in the **wwwroot/css** folder and change the references in *_Layout.cshtml* to point it. Change the links for all environments:

```
<environment names="Development">
    <link rel="stylesheet" href="~/css/bootstrap.css" />
```

```
<environment names="Staging,Production">
    <link rel="stylesheet" href="~/css/bootstrap.min.css" />
```

If you want to build your own dashboard, you can start from the free example available here: http://getbootstrap.com/examples/dashboard/.

Components

In addition to those elements already discussed, Bootstrap includes support for a variety of built-in UI components.

Glyphicons

Bootstrap includes icon sets from Glyphicons (http://glyphicons.com), with over 200 icons freely available for use within your Bootstrap-enabled web application. Here's just a small sample:

Input groups

Input groups allow bundling of additional text or buttons with an input element, providing the user with a more intuitive experience:

Breadcrumbs

Breadcrumbs are a common UI component used to show a user their recent history or depth within a site's navigation hierarchy. Add them easily by applying the "breadcrumb" class to any `<ol>` list element. Include built-in support for pagination by using the "pagination" class on a `<ul>` element within a `<nav>`. Add responsive embedded slideshows and video by using `<iframe>`, `<embed>`, `<video>`, or `<object>` elements, which Bootstrap will style automatically. Specify a particular aspect ratio by using specific classes like "embed-responsive-16by9".

JavaScript support

Bootstrap's JavaScript library includes API support for the included components, allowing you to control their behavior programmatically within your application. In addition, *bootstrap.js* includes over a dozen custom jQuery plugins, providing additional features like transitions, modal dialogs, scroll detection (updating styles based on where the user has scrolled in the document), collapse behavior, carousels, and affixing menus to the window so they do not scroll off the screen. There's not sufficient room to cover all of the JavaScript add-ons built into Bootstrap – to learn more please visit http://getbootstrap.com/javascript/.

Summary

Bootstrap provides a web framework that can be used to quickly and productively lay out and style a wide variety of websites and applications. Its basic typography and styles provide a pleasant look and feel that can easily be manipulated through custom theme support, which can be hand-crafted or purchased commercially. It supports a host of web components that in the past would have required expensive third-party controls to accomplish, while supporting modern and open web standards.

Knockout.js MVVM Framework in ASP.NET Core

By Steve Smith

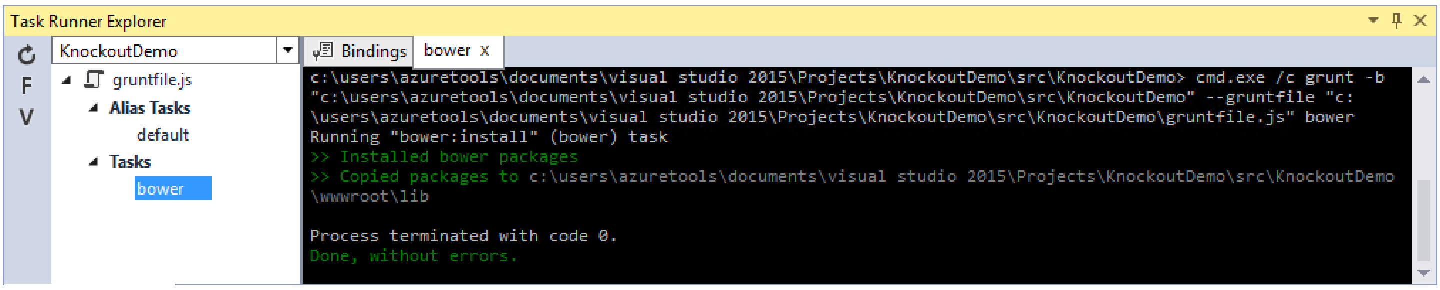
Knockout is a popular JavaScript library that simplifies the creation of complex data-based user interfaces. It can be used alone or with other libraries, such as jQuery. Its primary purpose is to bind UI elements to an underlying data model defined as a JavaScript object, such that when changes are made to the UI, the model is updated, and vice versa. Knockout facilitates the use of a Model-View-ViewModel (MVVM) pattern in a web application's client-side behavior. The two main concepts one must learn when working with Knockout's MVVM implementation are Observables and Bindings.

Getting started

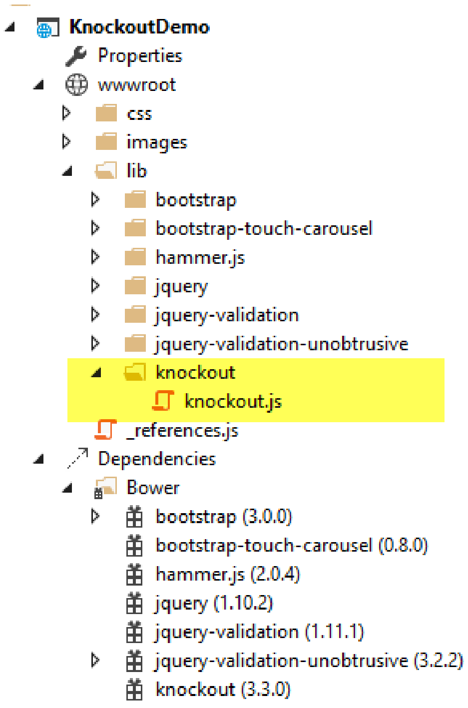
Knockout is deployed as a single JavaScript file, so installing and using it is very straightforward using bower. Assuming you already have bower and gulp configured, open *bower.json* in your ASP.NET Core project and add the knockout dependency as shown here:

```
{
  "name": "KnockoutDemo",
  "private": true,
  "dependencies": {
    "knockout" : "^3.3.0"
  },
  "exportsOverride": {
  }
}
```

With this in place, you can then manually run bower by opening the Task Runner Explorer (under View   Other Windows   Task Runner Explorer) and then under Tasks, right-click on bower and select Run. The result should appear similar to this:



Now if you look in your project's `wwwroot` folder, you should see knockout installed under the lib folder.



It's recommended that in your production environment you reference knockout via a Content Delivery Network, or CDN, as this increases the likelihood that your users will already have a cached copy of the file and thus will not need to download it at all. Knockout is available on several CDNs, including the Microsoft Ajax CDN, here:

http://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js

To include Knockout on a page that will use it, simply add a `<script>` element referencing the file from wherever you will be hosting it (with your application, or via a CDN):

```
<script type="text/javascript" src="knockout-3.3.0.js"></script>
```

Observables, ViewModels, and simple binding

You may already be familiar with using JavaScript to manipulate elements on a web page, either via direct access to the DOM or using a library like jQuery. Typically this kind of behavior is achieved by writing code to directly set element values in response to certain user actions. With Knockout, a declarative approach is taken instead, through which elements on the page are bound to properties on an object. Instead of writing code to manipulate DOM elements, user actions simply interact with the ViewModel object, and Knockout takes care of ensuring the page elements are synchronized.

As a simple example, consider the page list below. It includes a `<span>` element with a `data-bind` attribute indicating that the text content should be bound to authorName. Next, in a JavaScript block a variable viewModel is defined with a single property, `authorName`, set to some value. Finally, a call to `ko.applyBindings` is made, passing in this viewModel variable.

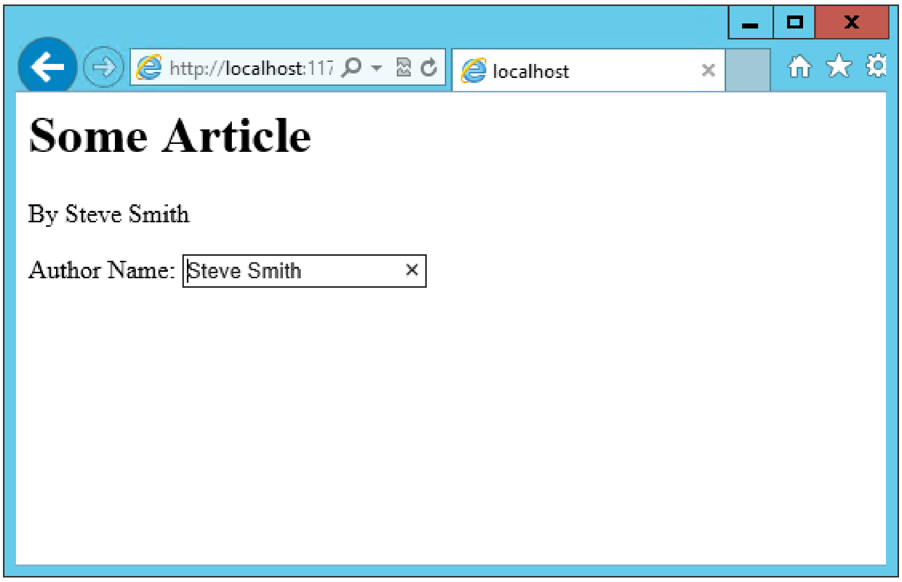
```
<html>
<head>
    <script type="text/javascript" src="lib/knockout/knockout.js"></script>
</head>
<body>
    <h1>Some Article</h1>
    <p>
        By <span data-bind="text: authorName"></span>
    </p>
    <script type="text/javascript">
      var viewModel = {
        authorName: 'Steve Smith'
      };
      ko.applyBindings(viewModel);
    </script>
</body>
</html>
```

When viewed in the browser, the content of the element is replaced with the value in the viewModel variable:

We now have simple one-way binding working. Notice that nowhere in the code did we write JavaScript to assign a value to the span's contents. If we want to manipulate the ViewModel, we can take this a step further and add an HTML input textbox, and bind to its value, like so:

```
<p>
    Author Name: <input type="text" data-bind="value: authorName" />
</p>
```

Reloading the page, we see that this value is indeed bound to the input box:



However, if we change the value in the textbox, the corresponding value in the `<span>` element doesn't change. Why not?

The issue is that nothing notified the `<span>` that it needed to be updated. Simply updating the ViewModel isn't by itself sufficient, unless the ViewModel's properties are wrapped in a special type. We need to use **observables** in the ViewModel for any properties that need to have changes automatically updated as they occur. By changing the ViewModel to use `ko.observable("value")` instead of just "value", the ViewModel will update any HTML elements that are bound to its value whenever a change occurs. Note that input boxes don't update their value until they lose focus, so you won't see changes to bound elements as you type.

Note

Adding support for live updating after each keypress is simply a matter of adding `valueUpdate: "afterkeydown"` to the `data-bind` attribute's contents. You can also get this behavior by using `data-bind="textInput: authorName"` to get instant updates of values.

Our viewModel, after updating it to use ko.observable:

Knockout supports a number of different kinds of bindings. So far we've seen how to bind to `text` and to `value`. You can also bind to any given attribute. For instance, to create a hyperlink with an anchor tag, the `src` attribute can be bound to the viewModel. Knockout also supports binding to functions. To demonstrate this, let's update the viewModel to include the author's twitter handle, and display the twitter handle as a link to the author's twitter page. We'll do this in three stages.

First, add the HTML to display the hyperlink, which we'll show in parentheses after the author's name:

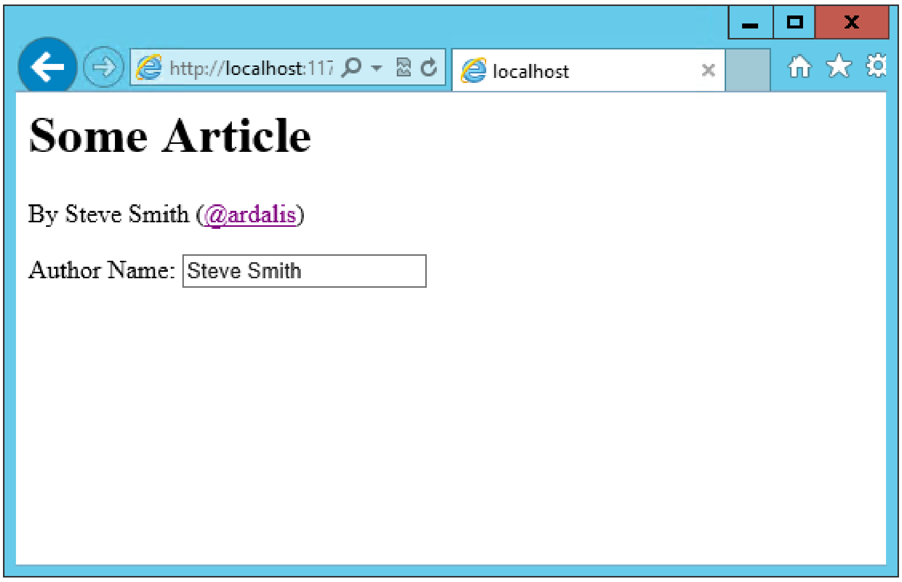
```
<h1>Some Article</h1>
<p>
    By <span data-bind="text: authorName"></span>
    (<a data-bind="attr: { href: twitterUrl}, text: twitterAlias" ></a>)
</p>
```

Next, update the viewModel to include the twitterUrl and twitterAlias properties:

Notice that at this point we haven't yet updated the twitterUrl to go to the correct URL for this twitter alias – it's just pointing at twitter.com. Also notice that we're using a new Knockout function, `computed`, for twitterUrl. This is an observable function that will notify any UI elements if it changes. However, for it to have access to other properties in the viewModel, we need to change how we are creating the viewModel, so that each property is its own statement.

The revised viewModel declaration is shown below. It is now declared as a function. Notice that each property is its own statement now, ending with a semicolon. Also notice that to access the twitterAlias property value, we need to execute it, so its reference includes ().

The result works as expected in the browser:



Knockout also supports binding to certain UI element events, such as the click event. This allows you to easily and declaratively bind UI elements to functions within the application's viewModel. As a simple example, we can add a button that, when clicked, modifies the author's twitterAlias to be all caps.

First, we add the button, binding to the button's click event, and referencing the function name we're going to add to the viewModel:

```
<p>
    <button data-bind="click: capitalizeTwitterAlias">Capitalize</button>
</p>
```

Then, add the function to the viewModel, and wire it up to modify the viewModel's state. Notice that to set a new value to the twitterAlias property, we call it as a method and pass in the new value.

Running the code and clicking the button modifies the displayed link as expected:

Control flow

Knockout includes bindings that can perform conditional and looping operations. Looping operations are especially useful for binding lists of data to UI lists, menus, and grids or tables. The foreach binding will iterate over an array. When used with an observable array, it will automatically update the UI elements when items are added or removed from the array, without re-creating every element in the UI tree. The following example uses a new viewModel which includes an observable array of game results. It is bound to a simple table with two columns using a `foreach` binding on the `<tbody>` element. Each `<tr>` element within `<tbody>` will be bound to an element of the gameResults collection.

```
<h1>Record</h1>
<table>
    <thead>
        <tr>
            <th>Opponent</th>
            <th>Result</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: gameResults">
        <tr>
            <td data-bind="text:opponent"></td>
            <td data-bind="text:result"></td>
        </tr>
    </tbody>
</table>
<script type="text/javascript">
  function GameResult(opponent, result) {
    var self = this;
    self.opponent = opponent;
    self.result = ko.observable(result);
  }

  function ViewModel() {
    var self = this;

    self.resultChoices = ["Win", "Loss", "Tie"];

    self.gameResults = ko.observableArray([
      new GameResult("Brendan", self.resultChoices[0]),
      new GameResult("Brendan", self.resultChoices[0]),
      new GameResult("Michelle", self.resultChoices[1])
    ]);
  };
  ko.applyBindings(new ViewModel);
</script>
```

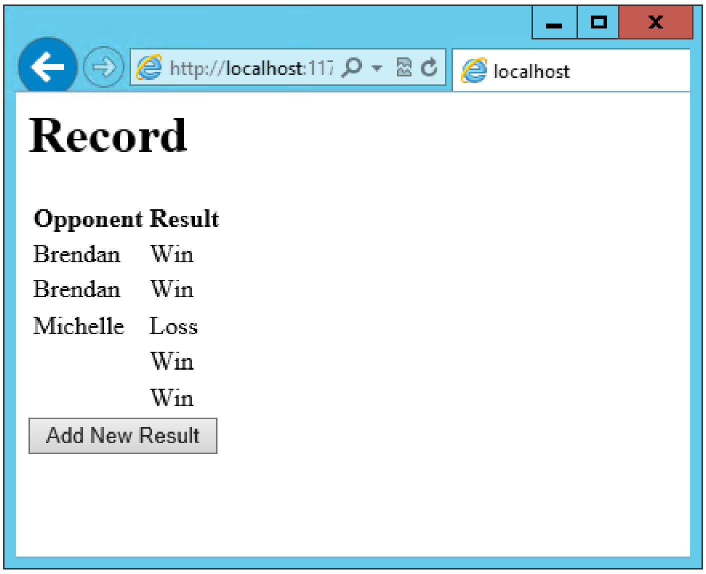
Notice that this time we're using ViewModel with a capital "V" because we expect to construct it using "new" (in the applyBindings call). When executed, the page results in the following output:

To demonstrate that the observable collection is working, let's add a bit more functionality. We can include the ability to record the results of another game to the ViewModel, and then add a button and some UI to work with this new function. First, let's create the addResult method:

Bind this method to a button using the `click` binding:

```
<button data-bind="click: addResult">Add New Result</button>
```

Open the page in the browser and click the button a couple of times, resulting in a new table row with each click:

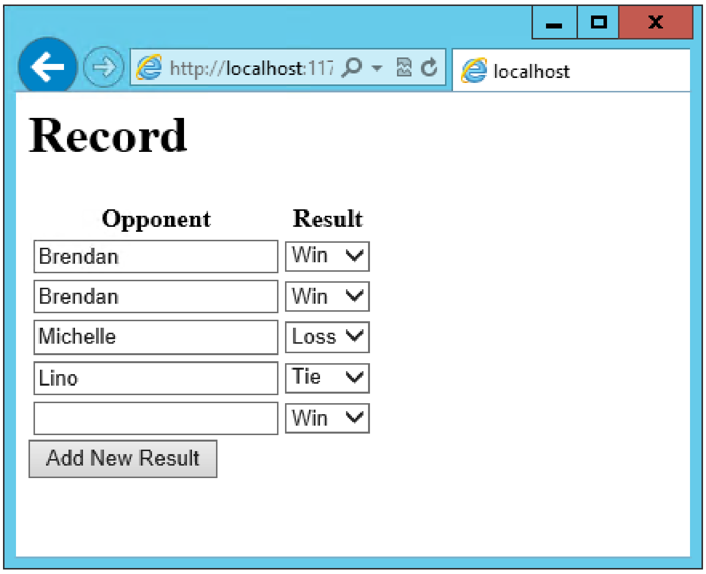


There are a few ways to support adding new records in the UI, typically either inline or in a separate form. We can easily modify the table to use textboxes and dropdownlists so that the whole thing is editable. Just change the `<tr>` element as shown:

```
<tbody data-bind="foreach: gameResults">
    <tr>
      <td><input data-bind="value:opponent" /></td>
      <td><select data-bind="options: $root.resultChoices, value:result, optionsText: $data"></select></td>
    </tr>
</tbody>
```

Note that `$root` refers to the root ViewModel, which is where the possible choices are exposed. `$data` refers to whatever the current model is within a given context - in this case it refers to an individual element of the resultChoices array, each of which is a simple string.

With this change, the entire grid becomes editable:



If we weren't using Knockout, we could achieve all of this using jQuery, but most likely it would not be nearly as efficient. Knockout tracks which bound data items in the ViewModel correspond to which UI elements, and only updates those elements that need to be added, removed, or updated. It would take significant effort to achieve this ourselves using jQuery or direct DOM manipulation, and even then if we then wanted to display aggregate results (such as a win-loss record) based on the table's data, we would need to once more loop through it and parse the HTML elements. With Knockout, displaying the win-loss record is trivial. We can perform the calculations within the ViewModel itself, and then display it with a simple text binding and a `<span>`.

To build the win-loss record string, we can use a computed observable. Note that references to observable properties within the ViewModel must be function calls, otherwise they will not retrieve the value of the observable (i.e. `gameResults()` not `gameResults` in the code shown):

Bind this function to a span within the `<h1>` element at the top of the page:

```
<h1>Record <span data-bind="text: displayRecord"></span></h1>
```

The result:

Adding rows or modifying the selected element in any row's Result column will update the record shown at the top of the window.

In addition to binding to values, you can also use almost any legal JavaScript expression within a binding. For example, if a UI element should only appear under certain conditions, such as when a value exceeds a certain threshold, you can specify this logically within the binding expression:

```
<div data-bind="visible: customerValue > 100"></div>
```

This `<div>` will only be visible when the customerValue is over 100.

Templates

Knockout has support for templates, so that you can easily separate your UI from your behavior, or incrementally load UI elements into a large application on demand. We can update our previous example to make each row its own template by simply pulling the HTML out into a template and specifying the template by name in the data-bind call on `<tbody>`.

```
<tbody data-bind="template: { name: 'rowTemplate', foreach: gameResults }">
</tbody>
<script type="text/html" id="rowTemplate">
  <tr>
      <td><input data-bind="value:opponent" /></td>
      <td><select data-bind="options: $root.resultChoices, value:result, optionsText: $data"></select></td>
  </tr>
</script>
```

Knockout also supports other templating engines, such as the jQuery.tmpl library and Underscore.js's templating engine.

Components

Components allow you to organize and reuse UI code, usually along with the ViewModel data on which the UI code depends. To create a component, you simply need to specify its template and its viewModel, and give it a name. This is done by calling `ko.components.register()`. In addition to defining the templates and viewmodel inline, they can be loaded from external files using a library like *require.js*, resulting in very clean and efficient code.

Communicating with APIs

Knockout can work with any data in JSON format. A common way to retrieve and save data using Knockout is with jQuery, which supports the `$.getJSON()` function to retrieve data, and the `$.post()` method to send data from the browser to an API endpoint. Of course, if you prefer a different way to send and receive JSON data, Knockout will work with it as well.

Summary

Knockout provides a simple, elegant way to bind UI elements to the current state of the client application, defined in a ViewModel. Knockout's binding syntax uses the data-bind attribute, applied to HTML elements that are to be processed. Knockout is able to efficiently render and update large data sets by tracking UI elements and only processing changes to affected elements. Large applications can break up UI logic using templates and components, which can be loaded on demand from external files. Currently version 3, Knockout is a stable JavaScript library that can improve web applications that require rich client interactivity.

Using AngularJS for Single Page Applications (SPAs) with ASP.NET Core

By Venkata Koppaka and Scott Addie

In this article, you will learn how to build a SPA-style ASP.NET application using AngularJS.

View or download sample code

What is AngularJS?

AngularJS is a modern JavaScript framework from Google commonly used to work with Single Page Applications (SPAs). AngularJS is open sourced under MIT license, and the development progress of AngularJS can be followed on its GitHub repository. The library is called Angular because HTML uses angular-shaped brackets.

AngularJS is not a DOM manipulation library like jQuery, but it uses a subset of jQuery called jQLite. AngularJS is primarily based on declarative HTML attributes that you can add to your HTML tags. You can try AngularJS in your browser using the Code School website or W3Schools website.

This article focuses on AngularJS with some notes on where Angular is heading.

Getting started

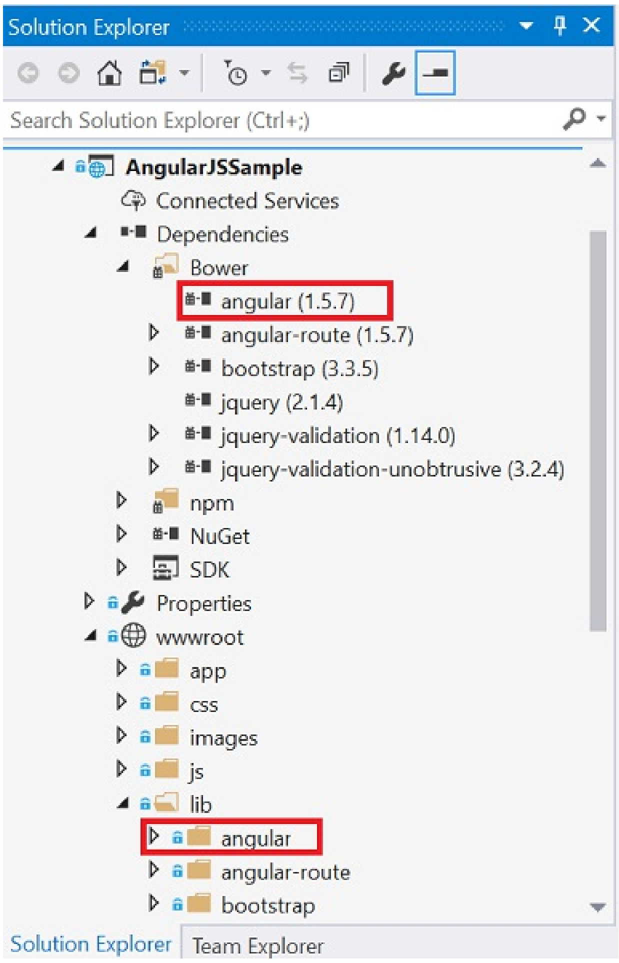
To start using AngularJS in your ASP.NET application, you must either install it as part of your project, or reference it from a content delivery network (CDN).

Installation

There are several ways to add AngularJS to your application. If you're starting a new ASP.NET Core web application in Visual Studio, you can add AngularJS using the built-in Bower support. Open *bower.json*, and add an entry to the `dependencies` property:

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.5",
    "jquery": "2.1.4",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.4",
    "angular": "1.5.7",
    "angular-route": "1.5.7"
  }
}
```

Upon saving the *bower.json* file, Angular will be installed in your project's *wwwroot/lib* folder. Additionally, it will be listed within the `Dependencies/Bower` folder. See the screenshot below.



Next, add a `<script>` reference to the bottom of the `<body>` section of your HTML page or *_Layout.cshtml* file, as shown here:

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/lib/angular/angular.js"></script>
</environment>
```

It's recommended that production applications utilize CDNs for common libraries like AngularJS. You can reference AngularJS from one of several CDNs, such as this one:

```
<environment names="Staging,Production">
    <script src="//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>
    <script src="//ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/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="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.7/angular.min.js"
            asp-fallback-src="~/lib/angular/angular.min.js"
            asp-fallback-test="window.angular">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>
```

Once you have a reference to the *angular.js* script file, you're ready to begin using AngularJS in your web pages.

Key components

AngularJS includes a number of major components, such as *directives*, *templates*, *repeaters*, *modules*, *controllers*, *components*, *component router* and more. Let's examine how these components work together to add behavior to your web pages.

Directives

AngularJS uses directives to extend HTML with custom attributes and elements. AngularJS directives are defined via `data-ng-*` or `ng-*` prefixes (`ng` is short for angular). There are two types of AngularJS directives:

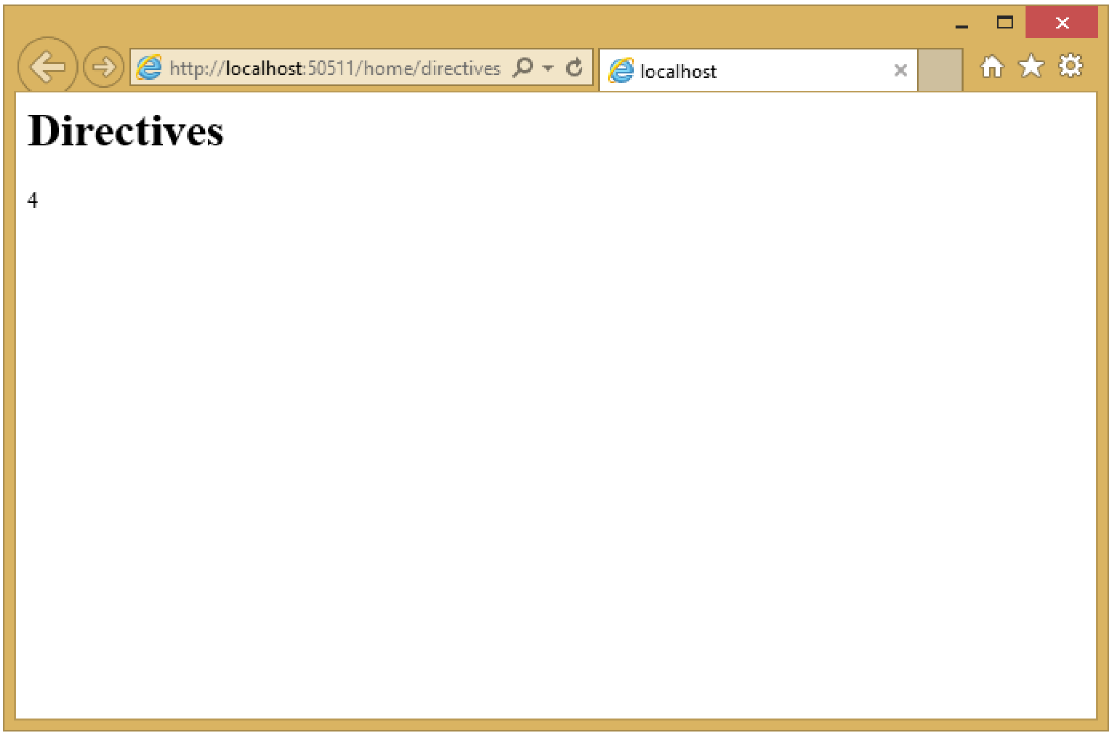
Primitive Directives: These are predefined by the Angular team and are part of the AngularJS framework.

Custom Directives: These are custom directives that you can define.

One of the primitive directives used in all AngularJS applications is the `ng-app` directive, which bootstraps the AngularJS application. This directive can be applied to the `<body>` tag or to a child element of the body. Let's see an example in action. Assuming you're in an ASP.NET project, you can either add an HTML file to the `wwwroot` folder, or add a new controller action and an associated view. In this case, I've added a new `Directives` action method to `HomeController.cs`. The associated view is shown here:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Directives</h1>
    {{2+2}}
    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

To keep these samples independent of one another, I'm not using the shared layout file. You can see that we decorated the body tag with the `ng-app` directive to indicate this page is an AngularJS application. The `{{2+2}}` is an Angular data binding expression that you will learn more about in a moment. Here is the result if you run this application:



Other primitive directives in AngularJS include:

`ng-controller` Determines which JavaScript controller is bound to which view.

`ng-model` Determines the model to which the values of an HTML element's properties are bound.

`ng-init` Used to initialize the application data in the form of an expression for the current scope.

`ng-if` Removes or recreates the given HTML element in the DOM based on the truthiness of the expression provided.

`ng-repeat` Repeats a given block of HTML over a set of data.

`ng-show` Shows or hides the given HTML element based on the expression provided.

For a full list of all primitive directives supported in AngularJS, please refer to the directive documentation section on the AngularJS documentation website.

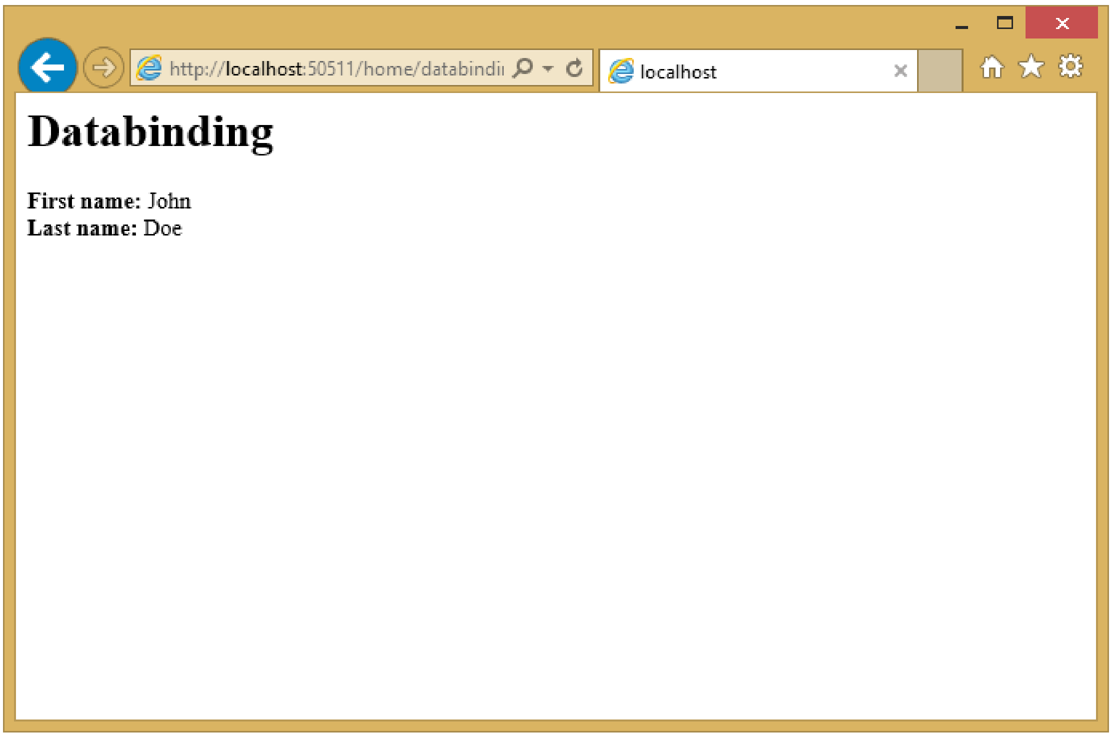
Data binding

AngularJS provides data binding support out-of-the-box using either the `ng-bind` directive or a data binding expression syntax such as `{{expression}}`. AngularJS supports two-way data binding where data from a model is kept in synchronization with a view template at all times. Any changes to the view are automatically reflected in the model. Likewise, any changes in the model are reflected in the view.

Create either an HTML file or a controller action with an accompanying view named `Databinding`. Include the following in the view:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Databinding</h1>

    <div ng-init="firstName='John'; lastName='Doe';">
        <strong>First name:</strong> {{firstName}} <br />
        <strong>Last name:</strong> <span ng-bind="lastName" />
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

Notice that you can display model values using either directives or data binding (`ng-bind`). The resulting page should look like this:



Templates

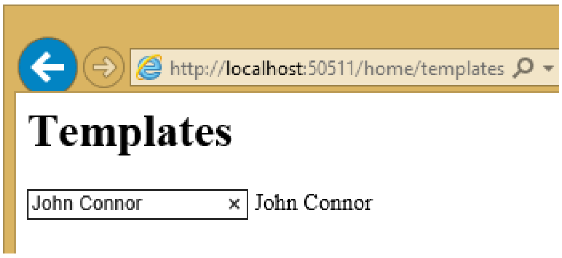
Templates in AngularJS are just plain HTML pages decorated with AngularJS directives and artifacts. A template in AngularJS is a mixture of directives, expressions, filters, and controls that combine with HTML to form the view.

Add another view to demonstrate templates, and add the following to it:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Templates</h1>

    <div ng-init="personName='John Doe'">
        <input ng-model="personName" /> {{personName}}
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

The template has AngularJS directives like `ng-app`, `ng-init`, `ng-model` and data binding expression syntax to bind the `personName` property. Running in the browser, the view looks like the screenshot below:

If you change the name by typing in the input field, you will see the text next to the input field dynamically update, showing Angular two-way data binding in action.



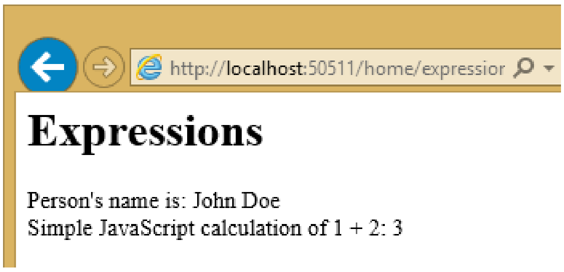
Expressions

Expressions in AngularJS are JavaScript-like code snippets that are written inside the `{{ expression }}` syntax. The data from these expressions is bound to HTML the same way as `ng-bind` directives. The main difference between AngularJS expressions and regular JavaScript expressions is that AngularJS expressions are evaluated against the `$scope` object in AngularJS.

The AngularJS expressions in the sample below bind `personName` and a simple JavaScript calculated expression:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Expressions</h1>

    <div ng-init="personName='John Doe'">
        Person's name is: {{personName}} <br />
        Simple JavaScript calculation of 1 + 2: {{1+2}}
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

The example running in the browser displays the `personName` data and the results of the calculation:

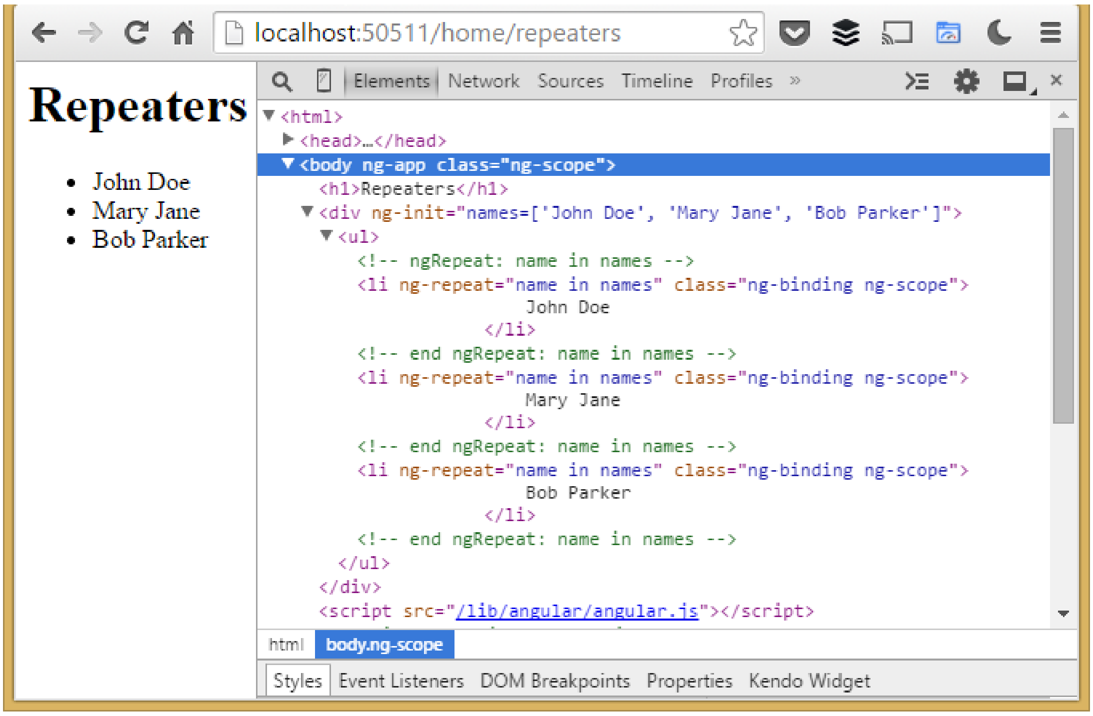


Repeaters

Repeating in AngularJS is done via a primitive directive called `ng-repeat`. The `ng-repeat` directive repeats a given HTML element in a view over the length of a repeating data array. Repeaters in AngularJS can repeat over an array of strings or objects. Here is a sample usage of repeating over an array of strings:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Repeaters</h1>

    <div ng-init="names=['John Doe', 'Mary Jane', 'Bob Parker']">
        <ul>
            <li ng-repeat="name in names">
                {{name}}
            </li>
        </ul>
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

The repeat directive outputs a series of list items in an unordered list, as you can see in the developer tools shown in this screenshot:



Here is an example that repeats over an array of objects. The `ng-init` directive establishes a `names` array, where each element is an object containing first and last names. The `ng-repeat` assignment, `name in names`, outputs a list item for every array element.

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Repeaters2</h1>

    <div ng-init="names=[
  {firstName:'John', lastName:'Doe'},
  {firstName:'Mary', lastName:'Jane'},
  {firstName:'Bob', lastName:'Parker'}]">
        <ul>
            <li ng-repeat="name in names">
                {{name.firstName + ' ' + name.lastName}}
            </li>
        </ul>
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

The output in this case is the same as in the previous example.

Angular provides some additional directives that can help provide behavior based on where the loop is in its execution.

`$index`

Use `$index` in the `ng-repeat` loop to determine which index position your loop currently is on.

`$even` and `$odd`

Use `$even` in the `ng-repeat` loop to determine whether the current index in your loop is an even indexed row. Similarly, use `$odd` to determine if the current index is an odd indexed row.

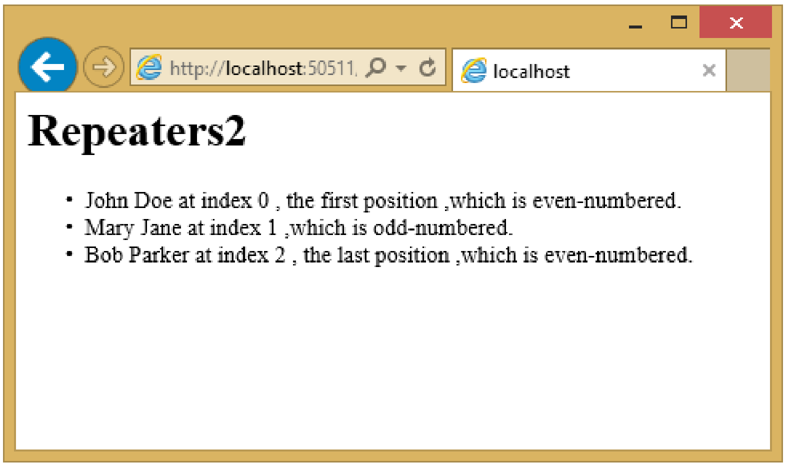
`$first` and `$last`

Use `$first` in the `ng-repeat` loop to determine whether the current index in your loop is the first row. Similarly, use `$last` to determine if the current index is the last row.

Below is a sample that shows `$index`, `$even`, `$odd`, `$first`, and `$last` in action:

```
@{
    Layout = "";
}
<html>
<body ng-app>
    <h1>Repeaters2</h1>

    <div ng-init="names=[
  {firstName:'John', lastName:'Doe'},
  {firstName:'Mary', lastName:'Jane'},
  {firstName:'Bob', lastName:'Parker'}]">
        <ul>
            <li ng-repeat="name in names">
                {{name.firstName + ' ' + name.lastName}} at index {{$index}}
                <span ng-show="{{$first}}">, the first position</span>
                <span ng-show="{{$last}}">, the last position</span>
                <span ng-show="{{$odd}}">,which is odd-numbered.</span>
                <span ng-show="{{$even}}">,which is even-numbered.</span>
            </li>
        </ul>
    </div>

    <script src="~/lib/angular/angular.js"></script>
</body>
</html>
```

Here is the resulting output:



$scope

`$scope` is a JavaScript object that acts as glue between the view (template) and the controller (explained below). A view template in AngularJS only knows about the values attached to the `$scope` object in the controller.

Note

In the MVVM world, the `$scope` object in AngularJS is often defined as the ViewModel. The AngularJS team refers to the `$scope` object as the Data-Model. Learn more about Scopes in AngularJS.

Below is a simple example showing how to set properties on `$scope` within a separate JavaScript file, *scope.js*:

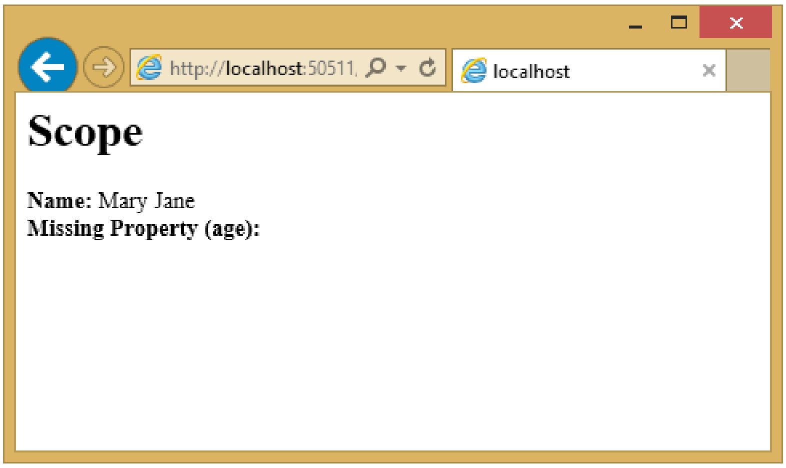
```
var personApp = angular.module('personApp', []);
personApp.controller('personController', ['$scope', function ($scope) {
    $scope.name = 'Mary Jane';
}]);
```

Observe the `$scope` parameter passed to the controller on line 2. This object is what the view knows about. On line 3, we are setting a property called "name" to "Mary Jane".

What happens when a particular property is not found by the view? The view defined below refers to "name" and "age" properties:

```
@{
    Layout = "";
}
<html>
<body ng-app="personApp">
    <h1>Scope</h1>

    <div ng-controller="personController">
        <strong>Name:</strong> {{name}} <br />
        <strong>Missing Property (age):</strong> {{age}}
    </div>

    <script src="~/lib/angular/angular.js"></script>
    <script src="~/app/scope.js"></script>
</body>
</html>
```

Notice on line 9 that we are asking Angular to show the "name" property using expression syntax. Line 10 then refers to "age", a property that does not exist. The running example shows the name set to "Mary Jane" and nothing for age. Missing properties are ignored.



Modules

A module in AngularJS is a collection of controllers, services, directives, etc. The `angular.module()` function call is used to create, register, and retrieve modules in AngularJS. All modules, including those shipped by the AngularJS team and third party libraries, should be registered using the `angular.module()` function.

Below is a snippet of code that shows how to create a new module in AngularJS. The first parameter is the name of the module. The second parameter defines dependencies on other modules. Later in this article, we will be showing how to pass these dependencies to an `angular.module()` method call.

Use the `ng-app` directive to represent an AngularJS module on the page. To use a module, assign the name of the module, `personApp` in this example, to the `ng-app` directive in our template.

```
<body ng-app="personApp">
```

Controllers

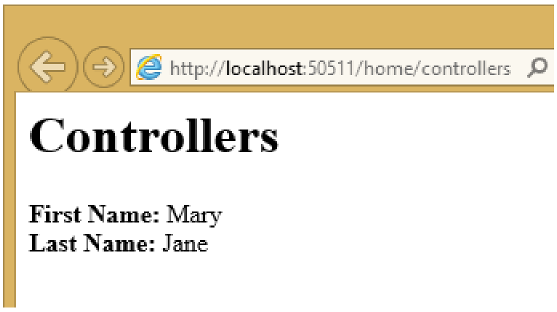
Controllers in AngularJS are the first point of entry for your code. The `<module name>.controller()` function call is used to create and register controllers in AngularJS. The `ng-controller` directive is used to represent an AngularJS controller on the HTML page. The role of the controller in Angular is to set state and behavior of the data model (`$scope`). Controllers should not be used to manipulate the DOM directly.

Below is a snippet of code that registers a new controller. The `personApp` variable in the snippet references an Angular module, which is defined on line 2.

The view using the `ng-controller` directive assigns the controller name:

```
@{
    Layout = "";
}
<html>
<body ng-app="personApp">
    <h1>Controllers</h1>

    <div ng-controller="personController">
        <strong>First Name:</strong> {{firstName}} <br />
        <strong>Last Name:</strong> {{lastName}}
    </div>

    <script src="~/lib/angular/angular.js"></script>
    <script src="~/app/controllers.js"></script>
</body>
</html>
```

The page shows "Mary" and "Jane" that correspond to the `firstName` and `lastName` properties attached to the `$scope` object:



Components

Components in Angular 1.5.x allow for the encapsulation and capability of creating individual HTML elements. In Angular 1.4.x you could achieve the same feature using the .directive() method.

By using the .component() method, development is simplified gaining the functionality of the directive and the controller. Other benefits include; scope isolation, best practices are inherent, and migration to Angular 2 becomes an easier task. The `<module name>.component()` function call is used to create and register components in AngularJS.

Below is a snippet of code that registers a new component. The `personApp` variable in the snippet references an Angular module, which is defined on line 2.

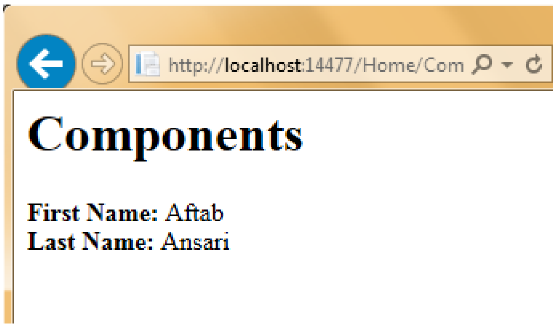
The view where we are displaying the custom HTML element.

```
@{
    Layout = "";
}
<html>
<body ng-app="personApp">
    <h1>Components</h1>

 <person-component></person-component>

    <script src="~/lib/angular/angular.js"></script>
    <script src="~/app/components.js"></script>
</body>
</html>
```

The associated template used by component:

```
<div>
    <strong>First Name:</strong> {{vm.firstName}} <br />
    <strong>Last Name:</strong> {{vm.lastName}}
</div>
```

The page shows "Aftab" and "Ansari" that correspond to the `firstName` and `lastName` properties attached to the `vm` object:



Services

Services in AngularJS are commonly used for shared code that is abstracted away into a file which can be used throughout the lifetime of an Angular application. Services are lazily instantiated, meaning that there will not be an instance of a service unless a component that depends on the service gets used. Factories are an example of a service used in AngularJS applications. Factories are created using the `myApp.factory()` function call, where `myApp` is the module.

Below is an example that shows how to use factories in AngularJS:

To call this factory from the controller, pass `personFactory` as a parameter to the `controller` function:

Using services to talk to a REST endpoint

Below is an end-to-end example using services in AngularJS to interact with an ASP.NET Core Web API endpoint. The example gets data from the Web API and displays the data in a view template. Let's start with the view first:

```
@{
    Layout = "";
}
<html>
<body ng-app="PersonsApp">
    <h1>People</h1>

    <div ng-controller="personController">
        <ul>
            <li ng-repeat="person in people">
                <h2>{{person.FirstName}} {{person.LastName}}</h2>
            </li>
        </ul>
    </div>

    <script src="~/lib/angular/angular.js"></script>
    <script src="~/app/personApp.js"></script>
    <script src="~/app/personFactory.js"></script>
    <script src="~/app/personController.js"></script>
</body>
</html>
```

In this view, we have an Angular module called `PersonsApp` and a controller called `personController`. We are using `ng-repeat` to iterate over the list of persons. We are referencing three custom JavaScript files on lines 17-19.

The *personApp.js* file is used to register the `PersonsApp` module; and, the syntax is similar to previous examples. We are using the `angular.module` function to create a new instance of the module that we will be working with.

Let's take a look at *personFactory.js*, below. We are calling the module's `factory` method to create a factory. Line 12 shows the built-in Angular `$http` service retrieving people information from a web service.

In *personController.js*, we are calling the module's `controller` method to create the controller. The `$scope` object's `people` property is assigned the data returned from the personFactory (line 13).

Let's take a quick look at the Web API and the model behind it. The `Person` model is a POCO (Plain Old CLR Object) with `Id`, `FirstName`, and `LastName` properties:

```
namespace AngularSample.Models
{
    public class Person
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

The `Person` controller returns a JSON-formatted list of `Person` objects:

```
using AngularSample.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace AngularSample.Controllers.Api
{
    public class PersonController : Controller
    {
        [Route("/api/people")]
        public JsonResult GetPeople()
        {
            var people = new List<Person>()
            {
                new Person { Id = 1, FirstName = "John", LastName = "Doe" },
                new Person { Id = 1, FirstName = "Mary", LastName = "Jane" },
                new Person { Id = 1, FirstName = "Bob", LastName = "Parker" }
            };

            // default behavior will convert fields in model to camelCase in Json object
            return Json(people);
        }
    }
}
```

Let's see the application in action:

You can view the application's structure on GitHub.

Note

For more on structuring AngularJS applications, see John Papa's Angular Style Guide

Note

To create AngularJS module, controller, factory, directive and view files easily, be sure to check out Sayed Hashimi's SideWaffle template pack for Visual Studio. Sayed Hashimi is a Senior Program Manager on the Visual Studio Web Team at Microsoft and SideWaffle templates are considered the gold standard. At the time of this writing, SideWaffle is available for Visual Studio 2012, 2013, and 2015.

Routing and multiple views

AngularJS has a built-in route provider to handle SPA (Single Page Application) based navigation. To work with routing in AngularJS, you must add the `angular-route` library using Bower. You can see in the bower.json file referenced at the start of this article that we are already referencing it in our project.

After you install the package, add the script reference (*angular-route.js*) to your view.

Now let's take the Person App we have been building and add navigation to it. First, we will make a copy of the app by creating a new `PeopleController` action called `Spa` and a corresponding `Spa.cshtml` view by copying the Index.cshtml view in the `People` folder. Add a script reference to `angular-route` (see line 11). Also add a `div` marked with the `ng-view` directive (see line 6) as a placeholder to place views in. We are going to be using several additional *.js* files which are referenced on lines 13-16.

```
@{
    Layout = "";
}
<html>
<body ng-app="personApp">
    <div ng-view>

    </div>

    <script src="~/lib/angular/angular.js"></script>
    <script src="~/lib/angular-route/angular-route.js"></script>

    <script src="~/app/personModule.js"></script>
    <script src="~/app/personRoutes.js"></script>
    <script src="~/app/personListController.js"></script>
    <script src="~/app/personDetailController.js"></script>
</body>
</html>
```

Let's take a look at *personModule.js* file to see how we are instantiating the module with routing. We are passing `ngRoute` as a library into the module. This module handles routing in our application.

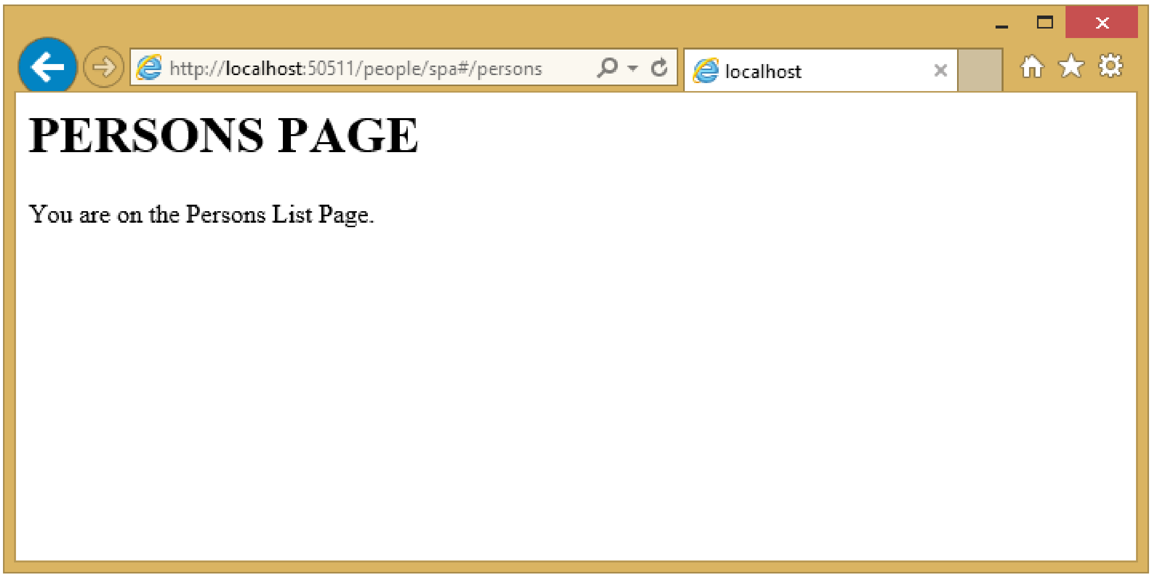
The *personRoutes.js* file, below, defines routes based on the route provider. Lines 4-7 define navigation by effectively saying, when a URL with `/persons` is requested, use a template called `partials/personlist` by working through `personListController`. Lines 8-11 indicate a detail page with a route parameter of `personId`. If the URL doesn't match one of the patterns, Angular defaults to the `/persons` view.

The `personlist.html` file is a partial view containing only the HTML needed to display person list.

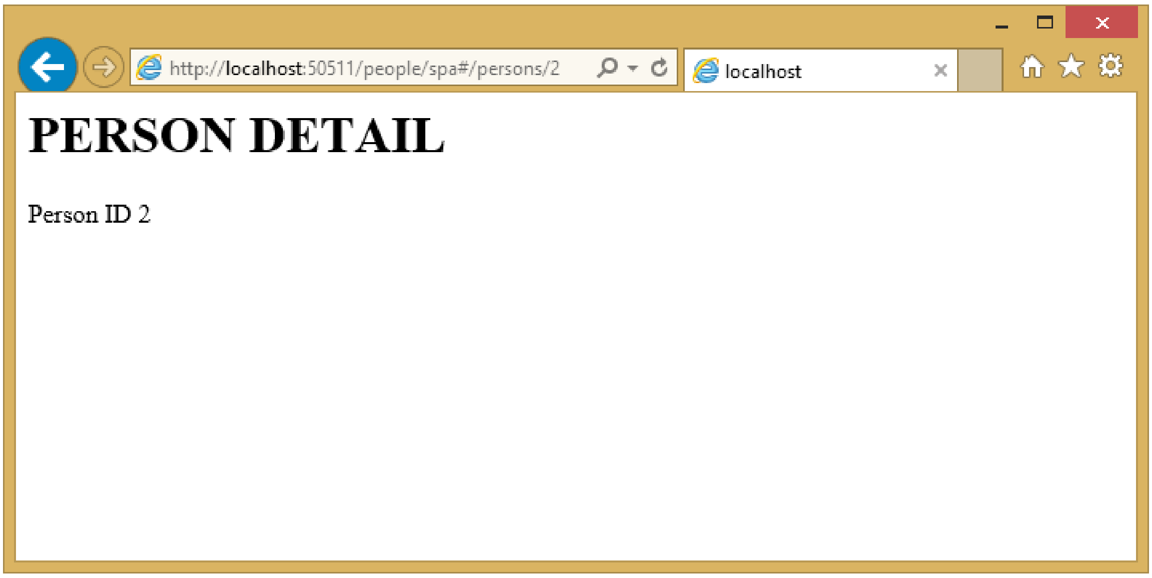
```
<div>
    <h1>PERSONS PAGE</h1>
    <span ng-bind="message"/>
</div>
```

The controller is defined by using the module's `controller` function in *personListController.js*.

If we run this application and navigate to the `people/spa#/persons` URL, we will see:



If we navigate to a detail page, for example `people/spa#/persons/2`, we will see the detail partial view:



You can view the full source and any files not shown in this article on GitHub.

Event Handlers

There are a number of directives in AngularJS that add event-handling capabilities to the input elements in your HTML DOM. Below is a list of the events that are built into AngularJS.

`ng-click`

`ng-dbl-click`

`ng-mousedown`

`ng-mouseup`

`ng-mouseenter`

`ng-mouseleave`

`ng-mousemove`

`ng-keydown`

`ng-keyup`

`ng-keypress`

`ng-change`

Note

You can add your own event handlers using the custom directives feature in AngularJS.

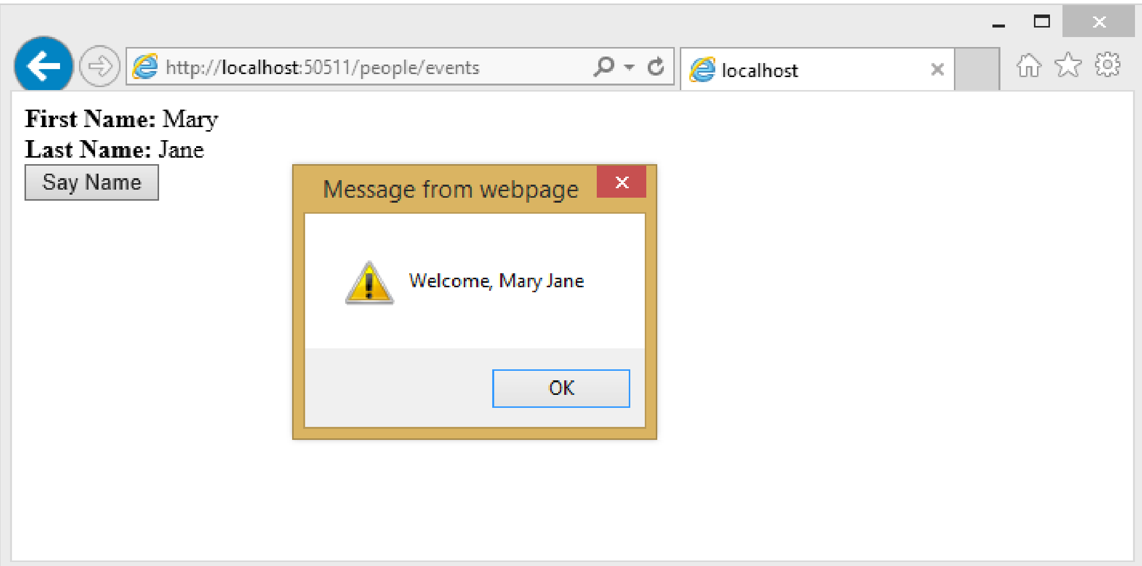
Let's look at how the `ng-click` event is wired up. Create a new JavaScript file named *eventHandlerController.js*, and add the following to it:

Notice the new `sayName` function in `eventHandlerController` on line 5 above. All the method is doing for now is showing a JavaScript alert to the user with a welcome message.

The view below binds a controller function to an AngularJS event. Line 9 has a button on which the `ng-click` Angular directive has been applied. It calls our `sayName` function, which is attached to the `$scope` object passed to this view.

```
@{
    Layout = "";
}
<html>
<body ng-app="personApp">
    <div ng-controller="eventHandlerController">
        <strong>First Name:</strong> {{firstName}} <br />
        <strong>Last Name:</strong> {{lastName}} <br />
        <input ng-click="sayName()" type="button" value="Say Name" />
    </div>
    <script src="~/lib/angular/angular.js"></script>
    <script src="~/lib/angular-route/angular-route.js"></script>

    <script src="~/app/personModule.js"></script>
    <script src="~/app/eventHandlerController.js"></script>
</body>
</html>
```

The running example demonstrates that the controller's `sayName` function is called automatically when the button is clicked.



For more detail on AngularJS built-in event handler directives, be sure to head to the documentation website of AngularJS.

Additional resources

Angular Docs

Introduction to styling applications with Less, Sass, and Font Awesome in ASP.NET Core

By Steve Smith

Users of web applications have increasingly high expectations when it comes to style and overall experience. Modern web applications frequently leverage rich tools and frameworks for defining and managing their look and feel in a consistent manner. Frameworks like Bootstrap can go a long way toward defining a common set of styles and layout options for web sites. However, most non-trivial sites also benefit from being able to effectively define and maintain styles and cascading style sheet (CSS) files, as well as having easy access to non-image icons that help make the site's interface more intuitive. That's where languages and tools that support Less and Sass, and libraries like Font Awesome, come in.

CSS preprocessor languages

Languages that are compiled into other languages, in order to improve the experience of working with the underlying language, are referred to as preprocessors. There are two popular preprocessors for CSS: Less and Sass. These preprocessors add features to CSS, such as support for variables and nested rules, which improve the maintainability of large, complex stylesheets. CSS as a language is very basic, lacking support even for something as simple as variables, and this tends to make CSS files repetitive and bloated. Adding real programming language features via preprocessors can help reduce duplication and provide better organization of styling rules. Visual Studio provides built-in support for both Less and Sass, as well as extensions that can further improve the development experience when working with these languages.

As a quick example of how preprocessors can improve readability and maintainability of style information, consider this 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;
}
```

Using Less, this can be rewritten to eliminate all of the duplication, using a *mixin* (so named because it allows you to "mix in" properties from one class or rule-set into another):

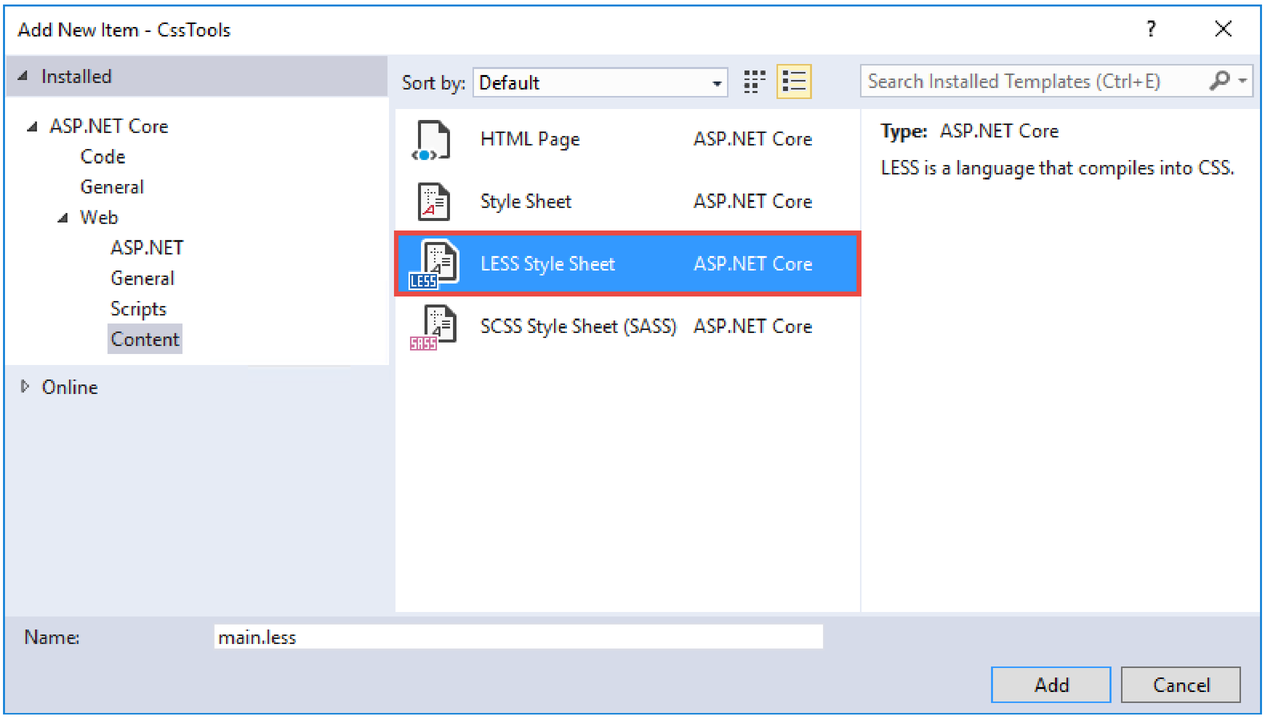
```
.header {
    color: black;
    font-weight: bold;
    font-size: 18px;
    font-family: Helvetica, Arial, sans-serif;
}

.small-header {
    .header;
    font-size: 14px;
}
```

Less

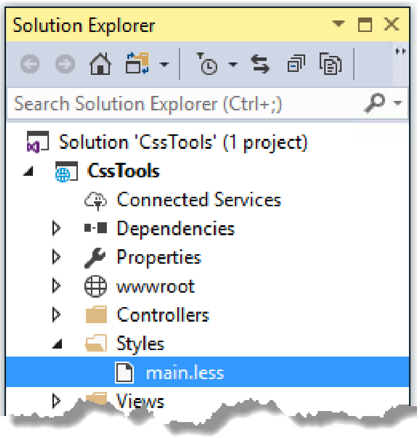
The Less CSS preprocessor runs using Node.js. To install Less, use Node Package Manager (npm) from a command prompt (-g means "global"):

```
npm install -g less
```

If you're using Visual Studio, you can get started with Less by adding one or more Less files to your project, and then configuring Gulp (or Grunt) to process them at compile-time. Add a *Styles* folder to your project, and then add a new Less file named *main.less* to this folder.



Once added, your folder structure should look something like this:



Now you can add some basic styling to the file, which will be compiled into CSS and deployed to the wwwroot folder by Gulp.

Modify *main.less* to include the following content, which creates a simple color palette from a single base color.

```
@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` and the other @-prefixed items are variables. Each of them represents a color. Except for `@base`, they are set using color functions: lighten, darken, and spin. Lighten and darken do pretty much what you would expect; spin adjusts the hue of a color by a number of degrees (around the color wheel). The Less processor is smart enough to ignore variables that aren't used, so to demonstrate how these variables work, we need to use them somewhere. The classes `.baseColor`, etc. will demonstrate the calculated values of each of the variables in the CSS file that is produced.

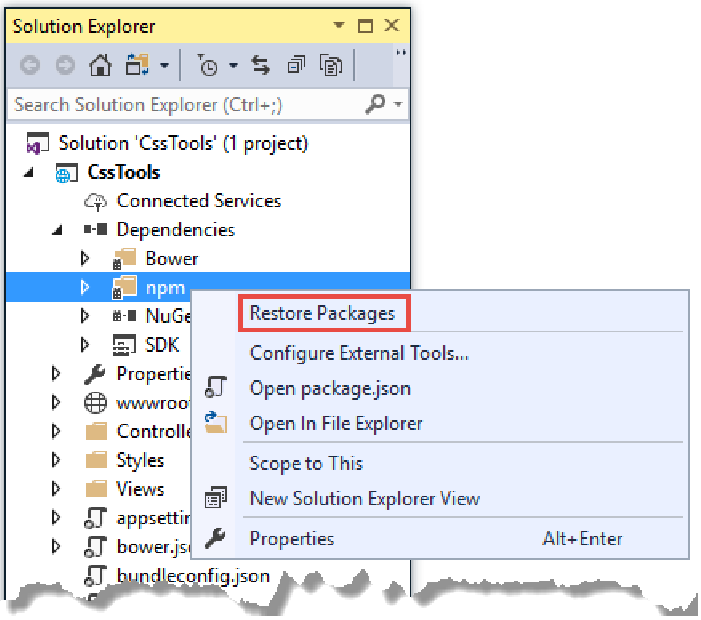
Getting started

Create an **npm Configuration File** (*package.json*) in your project folder and edit it to reference `gulp` and `gulp-less`:

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "devDependencies": {
    "gulp": "3.9.1",
    "gulp-less": "3.3.0"
  }
}
```

Install the dependencies either at a command prompt in your project folder, or in Visual Studio **Solution Explorer** (**Dependencies > npm > Restore packages**).

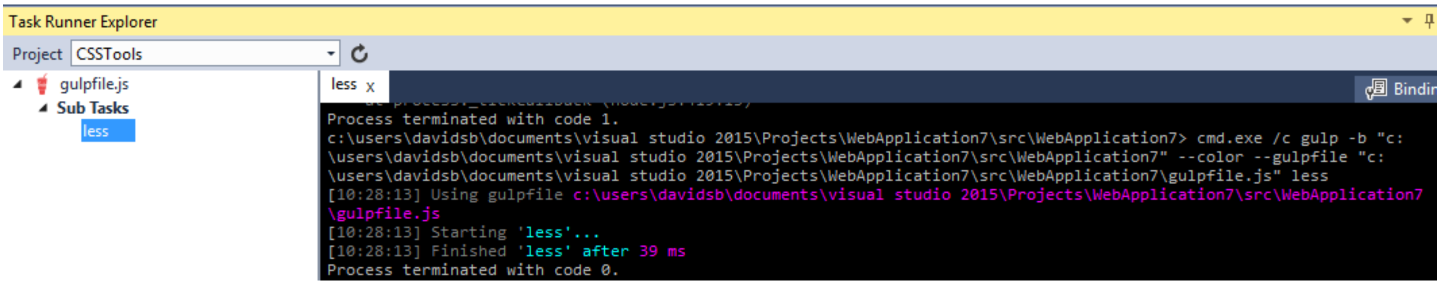
```
npm install
```



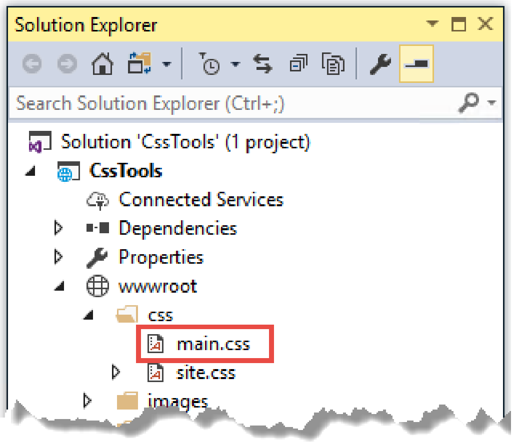
In the project folder, create a **Gulp Configuration File** (*gulpfile.js*) to define the automated process. Add a variable at the top of the file to represent Less, and a task to run Less:

Open the **Task Runner Explorer** (**View > Other Windows > Task Runner Explorer**). Among the tasks, you should see a new task named `less`. You might have to refresh the window.

Run the `less` task, and you see output similar to what is shown here:



The *wwwroot/css* folder now contains a new file, *main.css*:



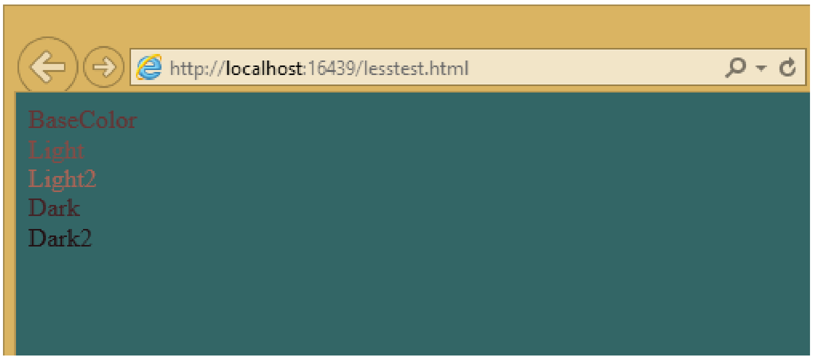
Open *main.css* and you see something like the following:

```
body {
    background-color: #336666;
}
.baseColor {
    color: #663333;
}
.bgLight {
    color: #884a44;
}
.bgLight2 {
    color: #aa6355;
}
.bgDark {
    color: #442225;
}
.bgDark2 {
    color: #221114;
}
```

Add a simple HTML page to the *wwwroot* folder, and reference *main.css* to see the color palette in action.

```
<!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>
```

You can see that the 180 degree spin on `@base` used to produce `@background` resulted in the color wheel opposing color of `@base`:



Less also provides support for nested rules, as well as nested media queries. For example, defining nested hierarchies like menus can result in verbose CSS rules like these:

```
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;
}
```

Ideally all of the related style rules will be placed together within the CSS file, but in practice there is nothing enforcing this rule except convention and perhaps block comments.

Defining these same rules using Less looks like this:

```
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}
        }
    }
}
```

Note that in this case, all of the subordinate elements of `nav` are contained within its scope. There is no longer any repetition of parent elements (`nav`, `li`, `a`), and the total line count has dropped as well (though some of that is a result of putting values on the same lines in the second example). It can be very helpful, organizationally, to see all of the rules for a given UI element within an explicitly bounded scope, in this case set off from the rest of the file by curly braces.

The `&` syntax is a Less selector feature, with & representing the current selector parent. So, within the a {...} block, `&` represents an `a` tag, and thus `&:link` is equivalent to `a:link`.

Media queries, extremely useful in creating responsive designs, can also contribute heavily to repetition and complexity in CSS. Less allows media queries to be nested within classes, so that the entire class definition doesn't need to be repeated within different top-level `@media` elements. For example, here is CSS for a responsive menu:

```
.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;
    }
}
```

This can be better defined in Less as:

```
.navigation {
    margin-top: 30%;
    width: 100%;
    @media screen and (min-width: 40em) {
        margin: 0;
    }
    @media screen and (min-width: 62em) {
        width: 960px;
        margin: 0;
    }
}
```

Another feature of Less that we have already seen is its support for mathematical operations, allowing style attributes to be constructed from pre-defined variables. This makes updating related styles much easier, since the base variable can be modified and all dependent values change automatically.

CSS files, especially for large sites (and especially if media queries are being used), tend to get quite large over time, making working with them unwieldy. Less files can be defined separately, then pulled together using `@import` directives. Less can also be used to import individual CSS files, as well, if desired.

Mixins can accept parameters, and Less supports conditional logic in the form of mixin guards, which provide a declarative way to define when certain mixins take effect. A common use for mixin guards is to adjust colors based on how light or dark the source color is. Given a mixin that accepts a parameter for color, a mixin guard can be used to modify the mixin based on that color:

```
.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);
}
```

Given our current `@base` value of `#663333`, this Less script will produce the following CSS:

```
.feature {
    background-color: #FFF;
    color: #663333;
}
```

Less provides a number of additional features, but this should give you some idea of the power of this preprocessing language.

Sass

Sass is similar to Less, providing support for many of the same features, but with slightly different syntax. It is built using Ruby, rather than JavaScript, and so has different setup requirements. The original Sass language did not use curly braces or semicolons, but instead defined scope using white space and indentation. In version 3 of Sass, a new syntax was introduced, **SCSS** ("Sassy CSS"). SCSS is similar to CSS in that it ignores indentation levels and whitespace, and instead uses semicolons and curly braces.

To install Sass, typically you would first install Ruby (pre-installed on Mac), and then run:

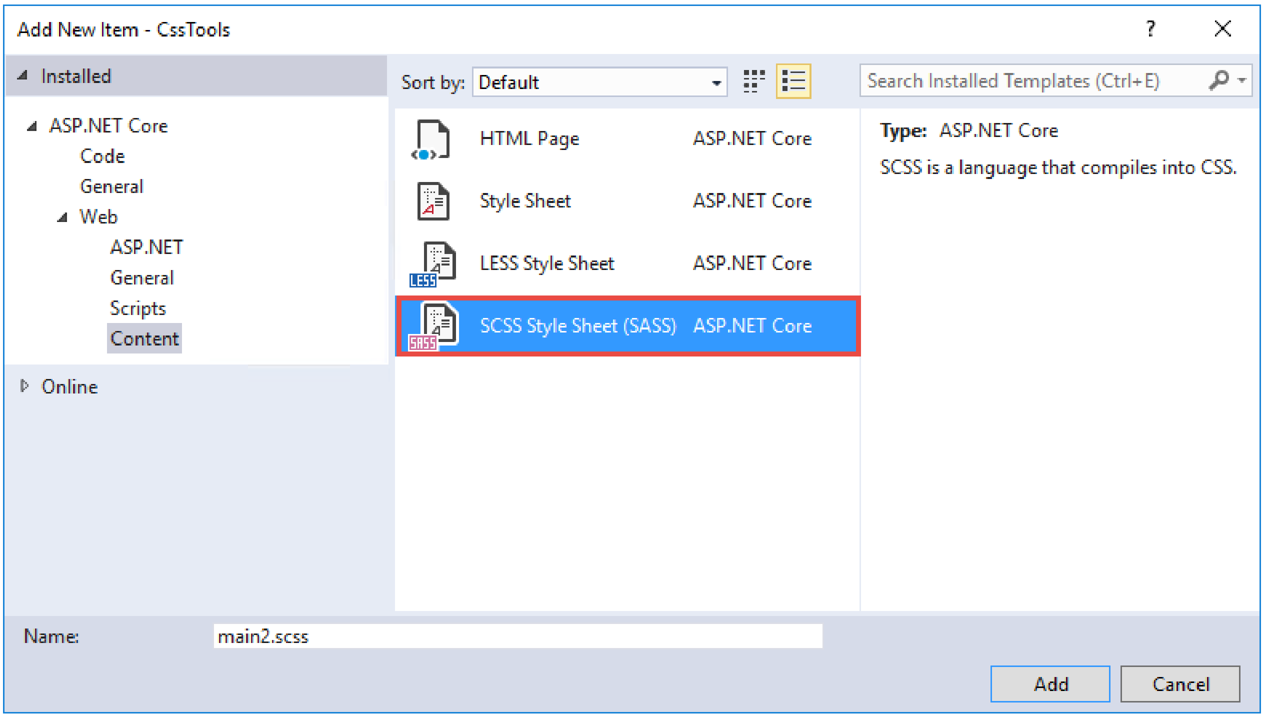
```
gem install sass
```

However, if you're running Visual Studio, you can get started with Sass in much the same way as you would with Less. Open *package.json* and add the "gulp-sass" package to `devDependencies`:

```
"devDependencies": {
  "gulp": "3.9.1",
  "gulp-less": "3.3.0",
  "gulp-sass": "3.1.0"
}
```

Next, modify *gulpfile.js* to add a sass variable and a task to compile your Sass files and place the results in the wwwroot folder:

Now you can add the Sass file *main2.scss* to the *Styles* folder in the root of the project:



Open *main2.scss* and add the following:

```
$base: #CC0000;
body {
    background-color: $base;
}
```

Save all of your files. Now when you refresh **Task Runner Explorer**, you see a `sass` task. Run it, and look in the */wwwroot/css* folder. There is now a *main2.css* file, with these contents:

```
body {
    background-color: #CC0000;
}
```

Sass supports nesting in much the same was that Less does, providing similar benefits. Files can be split up by function and included using the `@import` directive:

```
@import 'anotherfile';
```

Sass supports mixins as well, using the `@mixin` keyword to define them and `@include` to include them, as in this example from 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); }
```

In addition to mixins, Sass also supports the concept of inheritance, allowing one class to extend another. It's conceptually similar to a mixin, but results in less CSS code. It's accomplished using the `@extend` keyword. To try out mixins, add the following to your *main2.scss* file:

```
@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;
}
```

Examine the output in *main2.css* after running the `sass` task in **Task Runner Explorer**:

```
.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;
}
```

Notice that all of the common properties of the alert mixin are repeated in each class. The mixin did a good job of helping eliminate duplication at development time, but it's still creating CSS with a lot of duplication in it, resulting in larger than necessary CSS files - a potential performance issue.

Now replace the alert mixin with a `.alert` class, and change `@include` to `@extend` (remembering to extend `.alert`, not `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;
}
```

Run Sass once more, and examine the resulting 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;
}
```

Now the properties are defined only as many times as needed, and better CSS is generated.

Sass also includes functions and conditional logic operations, similar to Less. In fact, the two languages' capabilities are very similar.

Less or Sass?

There is still no consensus as to whether it's generally better to use Less or Sass (or even whether to prefer the original Sass or the newer SCSS syntax within Sass). Probably the most important decision is to **use one of these tools**, as opposed to just hand-coding your CSS files. Once you've made that decision, both Less and Sass are good choices.

Font Awesome

In addition to CSS preprocessors, another great resource for styling modern web applications is Font Awesome. Font Awesome is a toolkit that provides over 500 scalable vector icons that can be freely used in your web applications. It was originally designed to work with Bootstrap, but it has no dependency on that framework or on any JavaScript libraries.

The easiest way to get started with Font Awesome is to add a reference to it, using its public content delivery network (CDN) location:

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

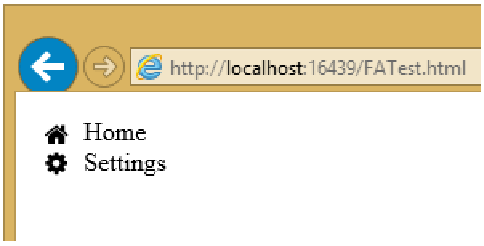
You can also add it to your Visual Studio project by adding it to the "dependencies" in *bower.json*:

```
{
  "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"
  }
}
```

Once you have a reference to Font Awesome on a page, you can add icons to your application by applying Font Awesome classes, typically prefixed with "fa-", to your inline HTML elements (such as `<span>` or `<i>`). For example, you can add icons to simple lists and menus using code like this:

```
<!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>
```

This produces the following in the browser - note the icon beside each item:



You can view a complete list of the available icons here:

http://fontawesome.io/icons/

Summary

Modern web applications increasingly demand responsive, fluid designs that are clean, intuitive, and easy to use from a variety of devices. Managing the complexity of the CSS stylesheets required to achieve these goals is best done using a preprocessor like Less or Sass. In addition, toolkits like Font Awesome quickly provide well-known icons to textual navigation menus and buttons, improving the overall user experience of your application.

Bundling and minification in ASP.NET Core

Bundling and minification are two techniques you can use in ASP.NET to improve page load performance for your web application. Bundling combines multiple files into a single file. Minification performs a variety of different code optimizations to scripts and CSS, which results in smaller payloads. Used together, bundling and minification improves load time performance by reducing the number of requests to the server and reducing the size of the requested assets (such as CSS and JavaScript files).

This article explains the benefits of using bundling and minification, including how these features can be used with ASP.NET Core applications.

Overview

In ASP.NET Core apps, there are multiple options for bundling and minifying client-side resources. The core templates for MVC provide an out-of-the-box solution using a configuration file and BuildBundlerMinifier NuGet package. Third party tools, such as Gulp and Grunt are also available to accomplish the same tasks should your processes require additional workflow or complexities. By using design-time bundling and minification, the minified files are created prior to the application's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that design-time bundling and minification increases build complexity and only works with static files.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the assets (JavaScript, CSS and images) so bundling and minification won't provide any performance boost when requesting the same page, or pages on the same site requesting the same assets. If you don't set the expires header correctly on your assets, and you don't use bundling and minification, the browser's freshness heuristics will mark the assets stale after a few days and the browser will require a validation request for each asset. In this case, bundling and minification provide a performance increase even after the first page request.

Bundling

Bundling is a feature that makes it easy to combine or bundle multiple files into a single file. Because bundling combines multiple files into a single file, it reduces the number of requests to the server that are required to retrieve and display a web asset, such as a web page. You can create CSS, JavaScript and other bundles. Fewer files means fewer HTTP requests from your browser to the server or from the service providing your application. This results in improved first page load performance.

Minification

Minification performs a variety of different code optimizations to reduce the size of requested assets (such as CSS, images, JavaScript files). Common results of minification include removing unnecessary white space and comments, and shortening variable names to one character.

Consider the following JavaScript function:

After minification, the function is reduced to the following:

In addition to removing the comments and unnecessary whitespace, the following parameters and variable names were renamed (shortened) as follows:

ORIGINAL	RENAMED
imageTagAndImageID	t
imageContext	a
imageElement	r

Impact of bundling and minification

The following table shows several important differences between listing all the assets individually and using bundling and minification on a simple web page:

ACTION	WITH B/M	WITHOUT B/M	CHANGE
File Requests	7	18	157%
KB Transferred	156	264.68	70%
Load Time (MS)	885	2360	167%

The bytes sent had a significant reduction with bundling as browsers are fairly verbose with the HTTP headers that they apply on requests. The load time shows a big improvement, however this example was run locally. You will get greater gains in performance when using bundling and minification with assets transferred over a network.

Using bundling and minification in a project

The MVC project template provides a `bundleconfig.json` configuration file which defines the options for each bundle. By default, a single bundle configuration is defined for the custom JavaScript (`wwwroot/js/site.js`) and Stylesheet (`wwwroot/css/site.css`) files.

```
[
  {
    "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
  }
]
```

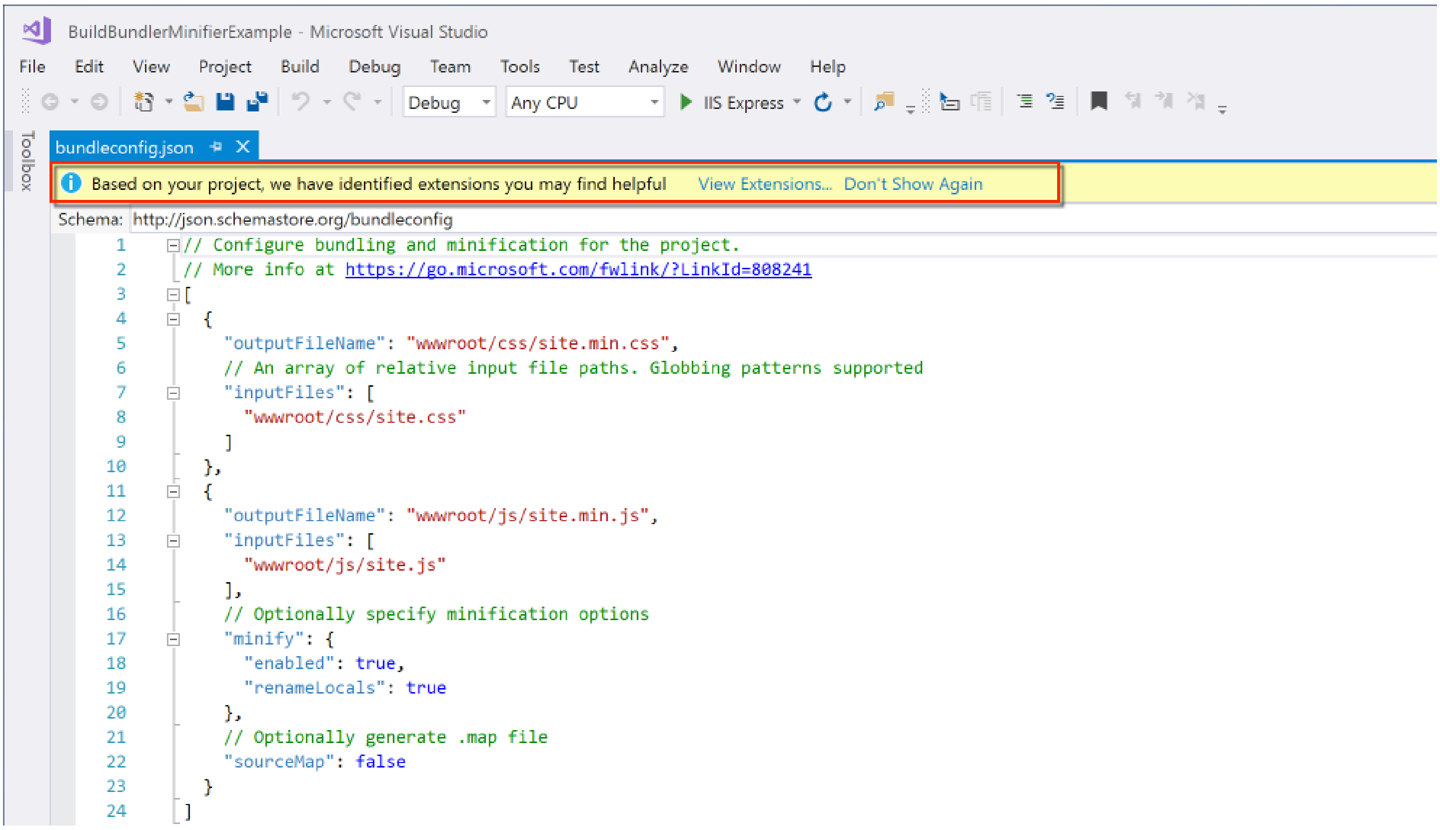
Bundle options include:

outputFileName - name of the bundle file to output. Can contain a relative path from the `bundleconfig.json` file. **required**
inputFiles - array of files to bundle together. These are relative paths to the configuration file. **optional**, *an empty value results in an empty output file. globbing patterns are supported.
minify - minification options for the output type. **optional**, *default* - `minify: { enabled: true }`
Configuration options are available per output file type.
CSS Minifier
JavaScript Minifier
HTML Minifier

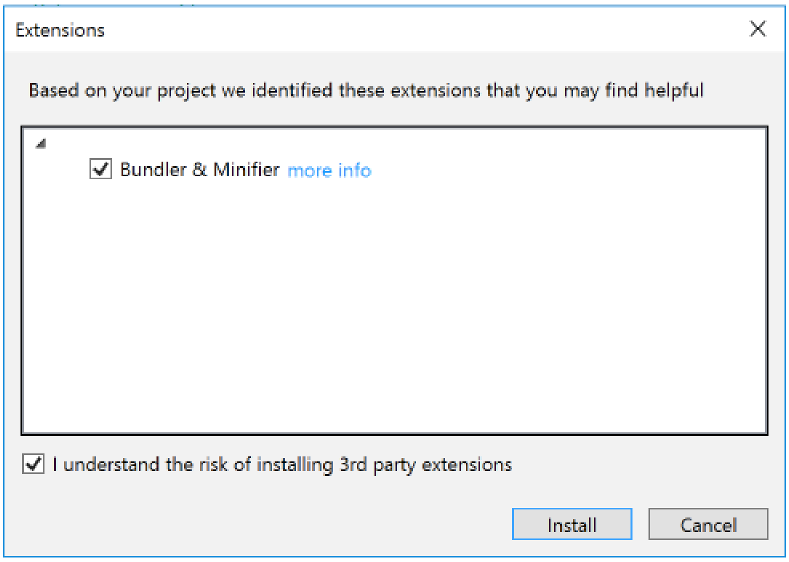
includeInProject - add generated files to project file. **optional**, *default - false*
sourceMaps - generate source maps for the bundled file. **optional**, *default - false*

Visual Studio 2015 / 2017

Open `bundleconfig.json` in Visual Studio, if your environment does not have the extension installed; a prompt is presented suggesting that there is one that could assist with this file type.



Select View Extensions, and install the **Bundler & Minifier** extension (Requires Visual Studio restart).



When the restart is complete, you need to configure the build to run the processes of minifying and bundling the client-side assets. Right-click the `bundleconfig.json` file and select *Enable bundle on build...*.

Build the project, and the `bundleconfig.json` is included in the build process to produce the output files based on the configuration.

```
1>------ Build started: Project: BuildBundlerMinifierExample, Configuration: Debug Any CPU ------
1>
1>Bundler: Begin processing bundleconfig.json
1>Bundler: Done processing bundleconfig.json
1>BuildBundlerMinifierExample ->
C:\BuildBundlerMinifierExample\bin\Debug\netcoreapp1.1\BuildBundlerMinifierExample.dll
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Visual Studio Code or Command Line

Visual Studio and the extension drive the bundling and minification process using GUI gestures; however, the same capabilities are available with the `dotnet` CLI and BuildBundlerMinifier NuGet package.

Add the NuGet package to your project:

```
dotnet add package BuildBundlerMinifier
```

Restore the dependencies:

```
dotnet restore
```

Build the app:

```
dotnet build
```

The output from the build command shows the results of the minification and/or bundling according to what is configured.

```
Microsoft (R) Build Engine version 15.1.545.13942
Copyright (C) Microsoft Corporation. All rights reserved.


  Bundler: Begin processing bundleconfig.json
     Minified wwwroot/css/site.min.css
  Bundler: Done processing bundleconfig.json
  BuildBundlerMinifierExample ->
/BuildBundlerMinifierExample/bin/Debug/netcoreapp1.0/BuildBundlerMinifierExample.dll
```

Adding files

In this example, an additional CSS file is added called `custom.css` and configured for bundling and minification with `site.css`, resulting in a single `site.min.css`.

custom.css

```
.about, [role=main], [role=complementary]
{
    margin-top: 60px;
}

footer
{
    margin-top: 10px;
}
```

Add the relative path to `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
  }
]
```

Note

Alternatively, the globbing pattern could be used - `"inputFiles": ["wwwroot/**/*(*.css|!(*.min.css)"]` which gets all css files and excludes the minified file pattern.

Build the application and if you open `site.min.css`, you'll now notice that contents of `custom.css` has been appended to the end of the file.

Controlling bundling and minification

In general, you want to use the bundled and minified files of your app only in a production environment. During development, you want to use your original files so your app is easier to debug.

You can specify which scripts and CSS files to include in your pages using the environment tag helper in your layout pages (see Tag Helpers). The environment tag helper will only render its contents when running in specific environments. See Working with Multiple Environments for details on specifying the current environment.

The following environment tag will render the unprocessed CSS files when running in the `Development` environment:

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

This environment tag will render the bundled and minified CSS files only when running in `Production` or `Staging`:

```
<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>
```

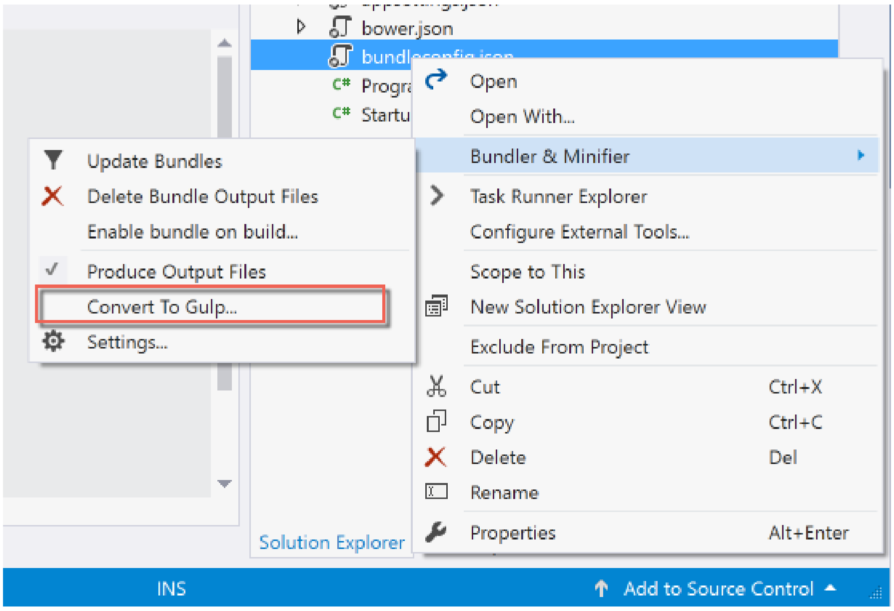
Consuming bundleconfig.json from Gulp

If your app bundling and minification workflow requires additional processes such as image processing, cache busting, CDN assest processing, etc., then you can convert the Bundle and Minify process to Gulp.

Note

Conversion option only available in Visual Studio 2015 and 2017.

Right-click the `bundleconfig.json` and select **Convert to Gulp...**. This will generate the `gulpfile.js` and install the necessary npm packages.



The `gulpfile.js` produced reads the `bundleconfig.json` file for the configuration, therefore it can continue to be used for the inputs/outputs and settings.

To enable Gulp when the project builds in Visual Studio 2017, add the following to the *.csproj file:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
    <Exec Command="gulp min" />
</Target>
```

To enable Gulp when the project builds in Visual Studio 2015, add the following to the `project.json` file:

```
"scripts": {
    "precompile": "gulp min"
}
```

Additional resources

Using Gulp
Using Grunt
Working with Multiple Environments
Tag Helpers

Introduction to building projects with Yeoman in ASP.NET Core

Yeoman is a project scaffolding system for creating many kinds of applications. The Yeoman generator for ASP.NET Core contains a variety of project templates for starting a new web, MVC, or console application.

Install Node.js, npm, and Yeoman

Prerequisites

Node.js and npm are required for Yeoman. Download from Node.js. The installer includes Node.js and npm. Bower is also required for installing UI components like stylesheets.

To install Yeoman and Bower run the following command:

```
npm install -g yo bower
```

Note

If you get the error `npm ERR! Please try running this command again as root/Administrator.` on macOS, run the following command using sudo: `sudo npm install -g yo bower`

From a command prompt, install the ASP.NET generator:

```
npm install -g generator-aspnet
```

Note

If you get a permission error, run the command under `sudo` as described above.

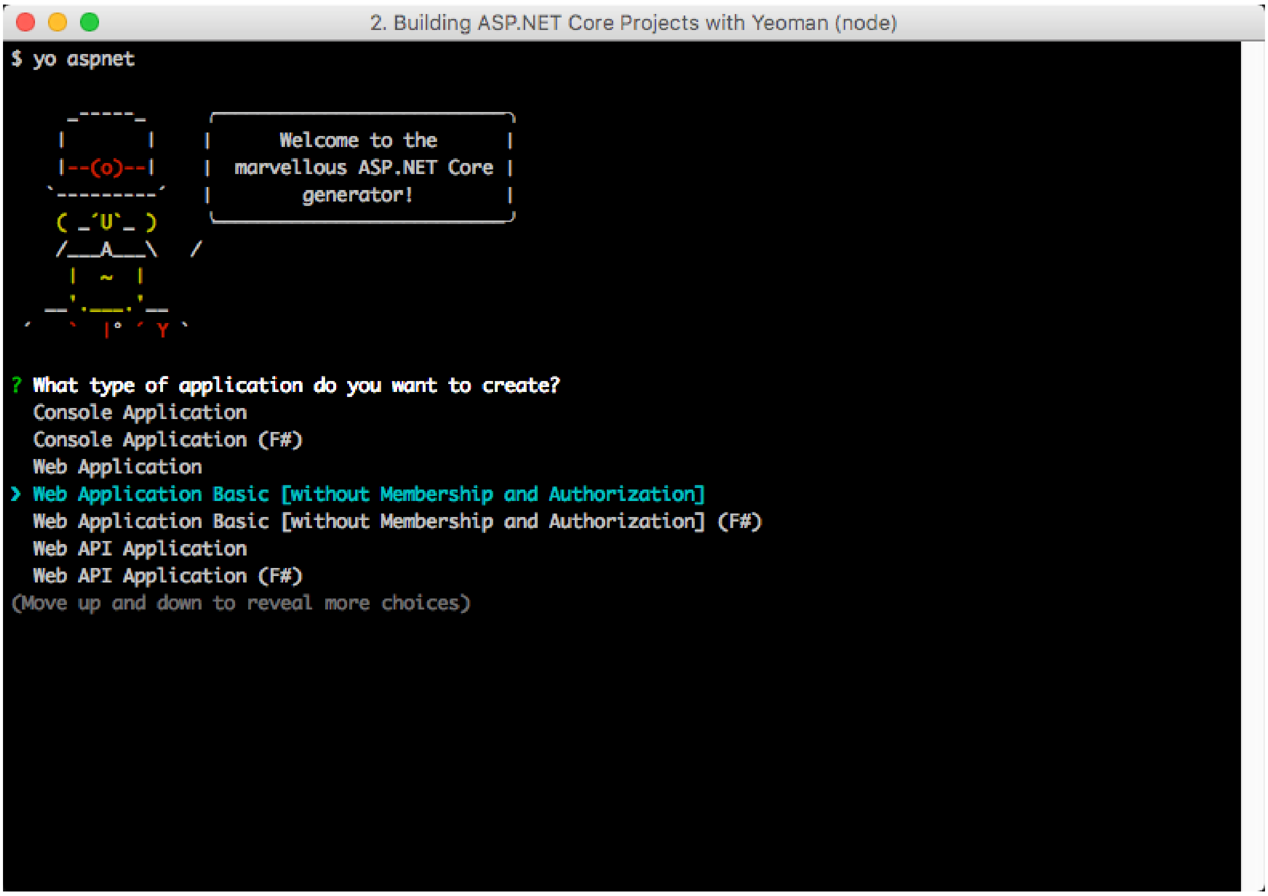
The `-g` flag installs the generator globally, so that it can be used from any path.

Create an ASP.NET app

Run the ASP.NET generator for `yo`

```
yo aspnet
```

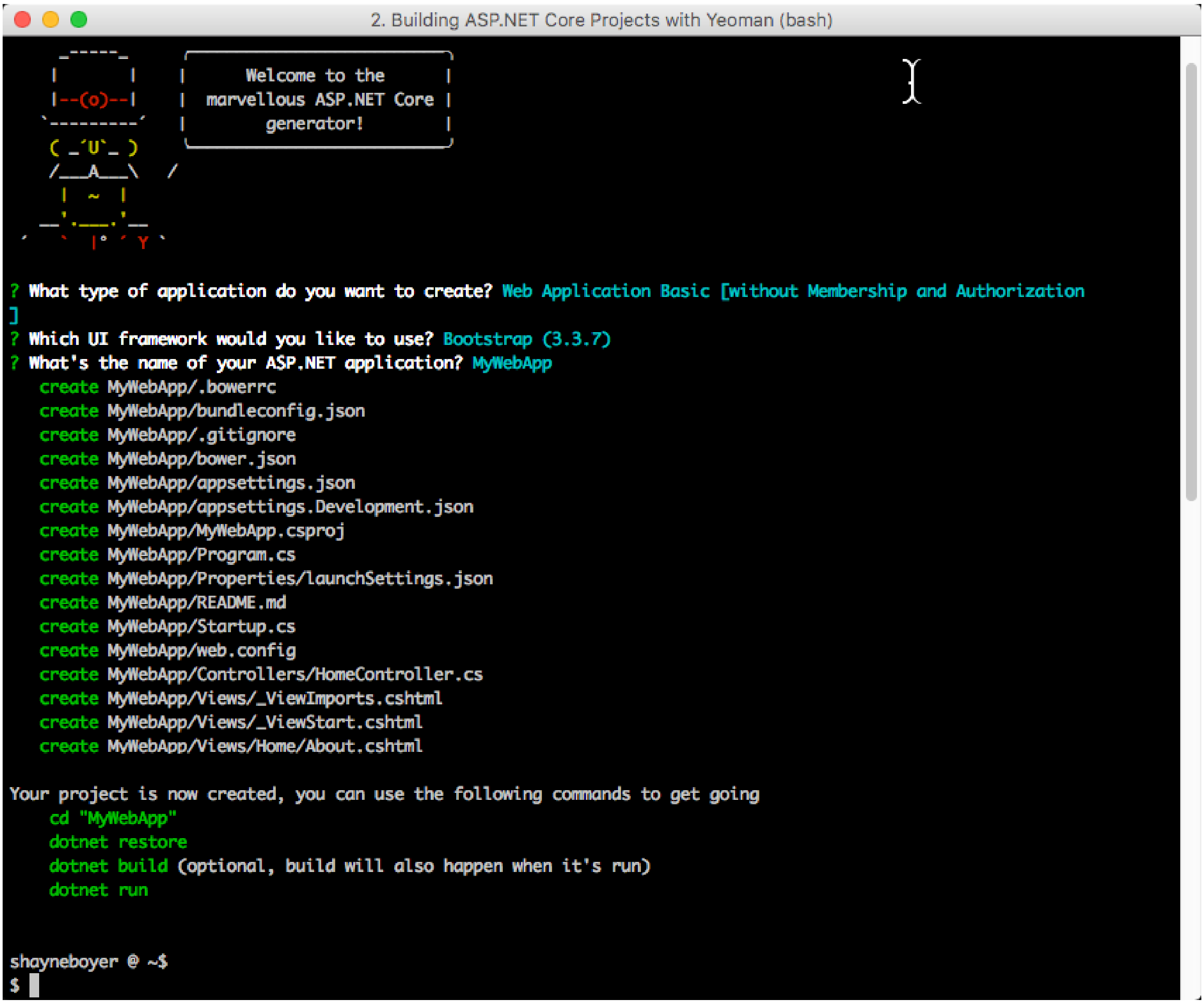
The generator displays a menu. Arrow down to the **Web Application Basic [without Membership and Authorization]** project and tap **Enter**:



Select Bootstrap as the UI Framework and tap **Enter**.

Use "**MyWebApp**" for the app name and then tap **Enter**.

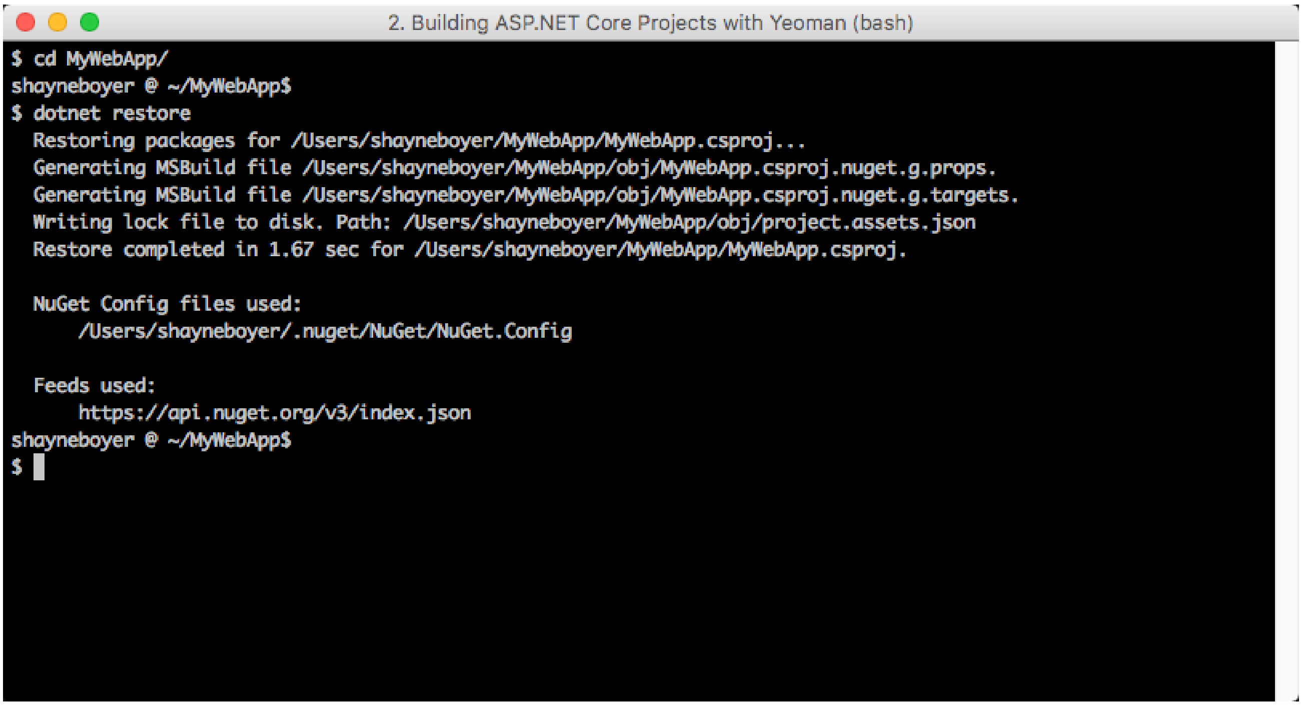
Yeoman will scaffold the project and its supporting files. Suggested next steps are also provided in the form of commands.



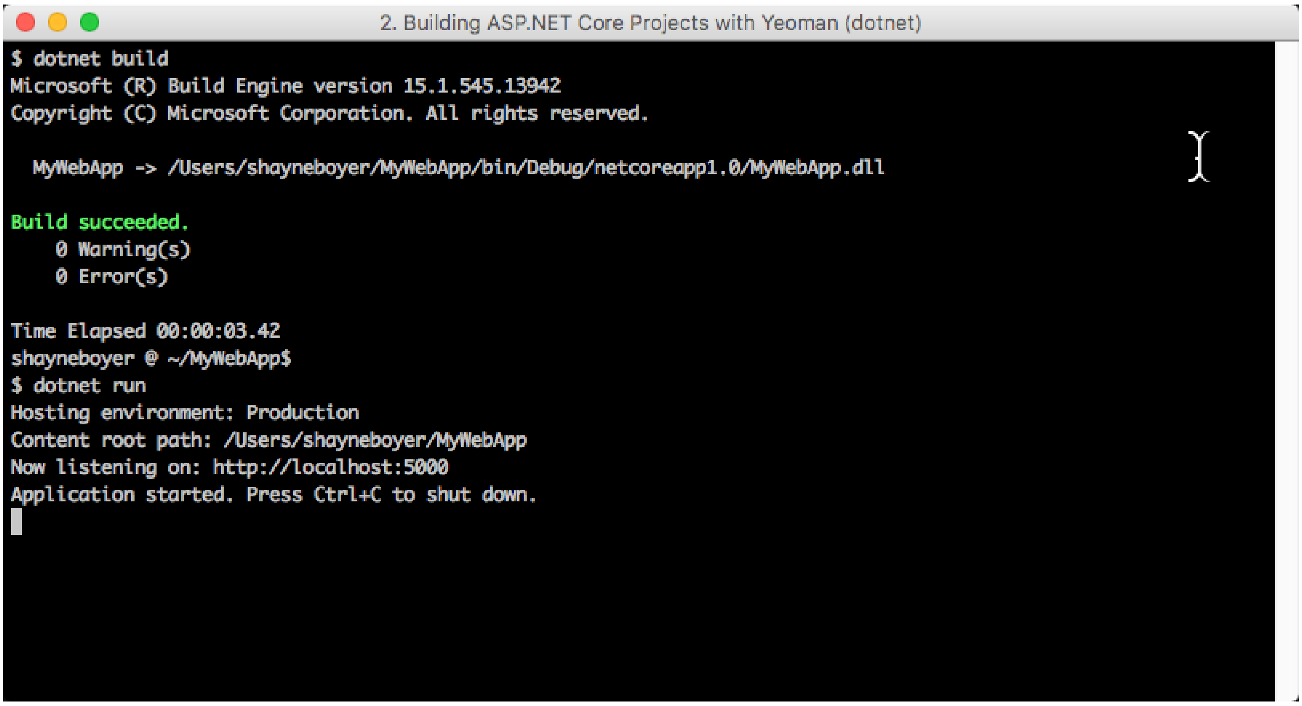
The ASP.NET generator creates ASP.NET Core projects that can be loaded into Visual Studio Code, Visual Studio, or run from the command line.

Restore, build, and run

Follow the suggested commands by changing directories to the `MyWebApp` directory. Then run `dotnet restore`.



Build and run the app using `dotnet build` and `dotnet run`:



At this point you can navigate to the URL shown to test the newly created ASP.NET Core app.

Client-side packages

The front end resources are provided by the templates from the yeoman generator using the Bower client-side package manager, adding *bower.json* and *.bowerrc* files to restore client-side packages using the Bower client-side package manager.

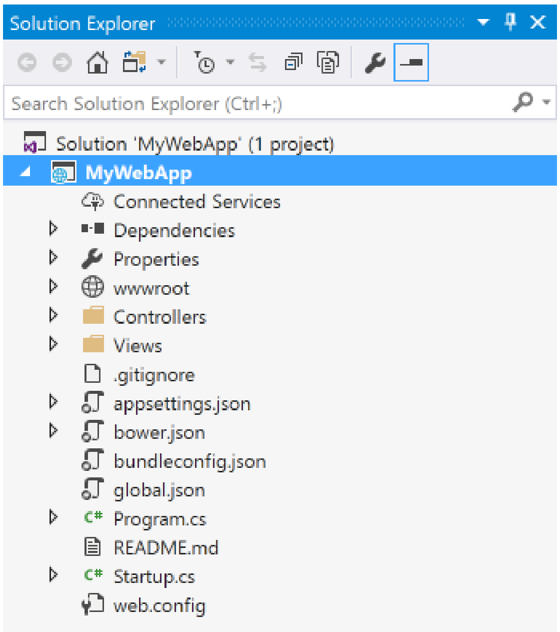
The BundlerMinifier component is also included by default for ease of concatenation (bundling) and minification of CSS, JavaScript and HTML.

Building and running from Visual Studio

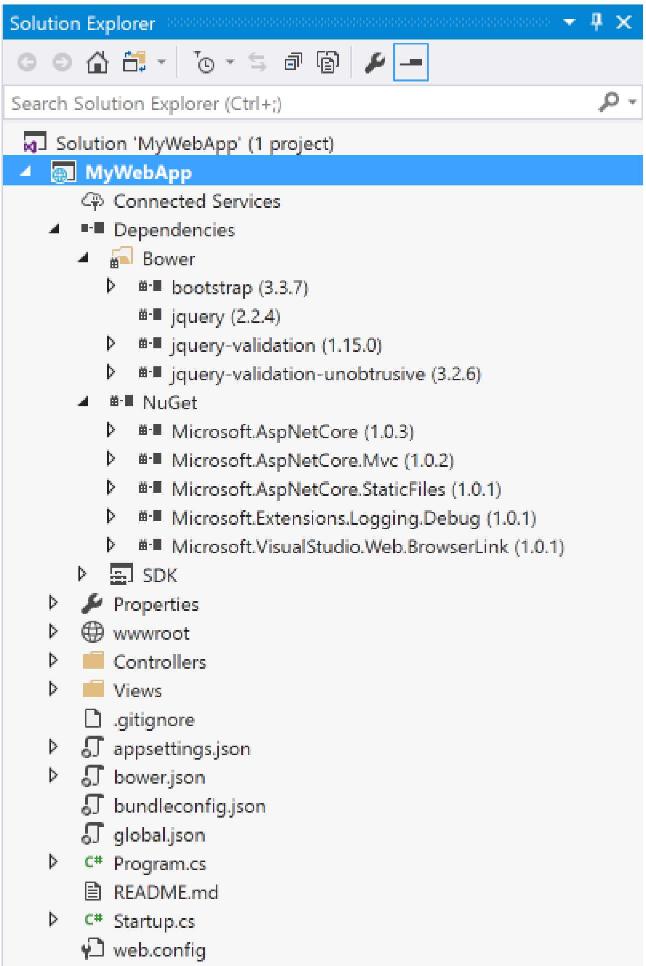
You can load your generated ASP.NET Core web project directly into Visual Studio, then build and run your project from there. Follow the instructions above to scaffold a new ASP.NET Core app using yeoman. This time, choose **Web Application** from the menu and name the app `MyWebApp`.

Open Visual Studio. From the File menu, select Open   Project/Solution.

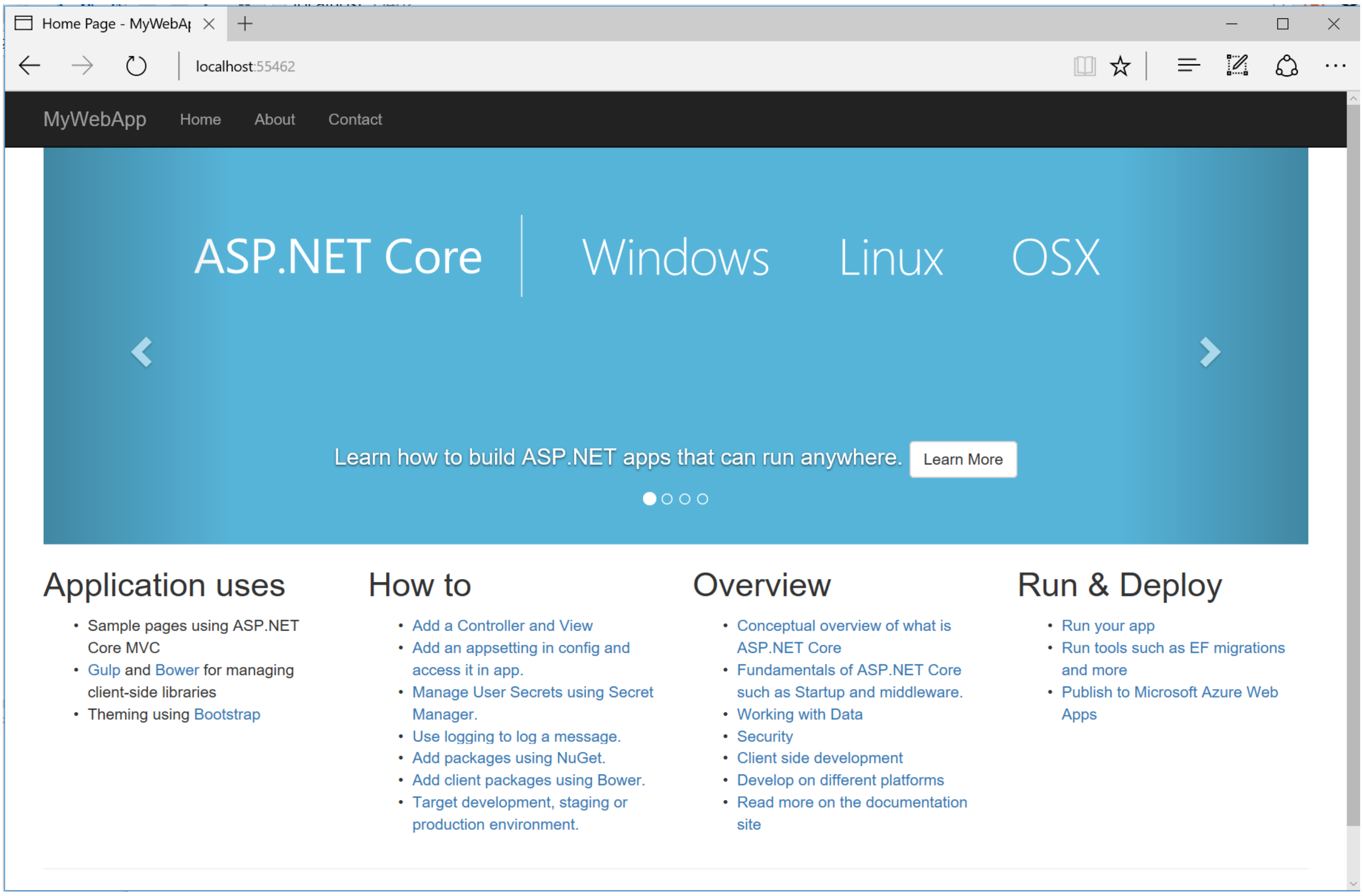
In the Open Project dialog, navigate to the *project.json* file, select it, and click the **Open** button. In the Solution Explorer, the project should look something like the screenshot below.



Yeoman scaffolds a MVC web application, complete with both server- and client-side build support. Server-side dependencies are listed under the **References** node, and client-side dependencies in the **Dependencies** node of Solution Explorer. Dependencies are restored automatically when the project is loaded.



When all the dependencies are restored, press **F5** to run the project. The default home page displays in the browser.



Restoring, building, and hosting from a command line

You can prepare and host your web application using the .NET Core command-line interface.

At a command prompt, change the current directory to the folder containing the project (that is, the folder containing the *project.json* file):

```
cd src\MyWebApp
```

Restore the project's NuGet package dependencies:

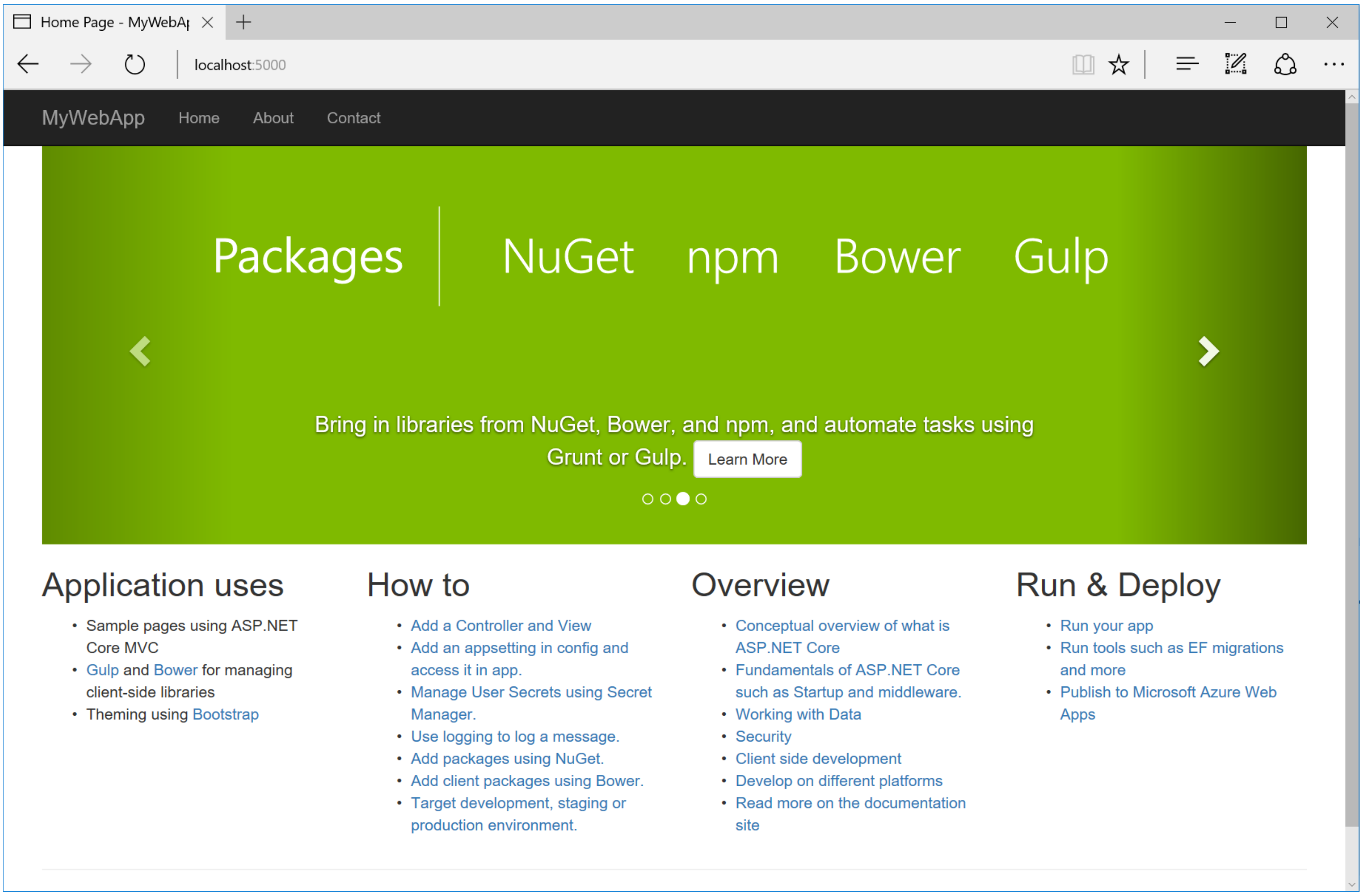
```
dotnet restore
```

Run the application:

```
dotnet run
```

The cross-platform Kestrel web server will begin listening on port 5000.

Open a web browser, and navigate to `http://localhost:5000`.



Adding to your project with sub generators

You can add new generated files using Yeoman even after the project is created. Use sub generators to add any of the file types that make up your project. For example, to add a new class to your project, enter the `yo aspnet:Class` command followed by the name of the class. Execute the following command from the directory in which the file should be created:

```
yo aspnet:Class Person
```

The result is a file named Person.cs with a class named `Person`:

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

namespace MyWebApp
{
    public class Person
    {
        public Person()
        {
        }
    }
}
```

Additional resources

Servers (Kestrel and WebListener)

Your First ASP.NET Core Application on a Mac Using Visual Studio Code

Introduction to Browser Link in ASP.NET Core

By Nicolò Carandini, Mike Wasson, and Tom Dykstra

Browser Link is a feature in Visual Studio that creates a communication channel between the development environment and one or more web browsers. You can use Browser Link to refresh your web application in several browsers at once, which is useful for cross-browser testing.

Browser Link setup

The ASP.NET Core **Web Application** project templates in Visual Studio 2015 and later include everything needed for Browser Link.

To add Browser Link to a project that you created by using the ASP.NET Core **Empty** or **Web API** template, follow these steps:

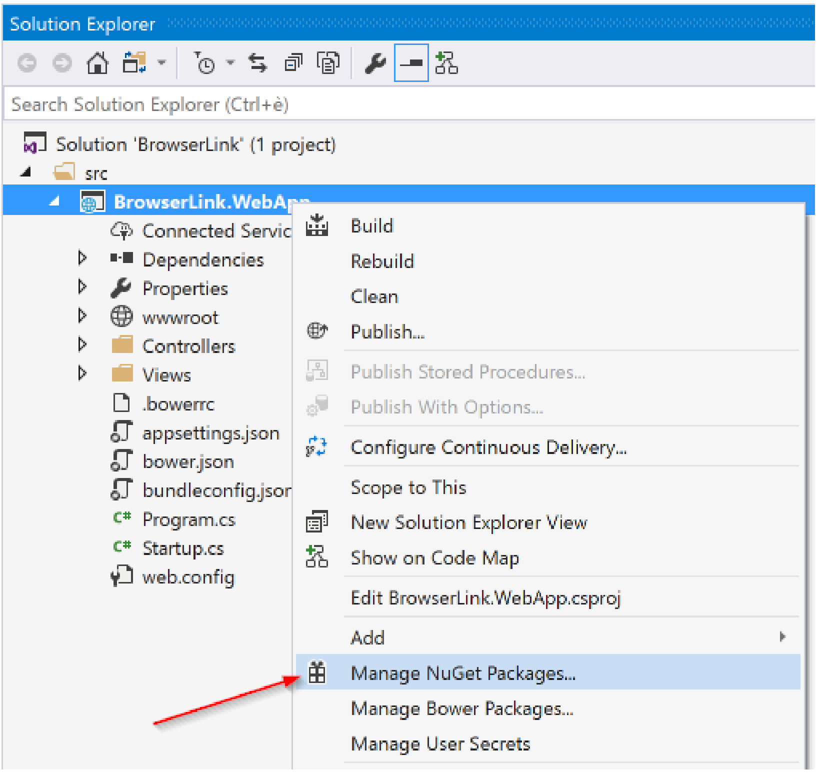
Add the *Microsoft.VisualStudio.Web.BrowserLink.Loader* package
Add configuration code in the *Startup.cs* file.

Add the package

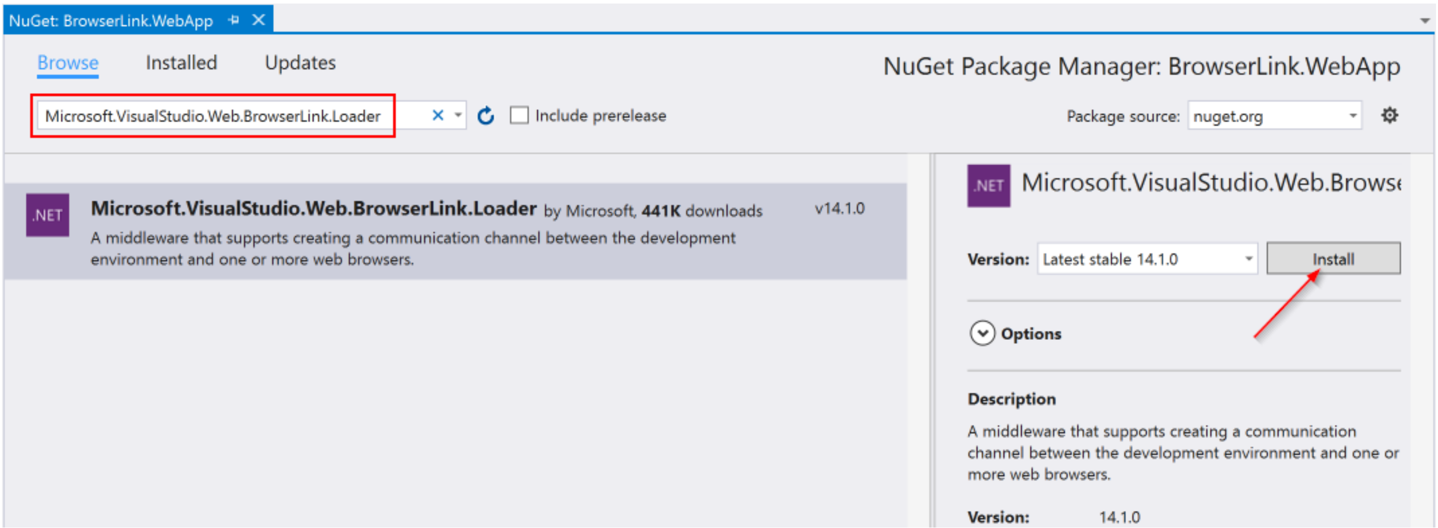
Since this is a Visual Studio feature, the easiest way to add the package is to open the **Package Manager Console** (**View > Other Windows > Package Manager Console**) and run the following command:

```
install-package Microsoft.VisualStudio.Web.BrowserLink.Loader
```

Alternatively, you can use **NuGet Package Manager**. Right-click the project name in **Solution Explorer**, and choose **Manage NuGet Packages**.



Then find and install the package.



Add configuration code

Open the *Startup.cs* file, and in the `Configure` method add the following code:

```
app.UseBrowserLink();
```

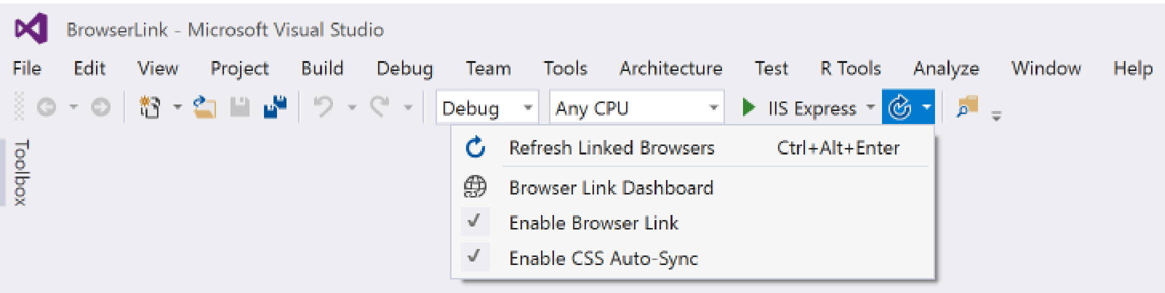
Usually that code is inside an `if` block that enables Browser Link only in the Development environment, as shown here:

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

For more information, see Working with Multiple Environments.

How to use Browser Link

When you have an ASP.NET Core project open, Visual Studio shows the Browser Link toolbar control next to the **Debug Target** toolbar control:



From the Browser Link toolbar control, you can:

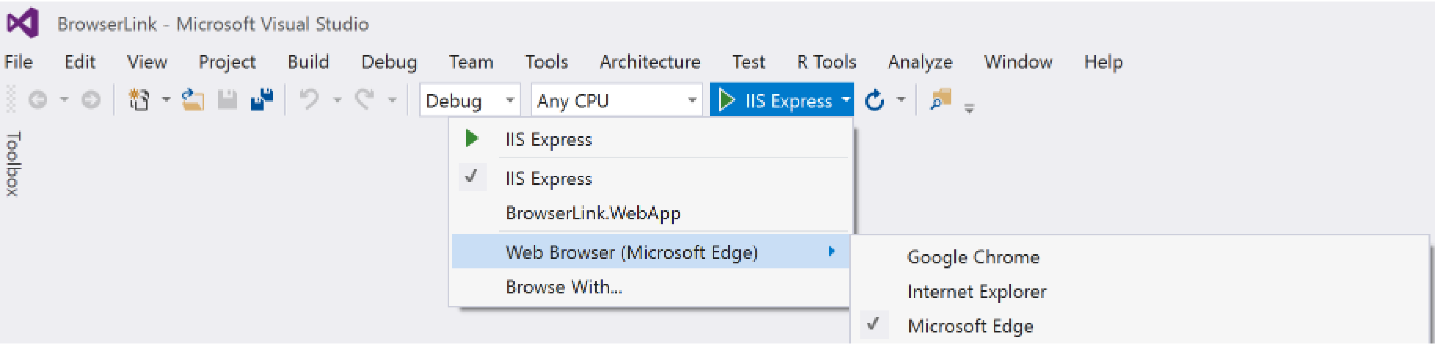
Refresh the web application in several browsers at once
Open the **Browser Link Dashboard**
Enable or disable **Browser Link**
Enable or disable CSS Auto-Sync

Note

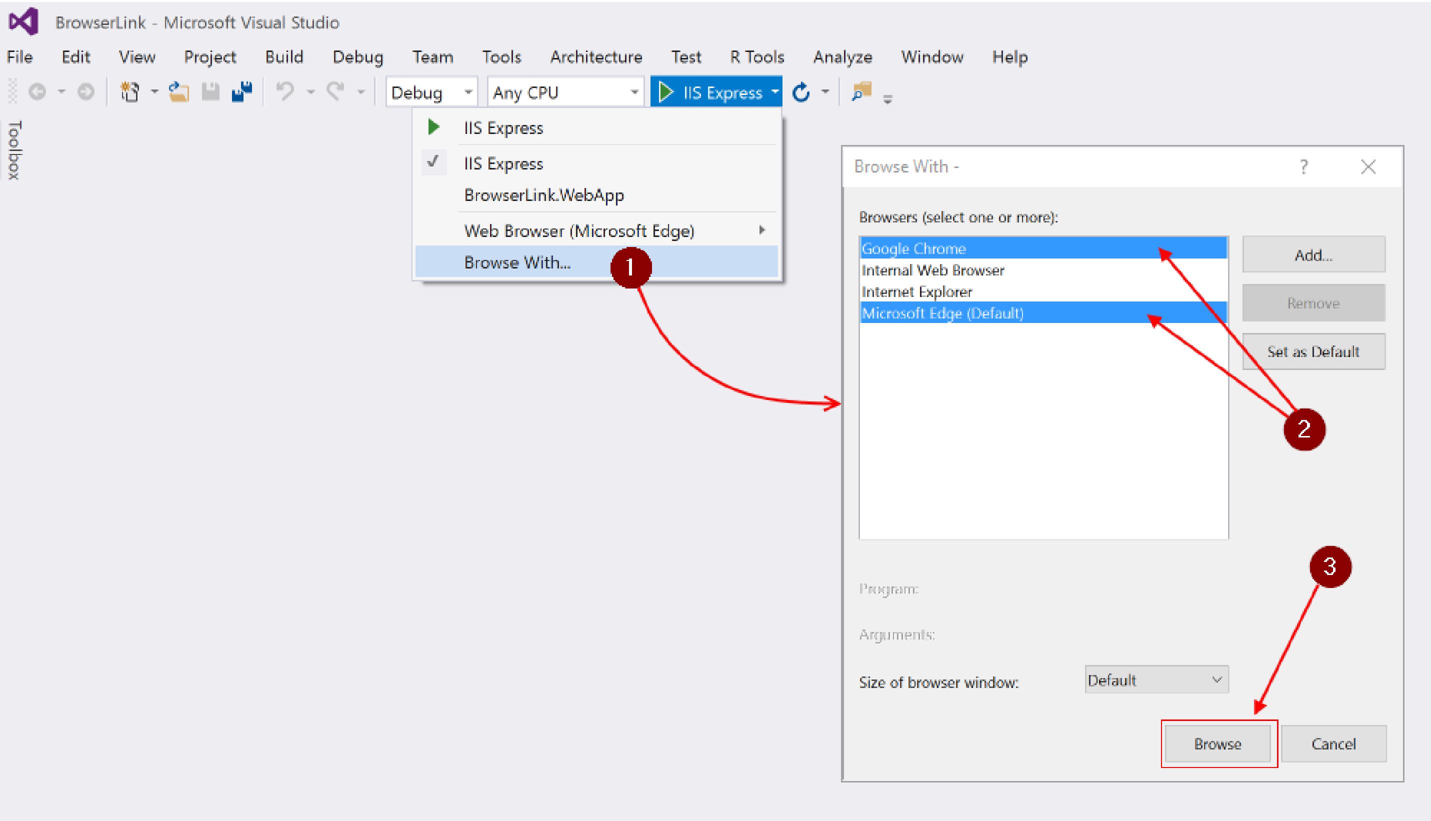
Some Visual Studio plug-ins, most notably *Web Extension Pack 2015* and *Web Extension Pack 2017*, offer extended functionality for Browser Link, but some of the additional features don't work with ASP.NET Core projects.

Refresh the web application in several browsers at once

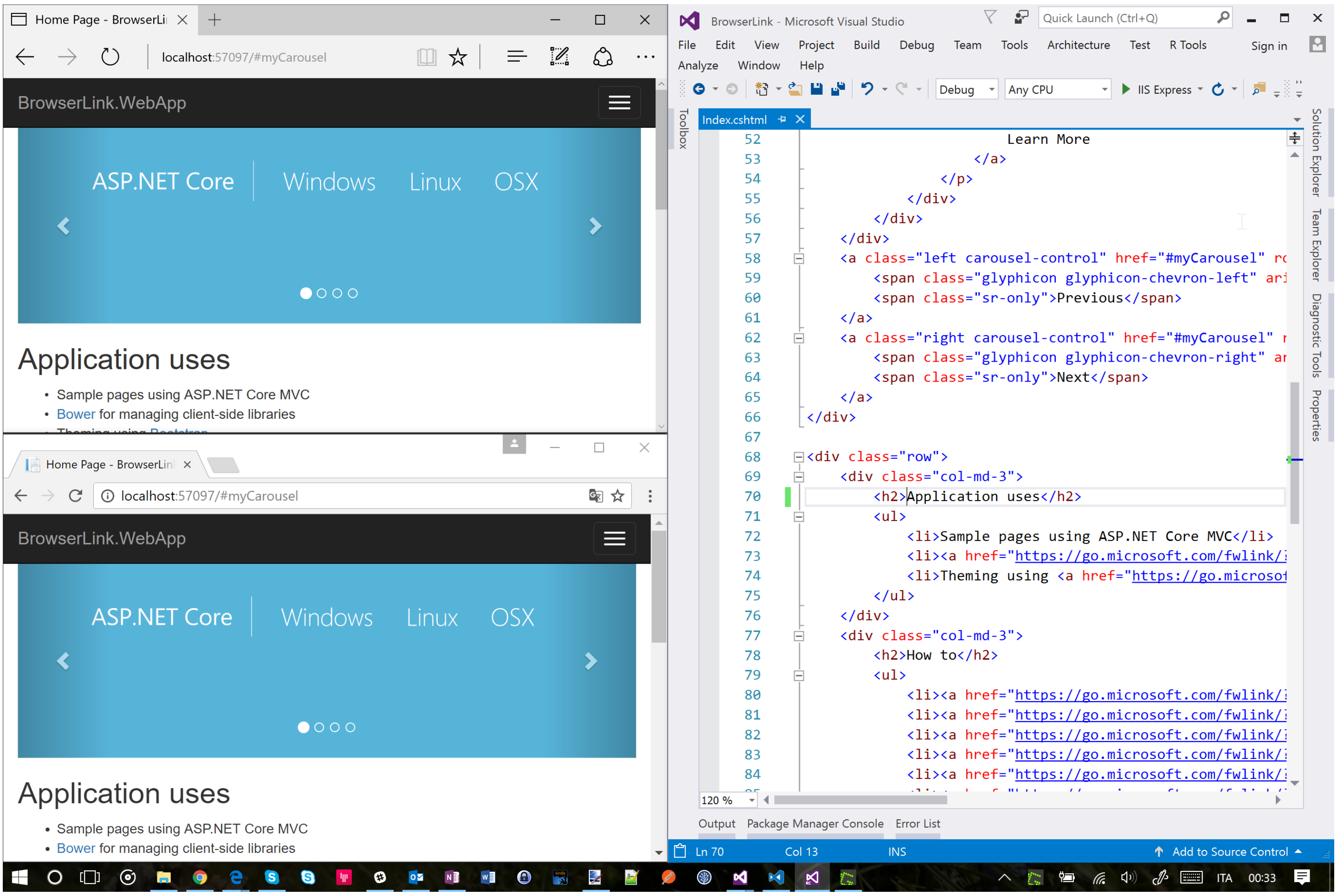
To choose a single web browser to launch when starting the project, use the drop-down menu in the **Debug Target** toolbar control:



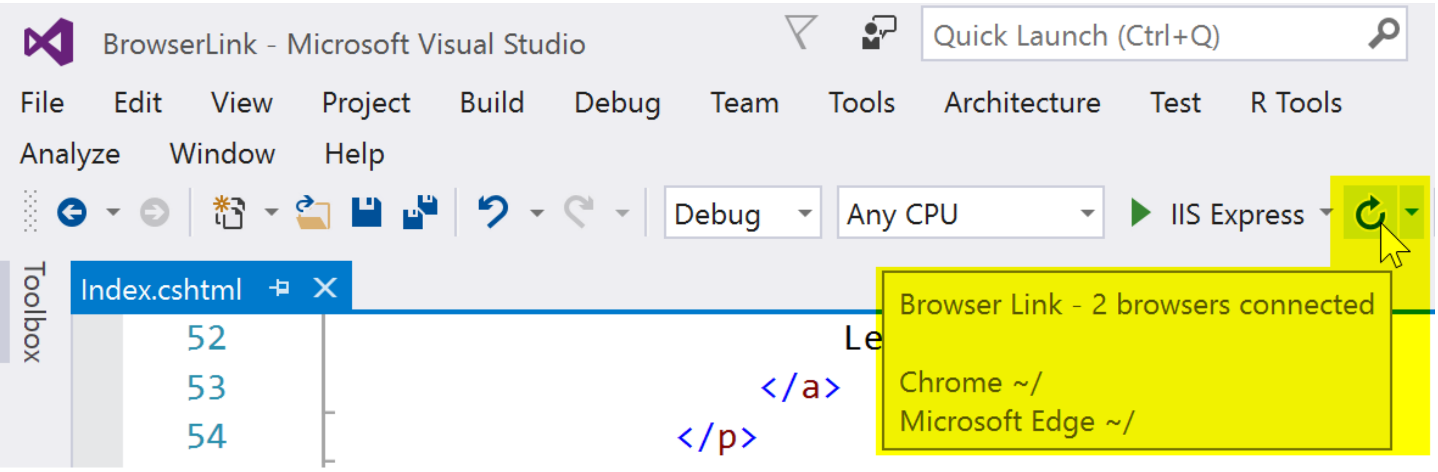
To open multiple browsers at once, choose **Browse with...** from the same drop-down. Hold down the CTRL key to select the browsers you want, and then click **Browse**:



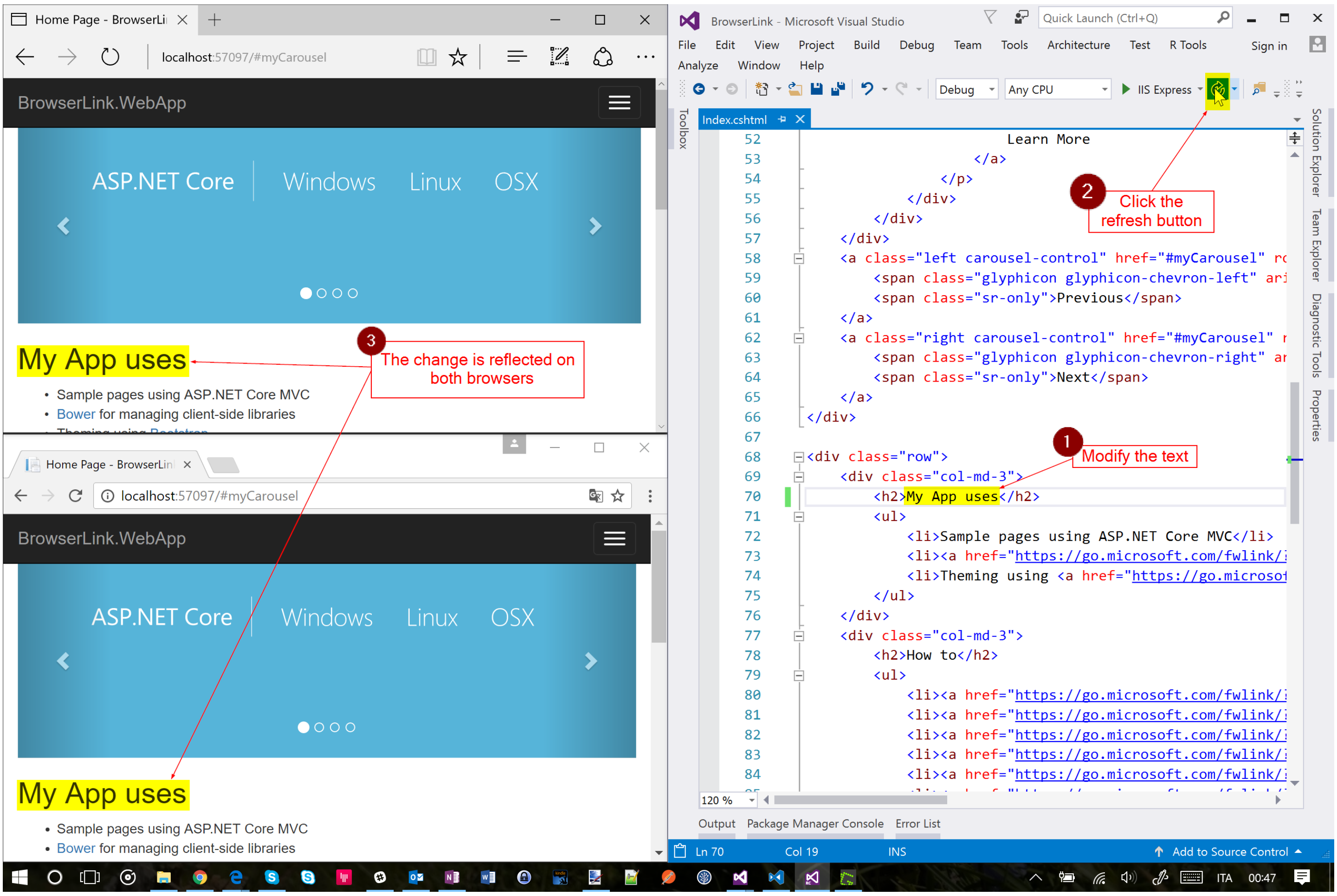
Here's a sample screenshot showing Visual Studio with the Index view open and two open browsers:



Hover over the Browser Link toolbar control to see the browsers that are connected to the project:



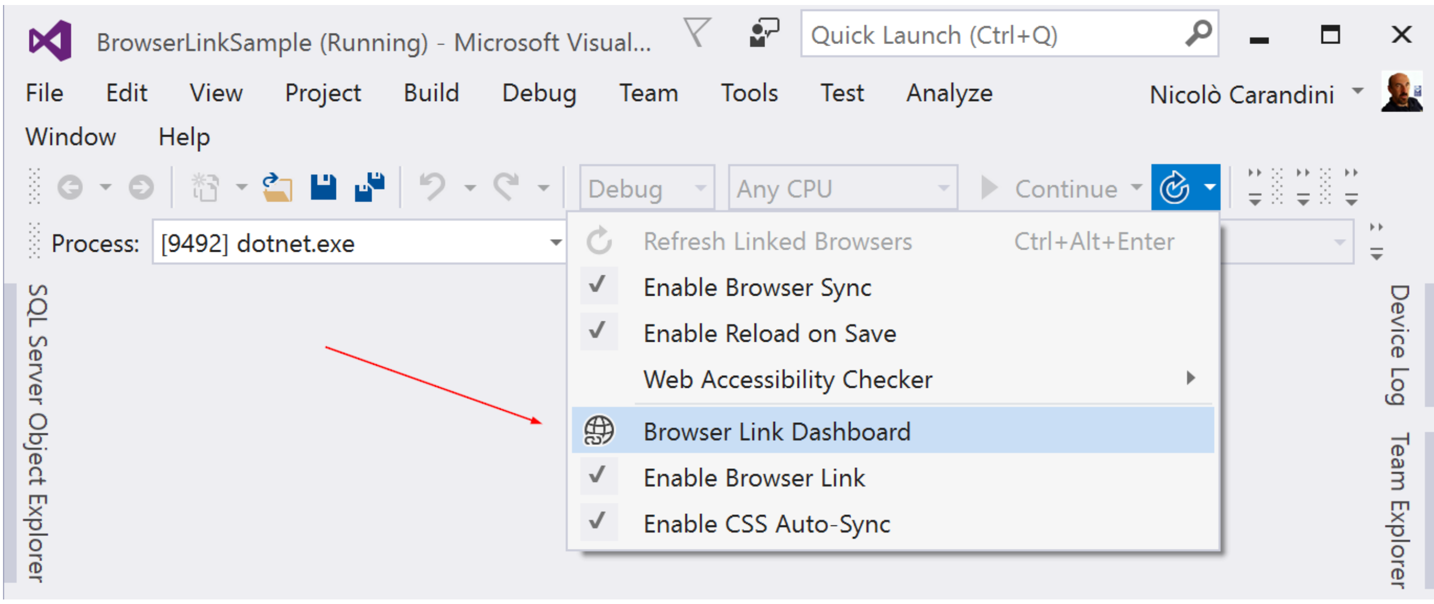
Change the Index view, and all connected browsers are updated when you click the Browser Link refresh button:



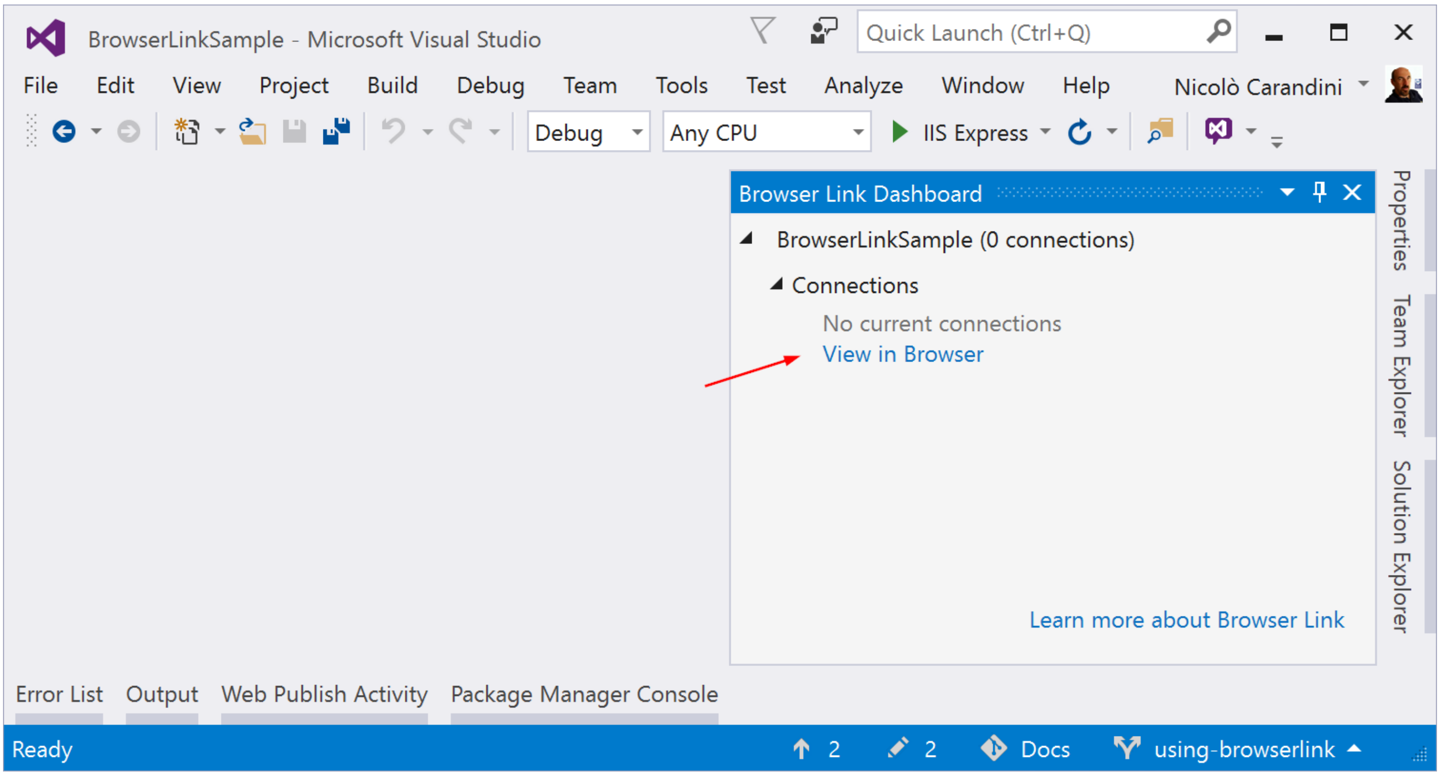
Browser Link also works with browsers that you launch from outside Visual Studio and navigate to the application URL.

The Browser Link Dashboard

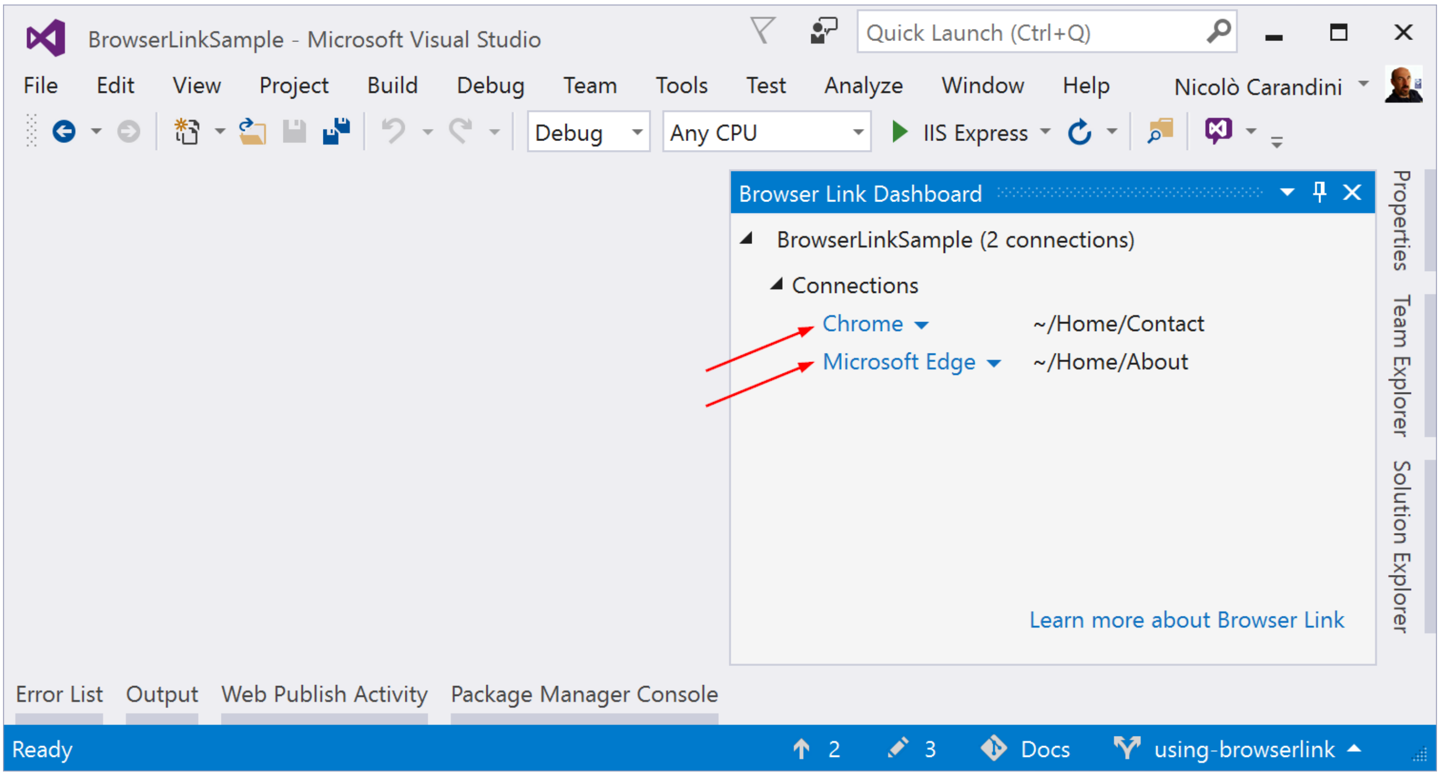
Open the Browser Link Dashboard from the Browser Link drop down menu to manage the connection with open browsers:



If no browser is connected, you can start a non debugging session clicking the *View in Browser* link:



Otherwise, the connected browsers are shown, with the path to the page that each browser is showing:



If you like, you can click on a listed browser name to refresh that single browser.

Enable or disable Browser Link

When you re-enable Browser Link after disabling it, you have to refresh the browsers to reconnect them.

Enable or disable CSS Auto-Sync

When CSS Auto-Sync is enabled, connected browsers are automatically refreshed when you make any change to CSS files.

How does it work?

Browser Link uses SignalR to create a communication channel between Visual Studio and the browser. When Browser Link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. Browser Link also registers a middleware component in the ASP.NET request pipeline. This component injects special `<script>` references into every page request from the server. You can see the script references by selecting **View source** in the browser and scrolling to the end of the `<body>` tag content:

Your source files are not modified. The middleware component injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports, without requiring any browser plug-in.

Creating Backend Services for Native Mobile Applications

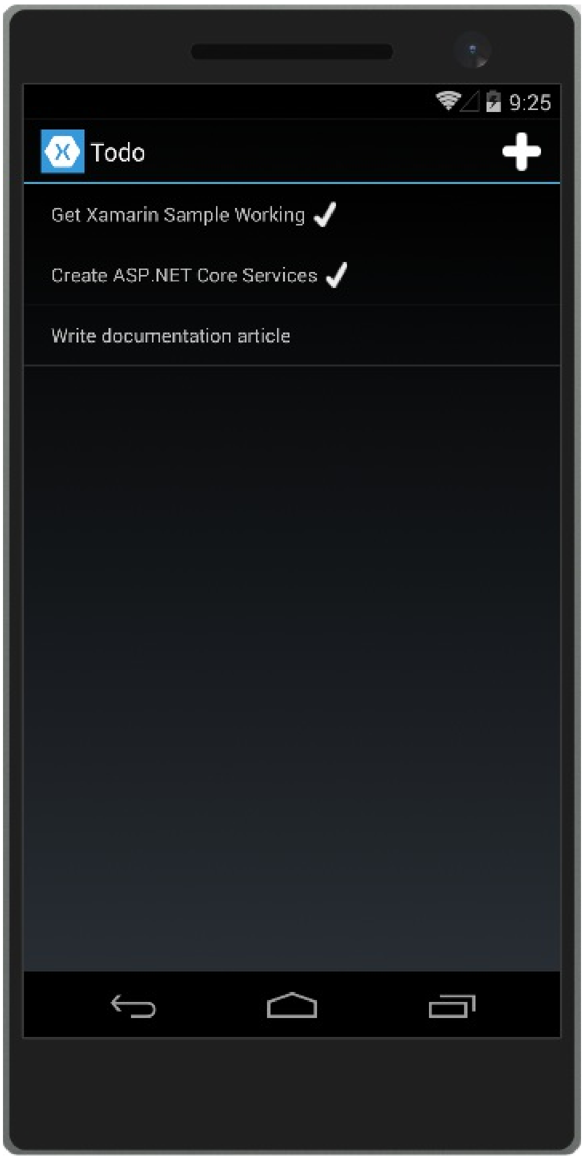
By Steve Smith

Mobile apps can easily communicate with ASP.NET Core backend services.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the Xamarin Forms ToDoRest app as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

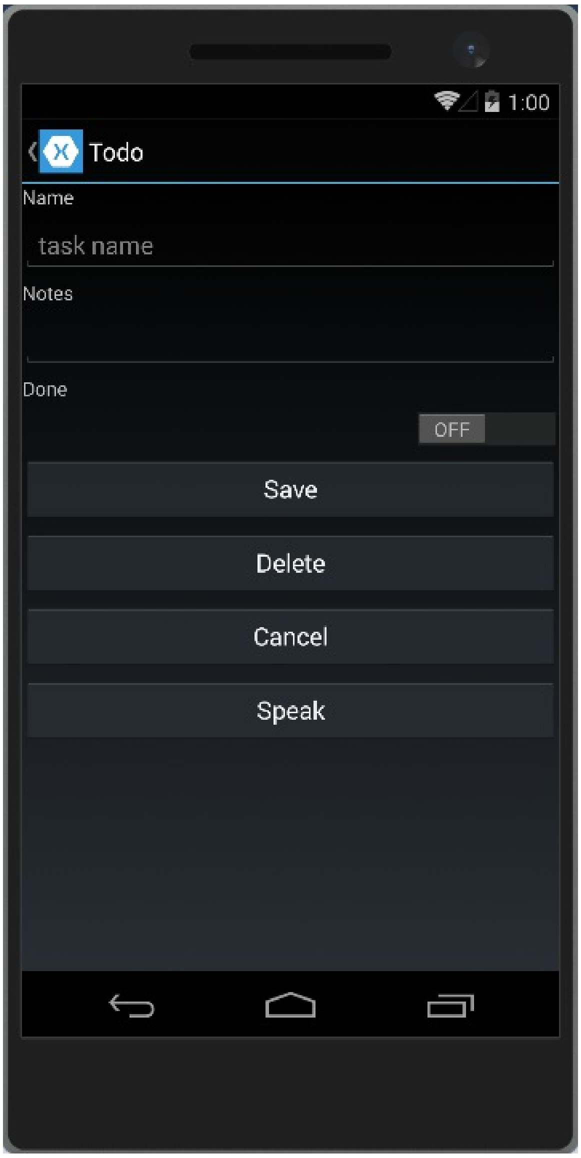


Features

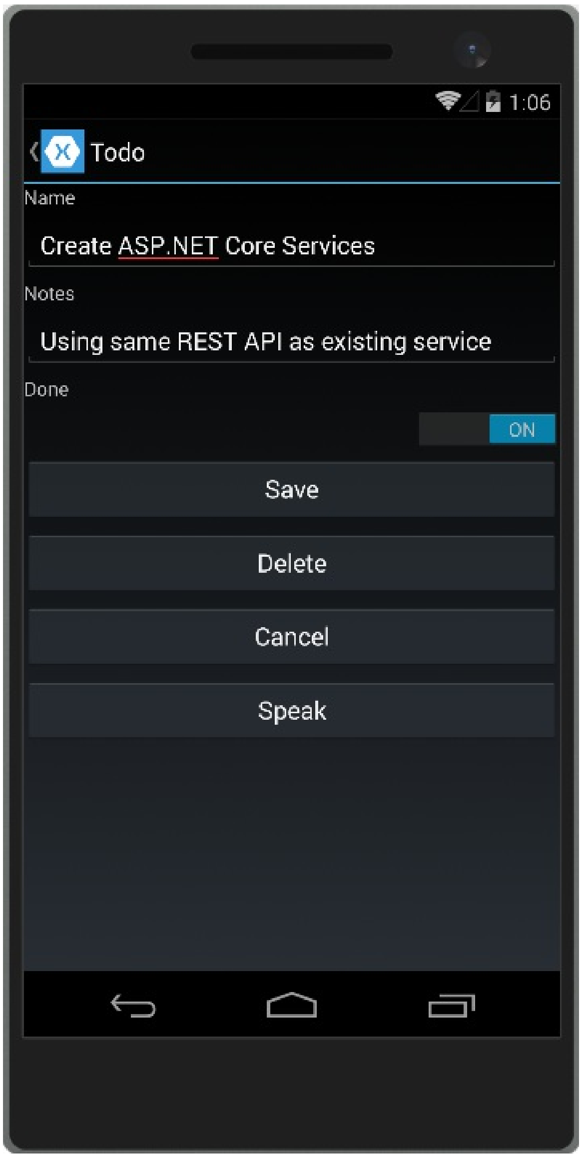
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it is done with a checkmark.

Tapping the `+` icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:

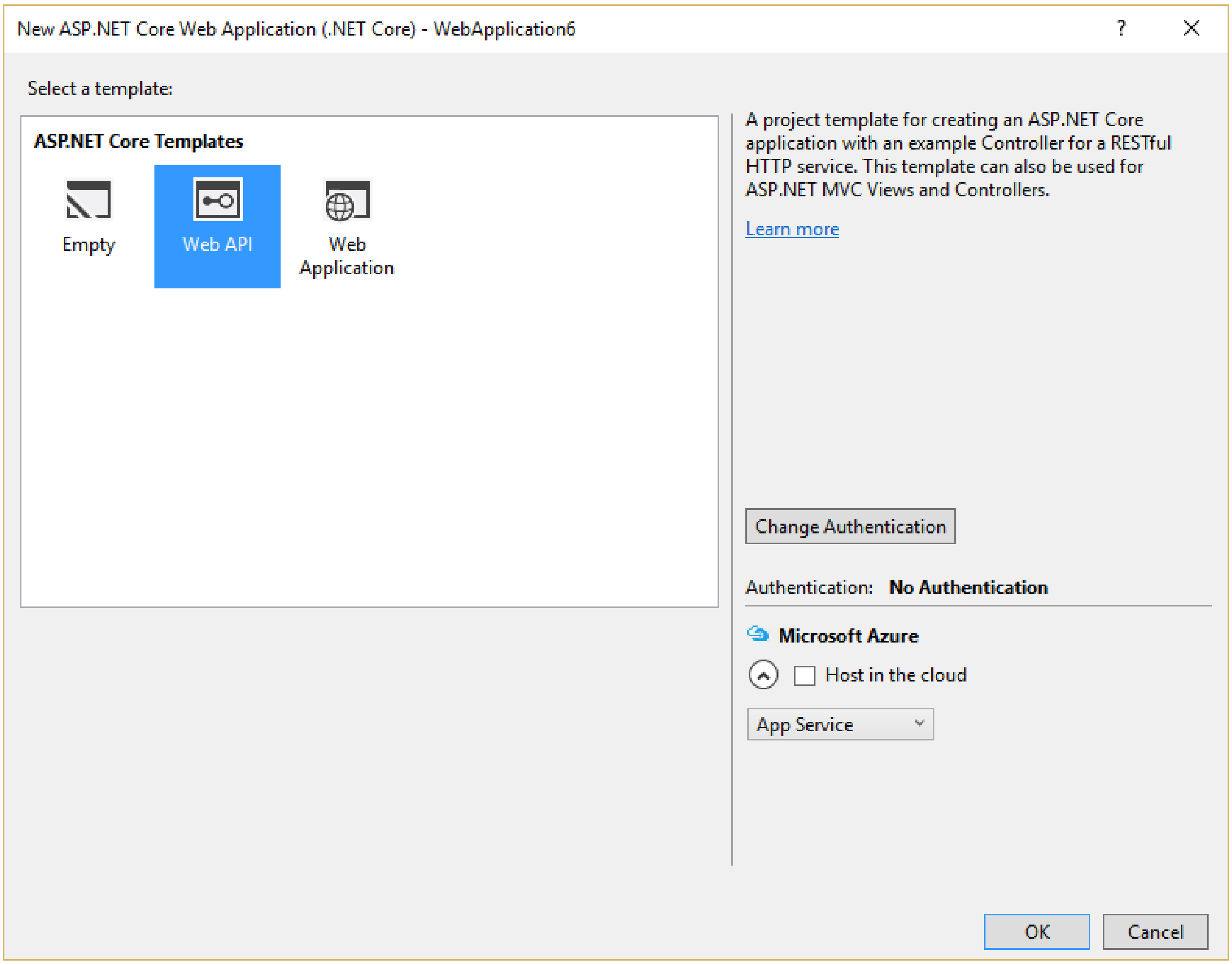


This sample is configured by default to use backend services hosted at developer.xamarin.com, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the *Constants.cs* file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// 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}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include `.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Note

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run `dotnet run` from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields using the `[Required]` attribute:

```
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; }
    }
}
```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```
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);
    }
}
```

For this sample, the implementation just uses a private collection of items:

```
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);
        }
    }
}
```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository,ToDoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

> 注 Tip
>
> Learn more about creating web APIs in Building Your First Web API with ASP.NET Core MVC and Visual Studio.

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from Microsoft.AspNetCore.Mvc.Controller. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `IToDoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for dependency injection.

```
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;
        }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

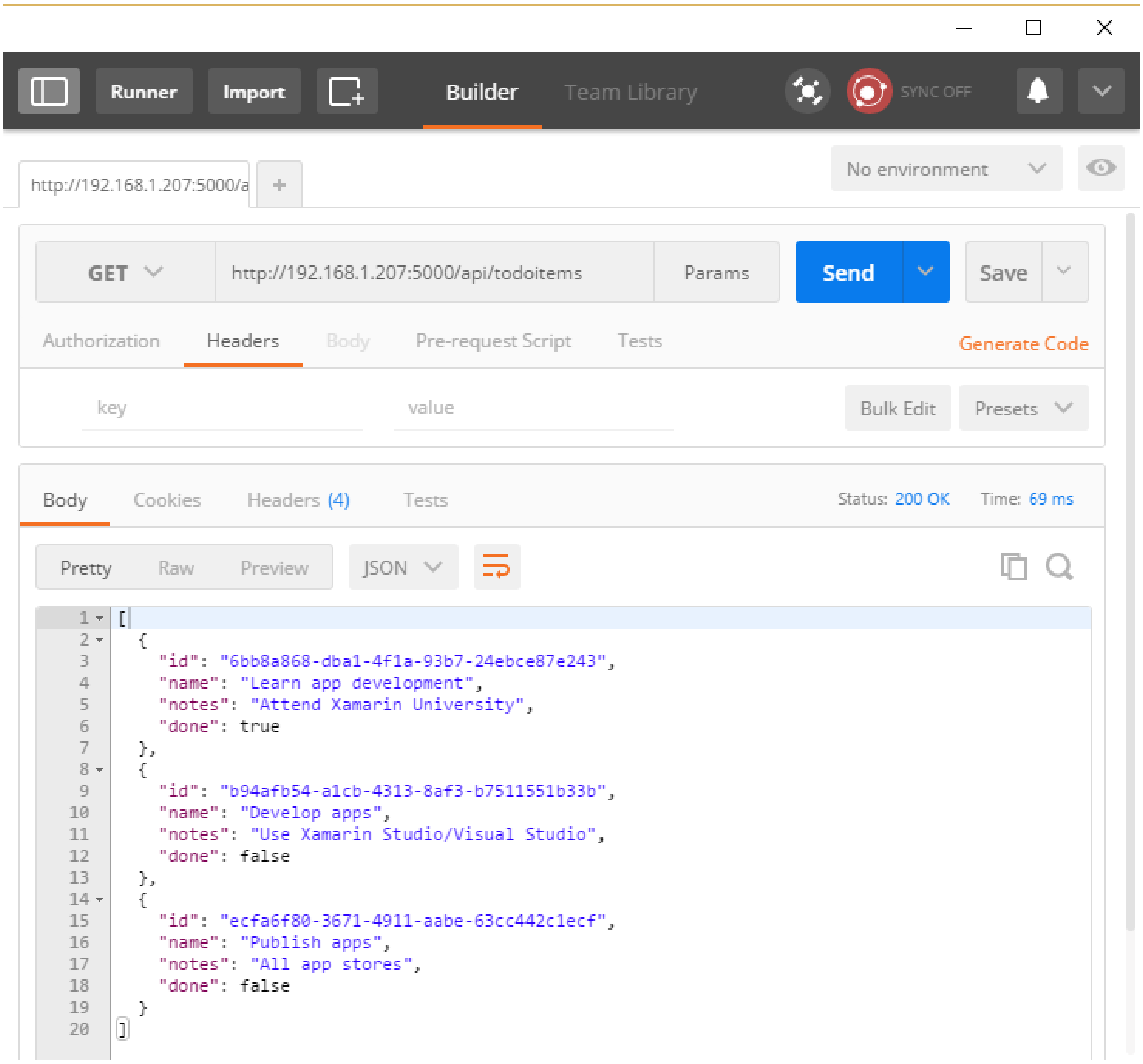
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_toDoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the ToDo items, serialized as JSON.

You can test your new API method using a variety of tools, such as Postman, shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it, and accepts an ID parameter and a `ToDoItem` instance. The HTTP verb attributes, like `[HttpPost]`, optionally accept a route template string (`{id}` in this example). This has the same effect as adding a `[Route]` attribute to the action. Since the `item` argument will be passed in the body of the POST, this parameter is decorated with the `[FromBody]` attribute.

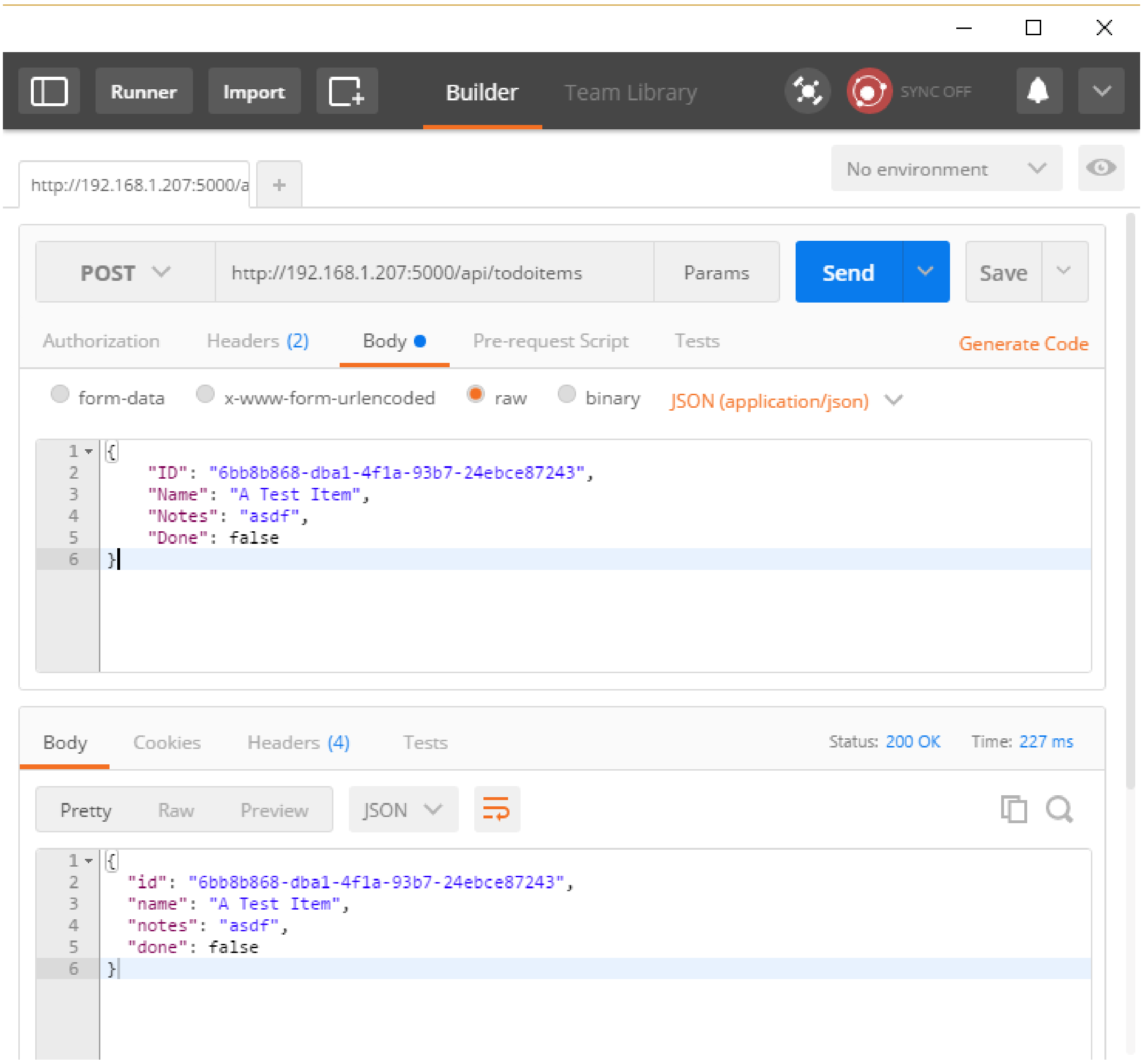
Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it is added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

```
[HttpPost("{id}")]
public IActionResult Create(string id, [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);
}
```

The sample uses an enum containing error codes that are passed to the mobile client:

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.



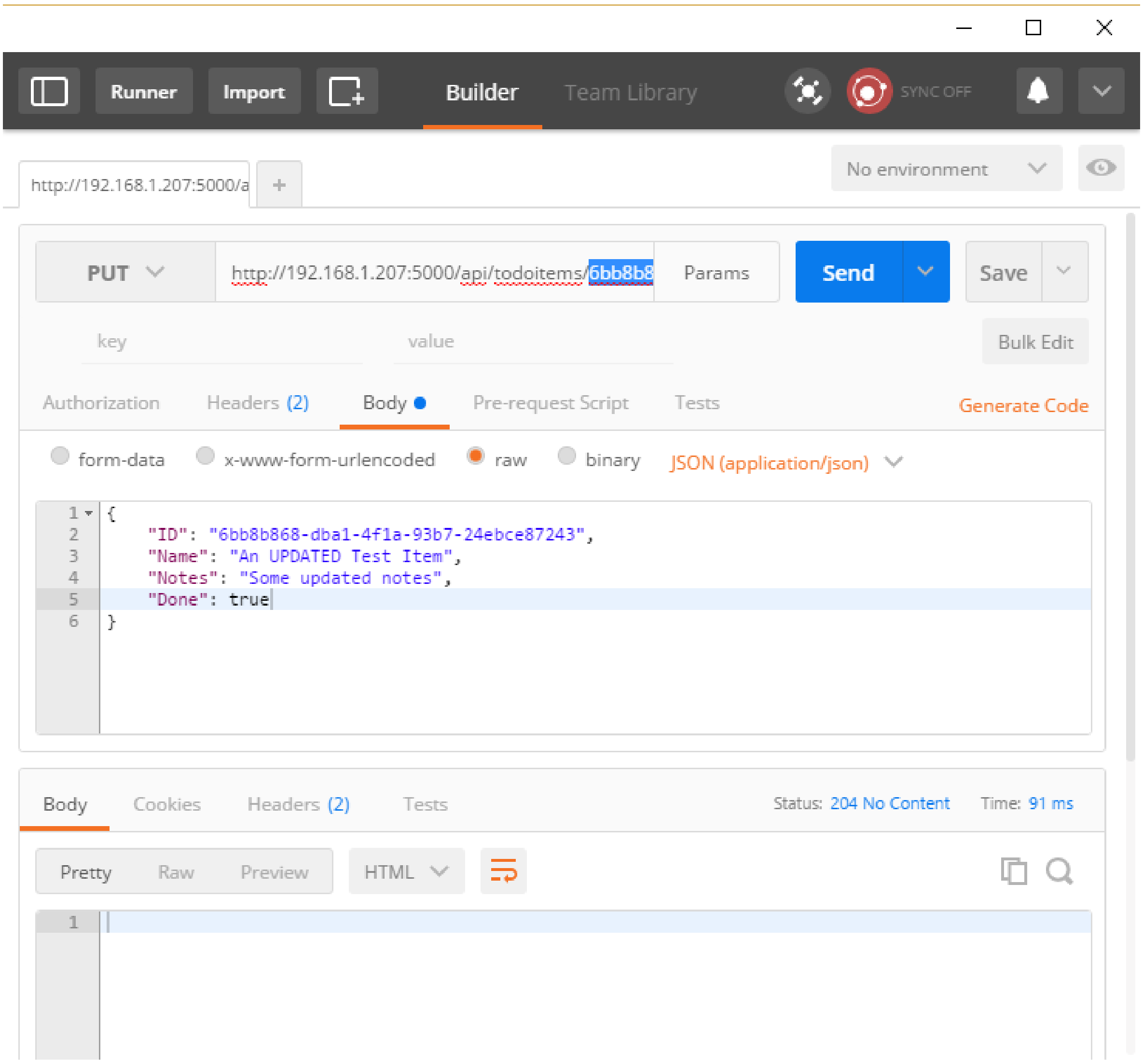
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut("{id}")]
public IActionResult Edit(string id, [FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _toDoRepository.Find(id);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT and add the ID of the record being updated to the URL. Specify the updated object data in the Body of the request.



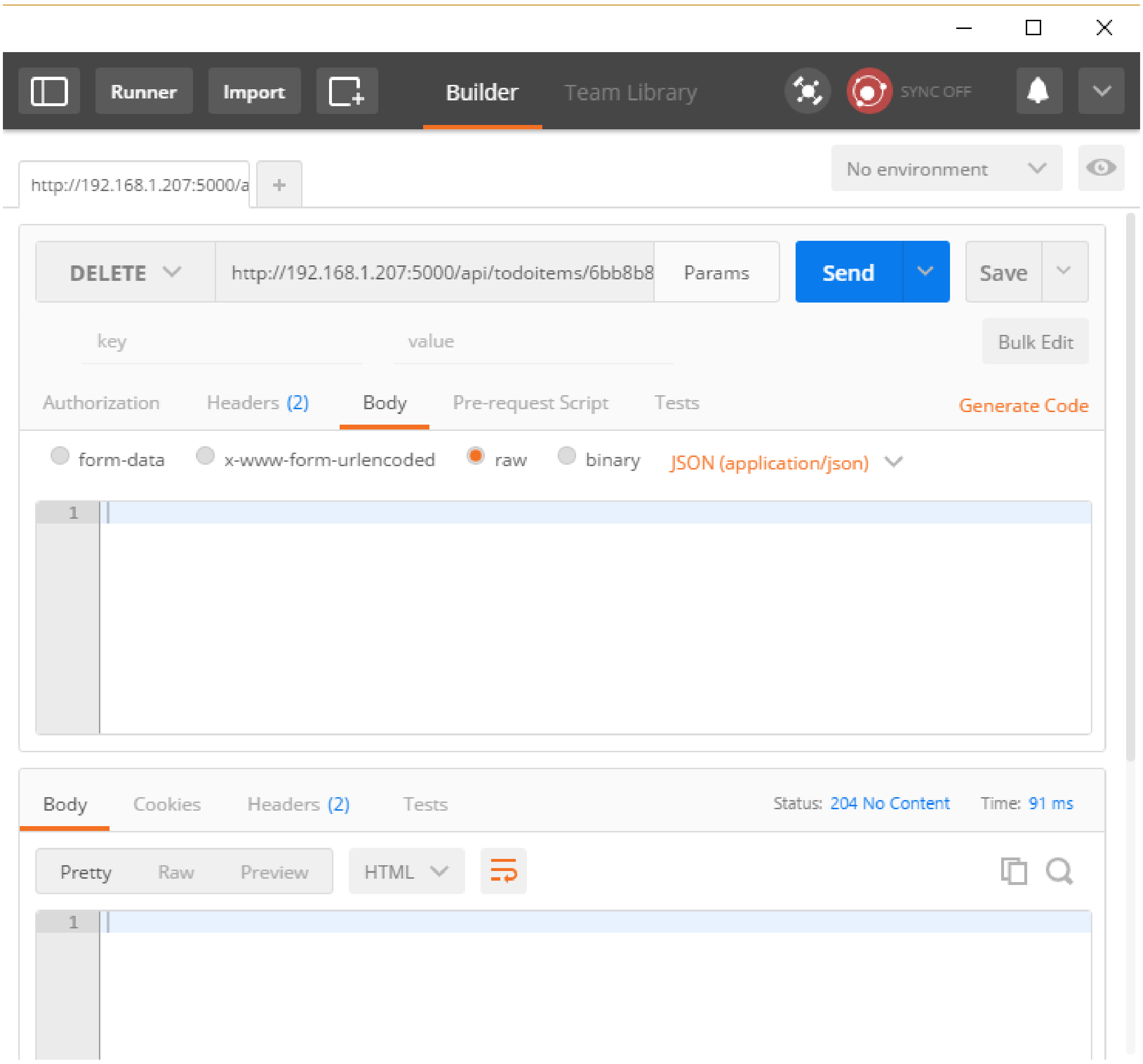
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[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();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Hosting and deployment overview for ASP.NET Core apps

Here are the main steps you perform to deploy an ASP.NET Core app to a hosting environment:

Publish the app to a folder on the hosting server.
Set up a process manager that starts the app when requests come in and restarts it after it crashes or the server reboots.
Set up a reverse proxy that forwards requests to the app.

Publish to a folder

The dotnet publish CLI command compiles application code and copies the files needed to run the application into a *publish* folder. When you deploy from Visual Studio the `dotnet publish` step is done for you automatically before files are copied to the deployment destination.

Folder contents

The *publish* folder contains *.exe* and *.dll* files for the application, its dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained* or *framework-dependent*. If the app is self-contained, the *.dll* files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files are not included because the app has a reference to a version of .NET that is installed on the computer. The default deployment model is framework-dependent. For more information, see .NET Core application deployment.

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see Directory structure.

Set up a process manager

An ASP.NET Core app is a console app that has to be started when a server boots and restarted after crashes. To automate starts and restarts you need a process manager. The most common process managers for ASP.NET Core are Nginx and Apache on Linux, and IIS and Windows Service on Windows.

Set up a reverse proxy

If your app uses the Kestrel web server and will be exposed to the Internet, you must use Nginx, Apache, or IIS as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel after some preliminary handling. The main reason for using a reverse proxy is security. For more information, see When to use Kestrel with a reverse proxy.

Using Visual Studio and MSBuild to automate deployment

Deployment often requires additional tasks besides copying the output from `dotnet publish` to a server. For example, you might want to include extra files in the *publish* folder, or exclude files from it. Visual Studio uses MSBuild for web deployment, and you can customize MSBuild to do many other tasks during deployment. For more information, see Publish profiles in Visual Studio and the Using MSBuild and Team Foundation Build book.

You can deploy directly from Visual Studio to Azure App Service by using the Publish Web feature or by using built-in Git support. Visual Studio Team Services supports continuous deployment to Azure App Service.

Additional resources

For information about using Docker as a hosting environment, see Host ASP.NET Core apps in Docker.

Set up a hosting environment for ASP.NET Core on Windows with IIS, and deploy to it

By Luke Latham and Rick Anderson

Supported operating systems

The following operating systems are supported:

Windows 7 and newer

Windows Server 2008 R2 and newer†

†Conceptually, the IIS configuration described in this document also applies to hosting ASP.NET Core applications on Nano Server IIS, but refer to ASP.NET Core with IIS on Nano Server for specific instructions.

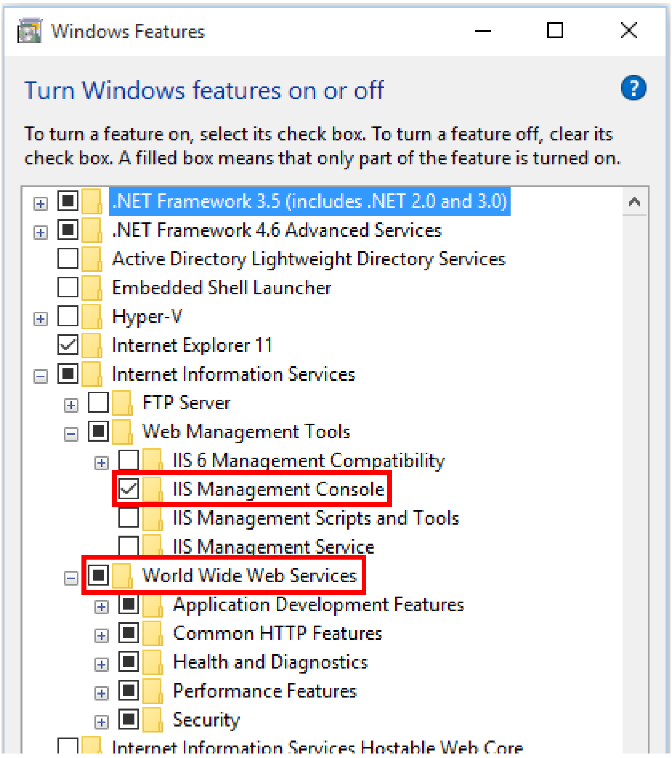
WebListener server will not work in a reverse-proxy configuration with IIS. You must use the Kestrel server.

IIS configuration

Enable the **Web Server (IIS)** role and establish role services.

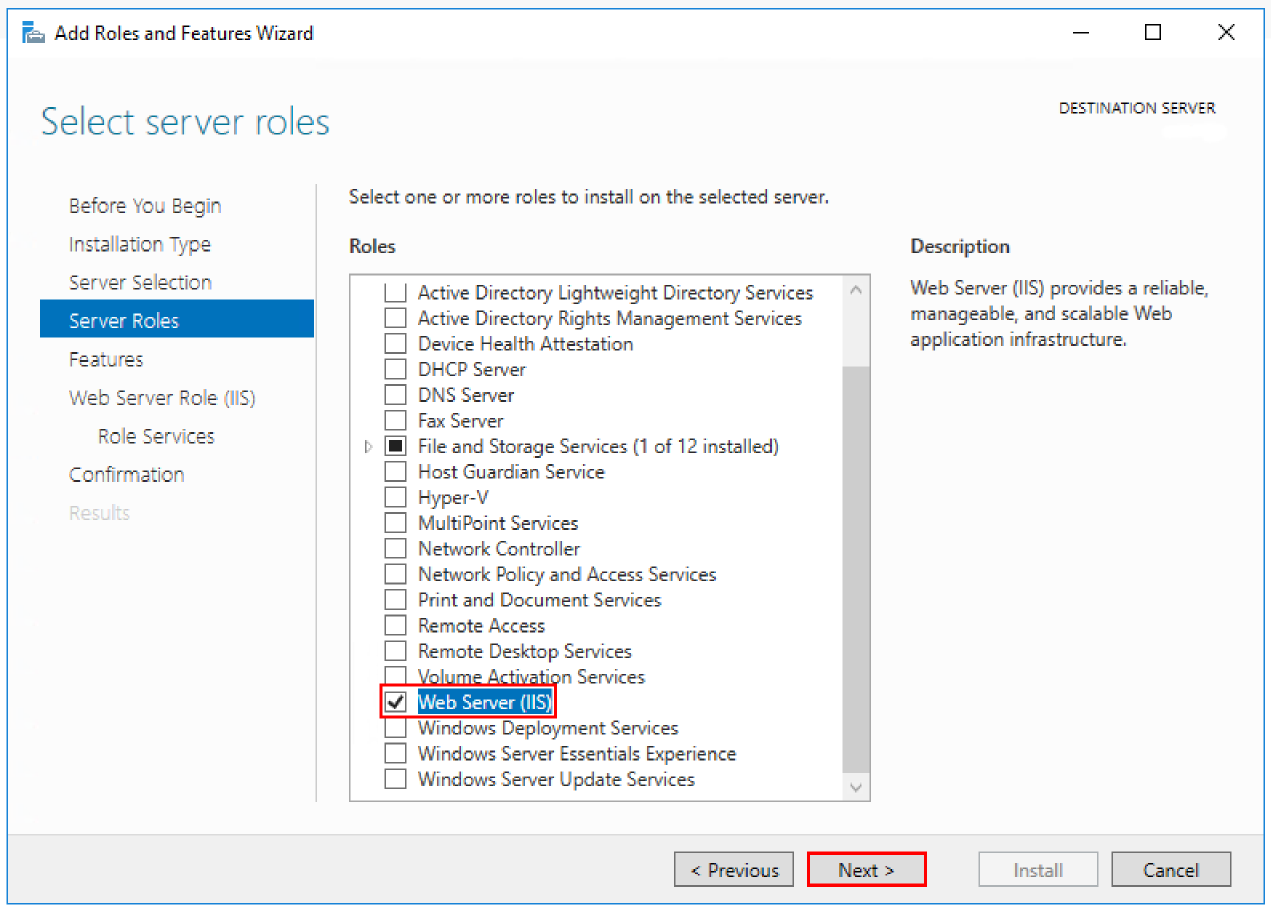
Windows desktop operating systems

Navigate to **Control Panel > Programs > Programs and Features > Turn Windows features on or off** (left side of the screen). Open the group for **Internet Information Services** and **Web Management Tools**. Check the box for **IIS Management Console**. Check the box for **World Wide Web Services**. Accept the default features for **World Wide Web Services** or customize the IIS features to suit your needs.

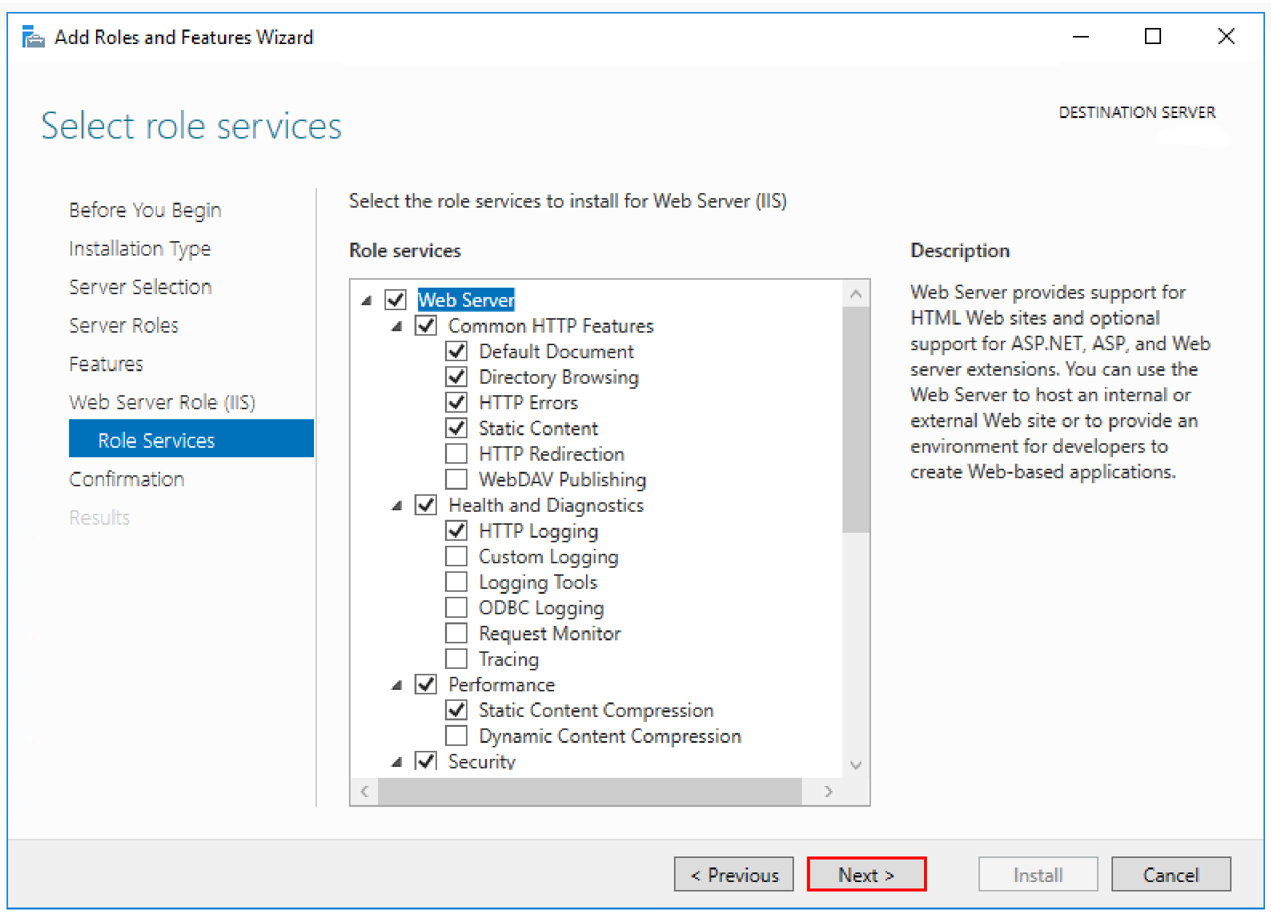


Windows Server operating systems

For server operating systems, use the **Add Roles and Features** wizard via the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



On the **Role services** step, select the IIS role services you desire or accept the default role services provided.



Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart is not required after installing the Web Server (IIS) role.

Install the .NET Core Windows Server Hosting bundle

Install the .NET Core Windows Server Hosting bundle on the hosting system. The bundle will install the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module creates the reverse-proxy between IIS and the Kestrel server. Note: If the system doesn't have an Internet connection, obtain and install the *Microsoft Visual C++ 2015 Redistributable* before installing the .NET Core Windows Server Hosting bundle.

Restart the system or execute **net stop was /y** followed by **net start w3svc** from a command prompt to pick up a change to the system PATH.

▯ Note

If you only plan to host self-contained deployments and thus don't require the .NET Core runtime on the system, you have the option of only installing the ASP.NET Core Module by running the installer from an Administrator command prompt: **DotNetCore.1.0.3_1.1.0-WindowsHosting.exe OPT_INSTALL_LTS_REDIST=0 OPT_INSTALL_FTS_REDIST=0**

▯ Note

If you use an IIS Shared Configuration, see ASP.NET Core Module with IIS Shared Configuration.

Install Web Deploy when publishing with Visual Studio

If you intend to deploy your applications with Web Deploy in Visual Studio, install the latest version of Web Deploy on the hosting system. To install Web Deploy, you can use the Web Platform Installer (WebPI) or obtain an installer directly from the Microsoft Download Center. The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

Application configuration

Enabling the IISIntegration components

Include a dependency on the *Microsoft.AspNetCore.Server.IISIntegration* package in the application dependencies. Incorporate IIS Integration middleware into the application by adding the *.UseIISIntegration()* extension method to *WebHostBuilder()*. Note that code calling *.UseIISIntegration()* does not affect code portability.

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

Setting IISOptions for the IISIntegration service

To configure *IISIntegration* service options, include a service configuration for *IISOptions* in *ConfigureServices*.

```
services.Configure<IISOptions>(options => {
  ...
});
```

OPTION	SETTING
AutomaticAuthentication	If true, the authentication middleware will alter the request user arriving and respond to generic challenges. If false,the authentication middleware will only provide identity and respond to challenges when explicitly indicated by theAuthenticationScheme

OPTION	SETTING
ForwardClientCertificate	If true and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `ITLSConnectionFeature` will be populated.
ForwardWindowsAuthentication	If true, authentication middleware will attempt to authenticate using platform handler windows authentication. If false, authentication middleware won't be added.

web.config

The *web.config* file configures the ASP.NET Core Module and provides other IIS configuration. Creating, transforming, and publishing *web.config* is handled by `Microsoft.NET.Sdk.Web`, which is included when you set your project's SDK at the top of your *.csproj* file, `<Project Sdk="Microsoft.NET.Sdk.Web">`.

If you don't have a *web.config* file in the project when you publish with *dotnet publish* or with Visual Studio publish, the file is created for you in published output. If you have the file in your project, it's transformed with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output. The transformation doesn't touch IIS configuration settings that you've included in the file.

Deploy the application

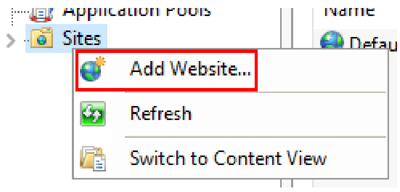
On the target IIS system, create a folder to contain the application's published folders and files, which are described in Directory Structure.

Within the folder you created, create a *logs* folder to hold application logs (if you plan to enable logging). If you plan to deploy your application with a *logs* folder in the payload, you may skip this step.

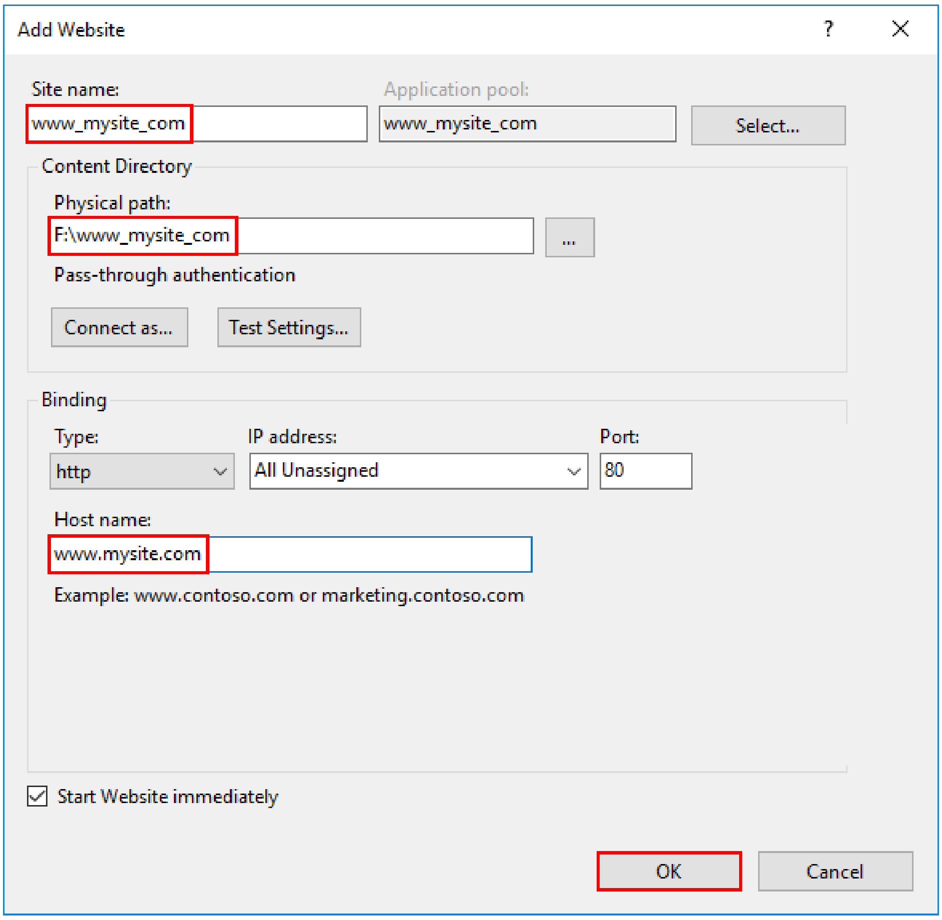
In **IIS Manager**, create a new website. Provide a **Site name** and set the **Physical path** to the application's deployment folder that you created. Provide the **Binding** configuration and create the website.

Set the application pool to **No Managed Code**. ASP.NET Core runs in a separate process and manages the runtime.

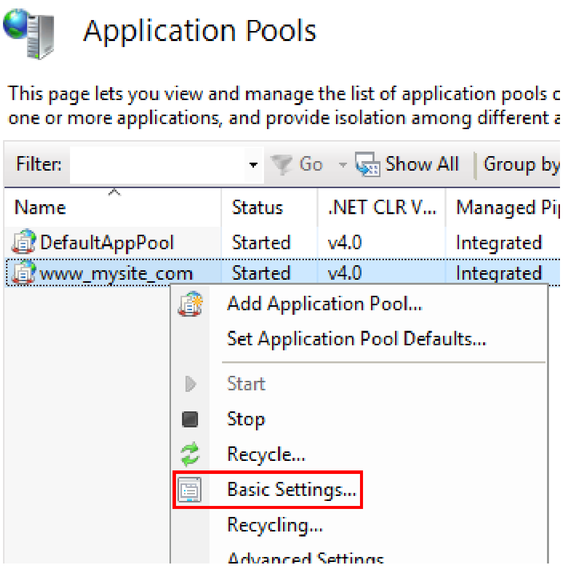
Open the **Add Website** window.



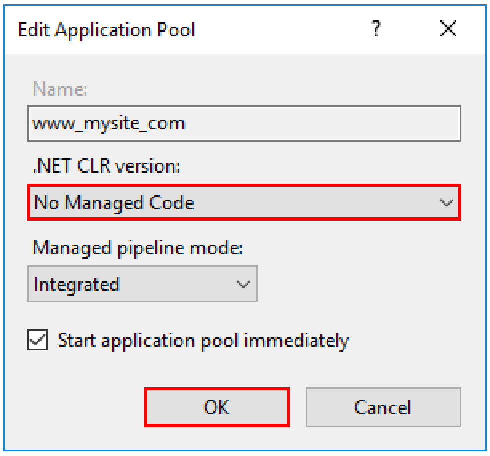
Configure the website.



In the **Application Pools** panel, open the **Edit Application Pool** window by right-clicking on the website's application pool and selecting **Basic Settings...** from the popup menu.



Set the **.NET CLR version** to **No Managed Code**.



Note

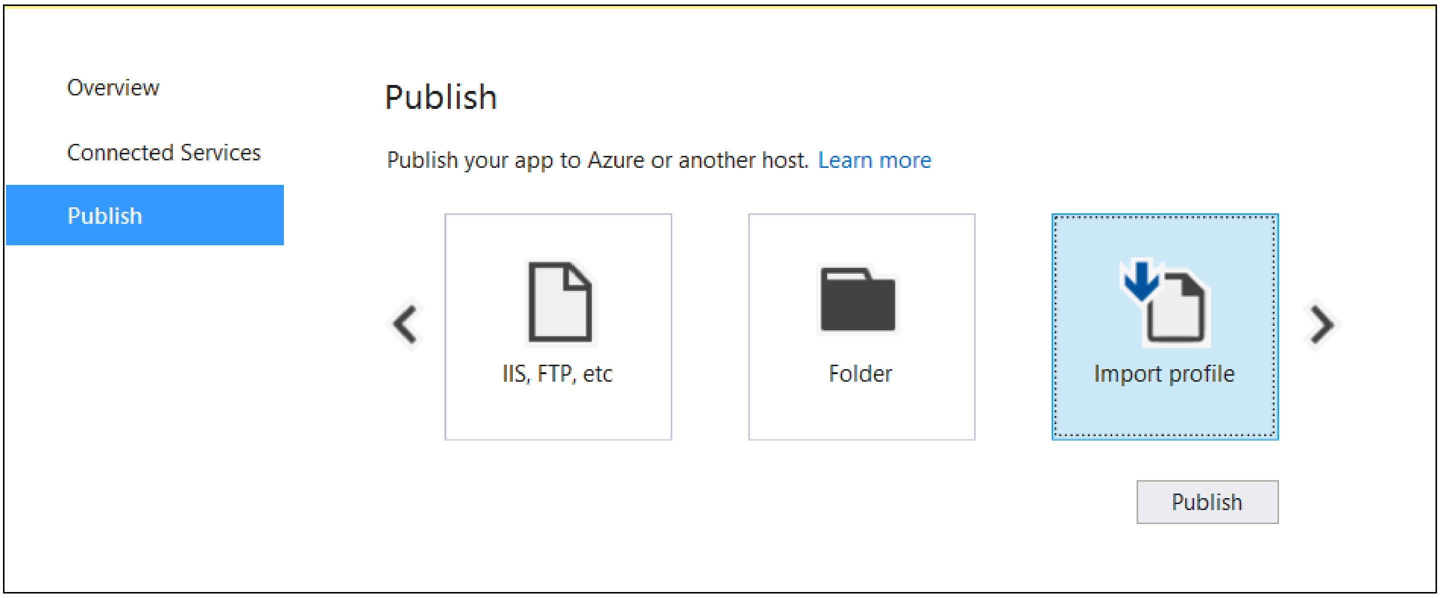
If you change the default identity of the application pool from **ApplicationPoolIdentity**, verify the new identity has the required permissions to access the application's folder and database.

Deploy the application

Deploy the application to the folder you created on the target IIS system. Web Deploy is the recommended mechanism for deployment. Alternatives to Web Deploy are listed below.

Web Deploy with Visual Studio

Create a Publish Profile in Visual Studio and click the **Publish** button to deploy your application. If your hosting provider supplies a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog.



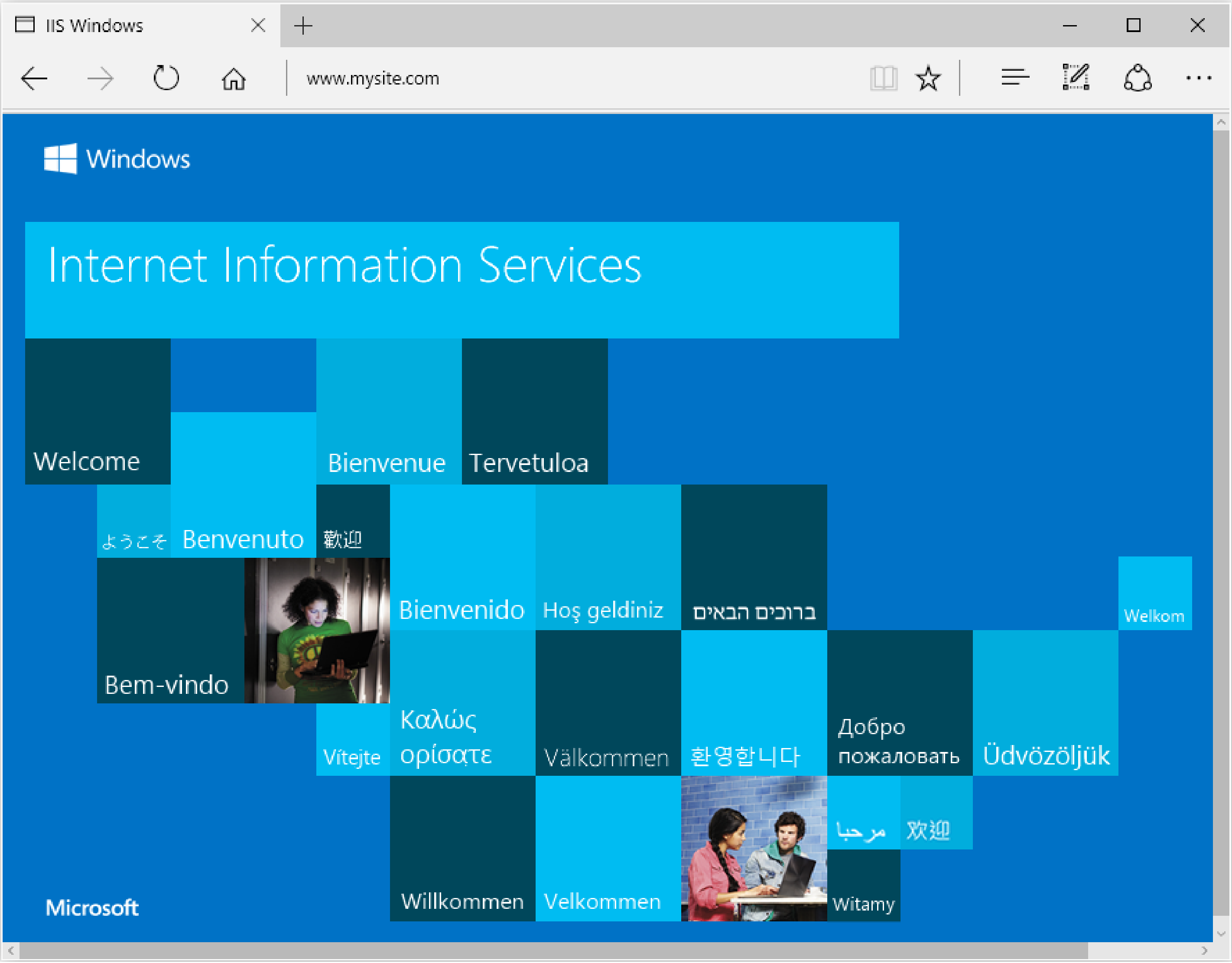
Web Deploy outside of Visual Studio

You can also use Web Deploy outside of Visual Studio from the command line. For more information, see Web Deployment Tool.

Alternatives to Web Deploy

If you don't wish to use Web Deploy or are not using Visual Studio, you may use any of several methods to move the application to the hosting system, such as Xcopy, Robocopy, or PowerShell. Visual Studio users may use the Publish Samples.

Browse the website



 Warning

.NET Core applications are hosted via a reverse-proxy between IIS and the Kestrel server. In order to create the reverse-proxy, the *web.config* file must be present at the content root path (typically the app base path) of the deployed application, which is the website physical path provided to IIS. Sensitive files exist on the app's physical path, including subfolders, such as *my_application.runtimeconfig.json*, *my_application.xml* (XML Documentation comments), and *my_application.deps.json*. The *web.config* file is required to create the reverse proxy to Kestrel, which prevents IIS from serving these and other sensitive files. **Therefore, it is important that the *web.config* file is never accidently renamed or removed from the deployment.**

Data protection

An ASP.NET Core application will store the keyring in memory under the following condition:

A website is hosted behind IIS.
The Data Protection stack has not been configured to store the keyring in a persistent store.

If the keyring is stored in memory, when the app restarts:

All forms authentication tokens will be invalid.
Users will need to login again on their next request.
Any data you protected with the keyring will no longer be unprotected.

 Warning

Data Protection is used by several ASP.NET middlewares, including those used in authentication. Even if you do not specifically call any Data Protection APIs from your own code you should configure Data Protection with a deployment script or in your own code. If you do not configure data protection, by default the keys will be held in memory and discarded when your app restarts. Restarting will invalidate any cookies written by the cookie authentication and users will have to login again.

To configure Data Protection under IIS you must use one of the following approaches:

Run a powershell script to create suitable registry entries (For example, `.\Provision-AutoGenKeys.ps1 DefaultAppPool`). This will store keys in the registry, protected using DPAPI with a machine wide key.
Configure the IIS Application Pool to load the user profile. This setting is in the **Process Model** section under the **Advanced Settings** for the application pool. Set **Load User Profile** to `True`. This will store keys under the user profile directory, and protected using DPAPI with a key specific to the user account used for the app pool.
Adjust your application code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure it is a trusted certificate. For example, if it is a self signed certificate you must place it in the Trusted Root store.

When using IIS in a web farm:

Use a file share all machines can access.
Deploy an X509 certificate to each machine. Configure data protection in code.

1. Create a Data Protection Registry Hive

Data Protection keys used by ASP.NET applications are stored in registry hives external to the applications. To persist the keys for a given application, you must create a registry hive for the application's application pool.

For standalone IIS installations, you may use the Data Protection Provision-AutoGenKeys.ps1 PowerShell script for each application pool used with an ASP.NET Core application. This script will create a special registry key in the HKLM registry that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

In web farm scenarios, an application can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys are not encrypted. You should ensure that the file permissions for such a share are limited to the Windows account the application runs as. In addition you may choose to protect keys at rest using an X509 certificate. You may wish to consider a mechanism to allow users to upload certificates, place them into the user's trusted certificate store, and ensure they are available on all machines the user's application will run on. See Configuring Data Protection for details.

2. Configure the IIS Application Pool to load the user profile

This setting is in the Process Model section under the Advanced Settings for the application pool. Set Load User Profile to True. This will store keys under the user profile directory, and protected using DPAPI with a key specific to the user account used for the app pool.

3. Machine-wide policy for data protection

The data protection system has limited support for setting default machine-wide policy for all applications that consume the data protection APIs. See the data protection documentation for more details.

Configuration of sub-applications

When adding applications to an IIS Site's root application, the root application *web.config* file should include the `<handlers>` section, which adds the ASP.NET Core Module as a handler for the app. Applications added to the root application shouldn't include the `<handlers>` section. If you repeat the `<handlers>` section in a sub-application's *web.config* file, you will receive a 500.19 (Internal Server Error) referencing the faulty config file when you attempt to browse the sub-application.

Configuration of IIS with web.config

IIS configuration is still influenced by the `<system.webServer>` section of *web.config* for those IIS features that apply to a reverse proxy configuration. For example, you may have IIS configured at the system level to use dynamic compression, but you could disable that setting for an app with the `<urlCompression>` element in the app's *web.config* file. For more information, see the configuration reference for `<system.webServer>`, ASP.NET Core Module Configuration Reference, and Using IIS Modules with

ASP.NET Core.

Configuration sections of web.config

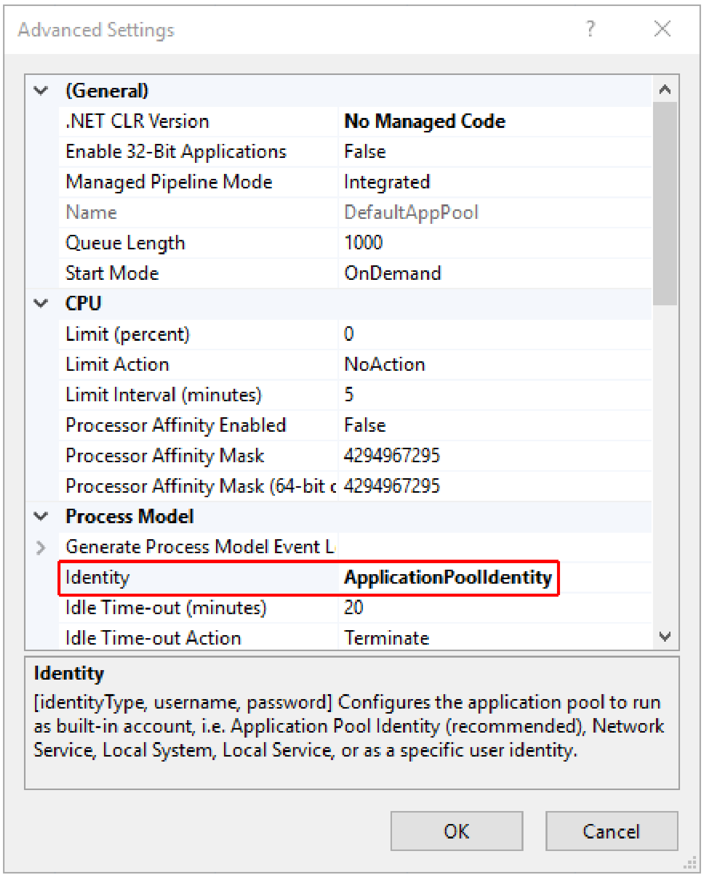
Unlike .NET Framework applications that are configured with the `<system.web>`, `<appSettings>`, `<connectionStrings>`, and `<location>` elements in *web.config*, ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Application Pools

When hosting multiple websites on a single system, you should isolate the applications from each other by running each app in its own application pool. The IIS **Add Website** dialog defaults to this behavior. When you provide a **Site name**, the text is automatically transferred to the **Application pool** textbox. A new application pool will be created using the site name when you add the website.

Application Pool Identity

An application pool identity account allows you to run an application under a unique account without having to create and manage domains or local accounts. On IIS 8.0+, the IIS Admin Worker Process (WAS) will create a virtual account with the name of the new application pool and run the application pool's worker processes under this account by default. In the IIS Management Console, under Advanced Settings for your application pool, ensure that the Identity is set to use **ApplicationPoolIdentity** as shown in the image below.



The IIS management process creates a secure identifier with the name of the application pool in the Windows Security System. Resources can be secured by using this identity; however, this identity is not a real user account and won't show up in the Windows User Management Console.

If you need to grant the IIS worker process elevated access to your application, you will need to modify the Access Control List (ACL) for the directory containing your application.

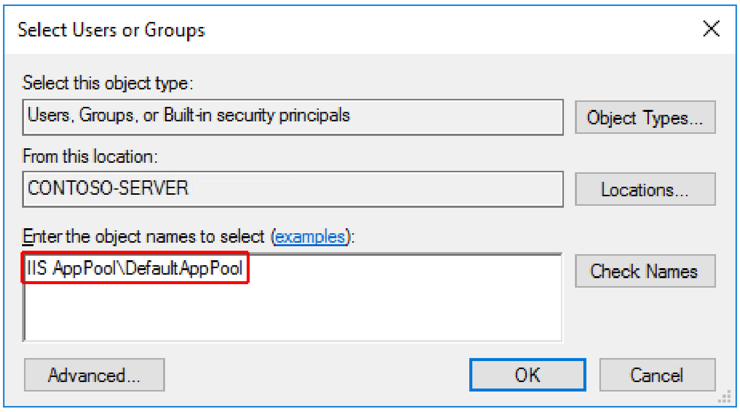
Open Windows Explorer and navigate to the directory.

Right click on the directory and click **Properties**.

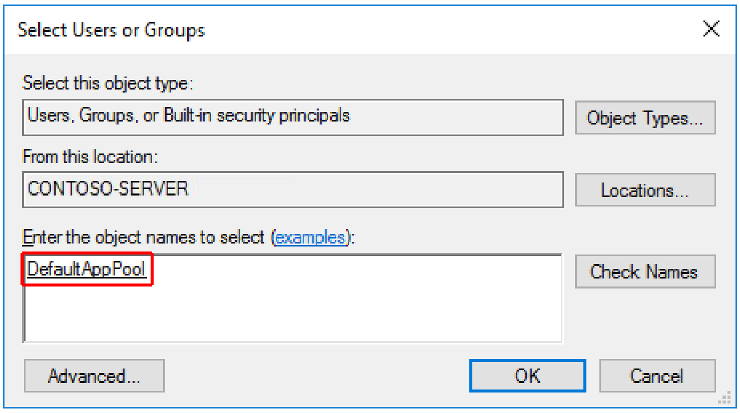
Under the **Security** tab, click the **Edit** button and then the **Add** button.

Click the **Locations** button and make sure you select your system.

Enter **IIS AppPool\DefaultAppPool** in **Enter the object names to select** textbox.



Click the **Check Names** button and then click **OK**.



You can also do this via a command prompt using **ICACLS** tool:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool" :F
```

Troubleshooting tips

To diagnose problems with IIS deployments, study browser output, examine the system's **Application** log through **Event Viewer**, and enable `stdout` logging. The **ASP.NET Core Module** log will be found on the path provided in the *stdoutLogFile* attribute of the `<aspNetCore>` element in *web.config*. Any folders on the path provided in the attribute value must exist in the deployment. You must also set *stdoutLogEnabled="true"*. Applications that use the `Microsoft.NET.Sdk.Web` SDK to create the *web.config* file will default the *stdoutLogEnabled* setting to *false*, so you must manually provide the *web.config* file or modify the file in order to enable `stdout` logging.

Several of the common errors do not appear in the browser, Application Log, and ASP.NET Core Module Log until the module *startupTimeLimit* (default: 120 seconds) and *startupRetryCount* (default: 2) have passed. Therefore, wait a full six minutes before deducing that the module has failed to start a process for the application.

One quick way to determine if the application is working properly is to run the application directly on Kestrel. If the application was published as a framework-dependent deployment, execute **dotnet my_application.dll** in the deployment folder, which is the IIS physical path to the application. If the application was published as a self-contained deployment, run the application's executable directly from a command prompt, **my_application.exe**, in the deployment folder. If Kestrel is listening on default port

5000, you should be able to browse the application at `http://localhost:5000/`. If the application responds normally at the Kestrel endpoint address, the problem is more likely related to the IIS-ASP.NET Core Module-Kestrel configuration and less likely within the application itself.

One way to determine if the IIS reverse proxy to the Kestrel server is working properly is to perform a simple static file request for a stylesheet, script, or image from the application's static files in *wwwroot* using Static File middleware. If the application can serve static files but MVC Views and other endpoints are failing, the problem is less likely related to the IIS-ASP.NET Core Module-Kestrel configuration and more likely within the application itself (for example, MVC routing or 500 Internal Server Error).

When Kestrel starts normally behind IIS but the app won't run on the system after successfully running locally, you can temporarily add an environment variable to *web.config* to set the `ASPNETCORE_ENVIRONMENT` to `Development`. As long as you don't override the environment in app startup, this will allow the developer exception page to appear when the app is run on the system. Setting the environment variable for `ASPNETCORE_ENVIRONMENT` in this way is only recommended for staging/testing systems that are not exposed to the Internet. Be sure you remove the environment variable from the *web.config* file when finished. For information on setting environment variables via *web.config* for the reverse proxy, see environmentVariables child element of aspNetCore.

In most cases, enabling application logging will assist in troubleshooting problems with application or the reverse proxy. See Logging for more information.

Our last troubleshooting tip pertains to apps that fail to run after upgrading either the .NET Core SDK on the development machine or package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. You can fix most of these issues by deleting the `bin` and `obj` folders in the project, clearing package caches at `%UserProfile%\.nuget\packages\` and `%LocalAppData%\Nuget\v3-cache`, restoring the project, and confirming that your prior deployment on the system has been completely deleted prior to re-deploying the app.

䤸 Tip

A convenient way to clear package caches is to obtain the `NuGet.exe` tool from NuGet.org, add it to your system PATH, and execute `nuget locals all -clear` from a command prompt.

Common errors

The following is not a complete list of errors. Should you encounter an error not listed here, please leave a detailed error message in the comments section below.

Installer unable to obtain VC++ Redistributable

Installer Exception: 0x80072efd or 0x80072f76 - Unspecified error

Installer Log Exception†: Error 0x80072efd or 0x80072f76: Failed to execute EXE package

†The log is located at C:\Users\{USER}\AppData\Local\Temp\dd_DotNetCoreWinSvrHosting__{timestamp}.log.

Troubleshooting:

If the system does not have Internet access while installing the server hosting bundle, this exception will occur when the installer is prevented from obtaining the *Microsoft Visual C++ 2015 Redistributable*. You may obtain an installer from the Microsoft Download Center. If the installer fails, you may not receive the .NET Core runtime required to host framework-dependent deployments. If you plan to host framework-dependent deployments, confirm that the runtime is installed in Programs & Features. You may obtain a runtime installer from .NET Downloads. After installing the runtime, restart the system or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory are not preserved during an OS upgrade. If you have the ASP.NET Core Module installed prior to an OS upgrade and then try to run any AppPool in 32-bit mode after an OS upgrade, you will encounter this issue. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Windows Server Hosting bundle. Select **Repair** when you run the installer.

Platform conflicts with RID

Browser: HTTP Error 502.5 - Process Failure

Application Log: Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:{PATH}\' failed to start process with commandline '"C:\{PATH}\my_application.{exe|dll}" ', ErrorCode = '0x80004005 : ff.

ASP.NET Core Module Log: Unhandled Exception: System.BadImageFormatException: Could not load file or assembly 'my_application.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

Confirm that the application runs locally on Kestrel. A process failure might be the result of a problem within the application. For more information, see Troubleshooting tips.

Confirm that you didn't set a `<PlatformTarget>` in your *.csproj* that conflicts with the RID. For example, don't specify a `<PlatformTarget>` of `x86` and publish with an RID of `win10-x64`, either by using *dotnet publish -c Release -r win10-x64* or by setting the `<RuntimeIdentifiers>` in your *.csproj* to `win10-x64`. The project will publish without warning or error but fail with the above logged exceptions on the system.

If this exception occurs for an Azure Apps deployment when upgrading an application and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

Browser: ERR_CONNECTION_REFUSED

Application Log: No entry

ASP.NET Core Module Log: Log file not created

Troubleshooting:

Confirm you are using the correct URI endpoint for the application. Check your bindings.

Confirm that the IIS website is not in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that you have enabled the proper role and features. See IIS Configuration.

Incorrect website physical path or application missing

Browser: 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.

Application Log: No entry

ASP.NET Core Module Log: Log file not created

Troubleshooting:

Check the IIS website **Basic Settings** and the physical application folder. Confirm that the application is in the folder at the IIS website **Physical path**.

Incorrect role, module not installed, or incorrect permissions

Browser: 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid.

Application Log: No entry

ASP.NET Core Module Log: Log file not created

Troubleshooting:

Confirm that you have enabled the proper role. See IIS Configuration.

Check **Programs & Features** and confirm that the **Microsoft ASP.NET Core Module** has been installed. If the **Microsoft ASP.NET Core Module** is not present in the list of installed programs, install the module. See Install the .NET Core Windows Server Hosting bundle.

Make sure that the **Application Pool > Process Model > Identity** is set to **ApplicationPoolIdentity** or your custom identity has the correct permissions to access the application's deployment folder.

Incorrect processPath, missing PATH variable, hosting bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

Browser: HTTP Error 502.5 - Process Failure

Application Log: Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:\{PATH}\' failed to start process with commandline '".\my_application.exe" ', ErrorCode = '0x80070002 : 0.

ASP.NET Core Module Log: Log file created but empty

Troubleshooting:

Confirm that the application runs locally on Kestrel. A process failure might be the result of a problem within the application. For more information, see Troubleshooting tips.

Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it is *dotnet* for a framework-dependent deployment or .*my_application.exe* for a self-contained deployment.

For a framework-dependent deployment, *dotnet.exe* might not be accessible via the PATH settings. Confirm that C:*Program Files\dotnet* exists in the System PATH settings.

For a framework-dependent deployment, *dotnet.exe* might not be accessible for the user identity of the Application Pool. Confirm that the AppPool user identity has access to the C:*Program Files\dotnet* directory. Confirm that there are no deny rules configured for the AppPool user identity on the C:*Program Files\dotnet* and application directories.

You may have deployed a framework-dependent deployment and installed .NET Core without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

You may have deployed a framework-dependent deployment without installing the .NET Core runtime on the hosting system. If you're attempting to deploy a framework-dependent deployment and have not installed the .NET Core runtime, run the **.NET Core Windows Server Hosting bundle installer** on the system. See Install the .NET Core Windows Server Hosting bundle. If you are attempting to install the .NET Core runtime on a system without an Internet connection, obtain the runtime from .NET Downloads and run the hosting bundle installer from an administrator command prompt to only install the module using **DotNetCore.1.1.0-WindowsHosting.exe OPT_INSTALL_LTS_REDIST=0 OPT_INSTALL_FTS_REDIST=0**. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

You may have deployed a framework-dependent deployment and installed .NET Core without restarting the system/IIS. Either restart the system or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

You may have deployed a framework-dependent deployment and the *Microsoft Visual C++ 2015 Redistributable (x64)* is not installed on the system. You may obtain an installer from the Microsoft Download Center.

Incorrect arguments of <aspNetCore> element

Browser: HTTP Error 502.5 - Process Failure

Application Log: Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:\{PATH}\' failed to start process with commandline '"dotnet" .\my_application.dll', ErrorCode = '0x80004005 : 80008081.

ASP.NET Core Module Log: The application to execute does not exist: 'PATH\my_application.dll'

Troubleshooting:

Confirm that the application runs locally on Kestrel. A process failure might be the result of a problem within the application. For more information, see Troubleshooting tips.

Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it is either (a) *.\my_applciation.dll* for a framework-dependent deployment; or (b) not present, an empty string (*arguments*=""), or a list of your application's arguments (*arguments*="*arg1, arg2, ...*") for a self-contained deployment.

Missing .NET Framework version

Browser: 502.3 Bad Gateway - There was a connection error while trying to route the request.

Application Log: ErrorCode = Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:\{PATH}\' failed to start process with commandline '"dotnet" .\my_application.dll', ErrorCode = '0x80004005 : 80008081.

ASP.NET Core Module Log: Missing method, file, or assembly exception. The method, file, or assembly specified in the exception is a .NET Framework method, file, or assembly.

Troubleshooting:

Install the .NET Framework version missing from the system.

For a framework-dependent deployment, confirm that you have the correct runtime installed on the system. For example if you upgrade a project from 1.0 to 1.1, deploy to the hosting system, and receive this exception, ensure you install the 1.1 framework on the hosting system.

Stopped Application Pool

Browser: 503 Service Unavailable

Application Log: No entry

ASP.NET Core Module Log: Log file not created

Troubleshooting

Confirm that the Application Pool is not in the *Stopped* state.

IIS Integration middleware not implemented

Browser: HTTP Error 502.5 - Process Failure

Application Log: Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:\{PATH}\' created process with commandline '"C:\{PATH}\my_application.{exe|dll}" ' but either crashed or did not reponse or did not listen on the given port '{PORT}', ErrorCode = '0x800705b4'

ASP.NET Core Module Log: Log file created and shows normal operation.

Troubleshooting

Confirm that the application runs locally on Kestrel. A process failure might be the result of a problem within the application. For more information, see Troubleshooting tips.

Confirm that you have correctly referenced the IIS Integration middleware by calling the *.UseIISIntegration()* method on the application's *WebHostBuilder()*.

Sub-application includes a <handlers> section

Browser: HTTP Error 500.19 - Internal Server Error

Application Log: No entry

ASP.NET Core Module Log: Log file created and shows normal operation for the root application. Log file not created for the sub-application.

Troubleshooting

Confirm that the sub-application's *web.config* file doesn't include a `<handlers>` section.

Application configuration general issue

Browser: HTTP Error 502.5 - Process Failure

Application Log: Application 'MACHINE/WEBROOT/APPHOST/MY_APPLICATION' with physical root 'C:\{PATH}\' created process with commandline '"C:\{PATH}\my_application.{exe|dll}" ' but either crashed or did not reponse or did not listen on the given port '{PORT}', ErrorCode = '0x800705b4'

ASP.NET Core Module Log: Log file created but empty

Troubleshooting

This general exception indicates that the process failed to start, most likely due to an application configuration issue. Referring to Directory Structure, confirm that your application's deployed files and folders are appropriate and that your application's configuration files are present and contain the correct settings for your app and environment. For more information, see Troubleshooting tips.

Resources

ASP.NET Core Module overview

ASP.NET Core Module Configuration Reference

Using IIS Modules with ASP.NET Core

Introduction to ASP.NET Core

The Official Microsoft IIS Site

Microsoft TechNet Library: Windows Server

ASP.NET Core Module configuration reference

By Luke Latham, Rick Anderson, and Sourabh Shirhatti

This document provides details on how to configure the ASP.NET Core Module for hosting ASP.NET Core applications. For an introduction to the ASP.NET Core Module and installation instructions, see the ASP.NET Core Module overview.

Configuration via web.config

The ASP.NET Core Module is configured via a site or application *web.config* file and has its own `aspNetCore` configuration section within `system.webServer`. Here's an example *web.config* file that the `Microsoft.NET.Sdk.Web` SDK will provide when the project is published for a framework-dependent deployment with placeholders for the `processPath` and `arguments`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%"
        arguments="%LAUNCHER_ARGS%"
        stdoutLogEnabled="false"
        stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

The *web.config* example below is for a self-contained deployment to the Azure App Service. For more information, see Publishing to IIS. See Configuration of sub-applications for an important note pertaining to the configuration of *web.config* files in sub-applications.

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\MyApp.exe"
        stdoutLogEnabled="false"
        stdoutLogFile="\\?\%home%\LogFiles\stdout" />
  </system.webServer>
</configuration>
```

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION
processPath	Required string attribute. Path to the executable that will launch a process listening for HTTP requests. Relative paths are supported. If the path begins with '.', the path is considered to be relative to the site root. There is no default value.
arguments	Optional string attribute. Arguments to the executable specified in **processPath**. The default value is an empty string.

ATTRIBUTE	DESCRIPTION
startupTimeLimit	Optional integer attribute. Duration in seconds that the module will wait for the executable to start a process listening on the port. If this time limit is exceeded, the module will kill the process. The module will attempt to launch the process again when it receives a new request and will continue to attempt to restart the process on subsequent incoming requests unless the application fails to start **rapidFailsPerMinute** number of times in the last rolling minute. The default value is 120.
shutdownTimeLimit	Optional integer attribute. Duration in seconds for which the module will wait for the executable to gracefully shutdown when the *app_offline.htm* file is detected. The default value is 10.
rapidFailsPerMinute	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module will stop launching the process for the remainder of the minute. The default value is 10.
requestTimeout	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module will wait for a response from the process listening on %ASPNETCORE_PORT%. The default value is "00:02:00".
stdoutLogEnabled	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** will be redirected to the file specified in **stdoutLogFile**. The default value is false.
stdoutLogFile	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** will be logged. Relative paths are relative to the root of the site. Any path starting with '.' will be relative to the site root and all other paths will be treated as absolute paths. A timestamp and the file extension will automatically be added to the filename provided. Any folders provided in the path must exist in order for the module to create the log file. The default value is `aspnetcore-stdout`.

ATTRIBUTE	DESCRIPTION
forwardWindowsAuthToken	true or false. If true, the token will be forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It is the responsibility of that process to call CloseHandle on this token per request. The default value is true.
disableStartUpErrorPage	true or false. If true, the **502.5 - Process Failure** page will be suppressed, and the 502 status code page configured in your *web.config* will take precedence. The default value is false.

Setting environment variables

The ASP.NET Core Module allows you specify environment variables for the process specified in the `processPath` attribute by specifying them in one or more `environmentVariable` child elements of an `environmentVariables` collection element under the `aspNetCore` element. Environment variables set in this section take precedence over system environment variables for the process.

The example below sets two environment variables. `ASPNETCORE_ENVIRONMENT` will configure the application's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the developer exception page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that will read the value on startup to form a path in order to load the app's configuration file.

```
<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>
```

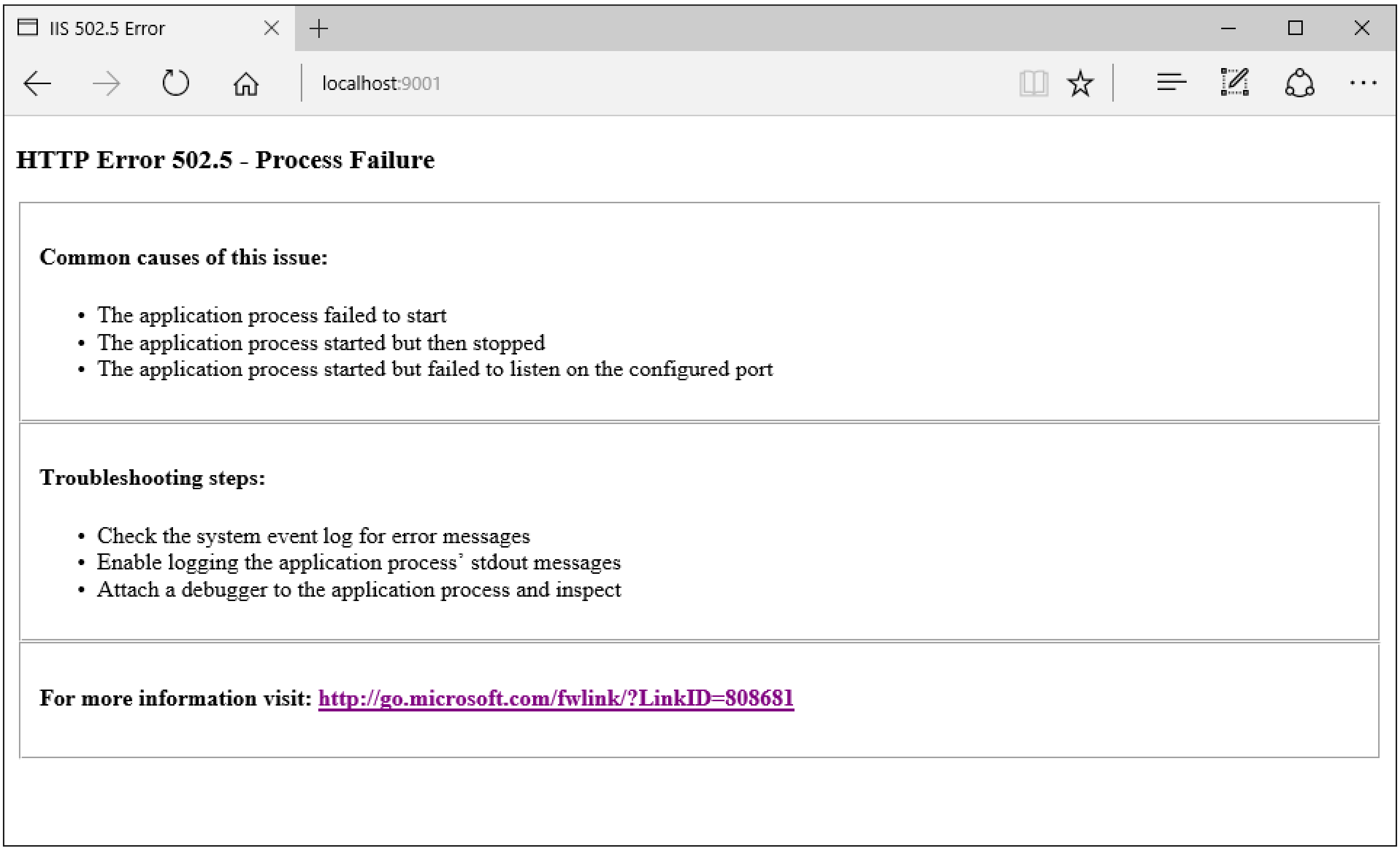
app_offline.htm

If you place a file with the name *app_offline.htm* at the root of a web application directory, the ASP.NET Core Module will attempt to gracefully shutdown the app and stop processing incoming requests. If the app is still running after `shutdownTimeLimit` number of seconds, the ASP.NET Core Module will kill the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module will respond to requests by sending back the contents of the *app_offline.htm* file. Once the *app_offline.htm* file is removed, the next request loads the application, which then responds to requests.

Start-up error page

If the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, you will see an HTTP 502.5 status code page. To suppress this page and revert to the default IIS 502 status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors `<httpErrors>`.



Log creation and redirection

The ASP.NET Core Module redirects `stdout` and `stderr` logs to disk if you set the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element. Any folders in the `stdoutLogFile` path must exist in order for the module to create the log file. A timestamp and file extension will be added automatically when the log file is created. Logs are not rotated, unless process recycling/restart occurs. It is the responsibility of the hoster to limit the disk space the logs consume. Using the `stdout` log is only recommended for troubleshooting application startup issues and not for general application logging purposes.

Here's a sample `aspNetCore` element that configures `stdout` logging. The `stdoutLogFile` path shown in the example is appropriate for the Azure App Service. A local path or network share path is acceptable for local logging. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile="\\?\%home%\LogFiles\stdout">
</aspNetCore>
```

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **SYSTEM** account. Because the local system account does not have modify permission for the share path which is used by the IIS Shared Configuration, the installer will hit an access denied error when attempting to configure the module settings in *applicationHost.config* on the share.

The unsupported workaround is to disable the IIS Shared Configuration, run the installer, export the updated *applicationHost.config* file to the share, and re-enable the IIS Shared Configuration.

Module, schema, and configuration file locations

Module

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

Schema

IIS

%windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml

IIS Express

%ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml

Configuration

IIS

%windir%\System32\inetsrv\config\applicationHost.config

IIS Express

.vs\config\applicationHost.config

You can search for *aspnetcore.dll* in the *applicationHost.config* file. For IIS Express, the *applicationHost.config* file won't exist by default. The file is created at *{application root}.vs\config* when you start any web application project in the Visual Studio solution.

Using IIS Modules with ASP.NET Core

By Luke Latham

ASP.NET Core applications are hosted by IIS in a reverse-proxy configuration. Some of the native IIS modules and all of the IIS managed modules are not available to process requests for ASP.NET Core apps. In many cases, ASP.NET Core offers an alternative to the features of IIS native and managed modules.

Native Modules

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
Anonymous Authentication `AnonymousAuthenticationModule`	Yes	
Basic Authentication `BasicAuthenticationModule`	Yes	
Client Certification Mapping Authentication `CertificateMappingAuthenticationModule`	Yes	
CGI `CgiModule`	No	
Configuration Validation `ConfigurationValidationModule`	Yes	
HTTP Errors `CustomErrorModule`	No	Status Code Pages Middleware
Custom Logging `CustomLoggingModule`	Yes	
Default Document `DefaultDocumentModule`	No	Default Files Middleware
Digest Authentication `DigestAuthenticationModule`	Yes	
Directory Browsing `DirectoryListingModule`	No	Directory Browsing Middleware
Dynamic Compression `DynamicCompressionModule`	Yes	Response Compression Middleware
Tracing `FailedRequestsTracingModule`	Yes	ASP.NET Core Logging
File Caching `FileCacheModule`	No	Response Caching Middleware
HTTP Caching `HttpCacheModule`	No	Response Caching Middleware

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
HTTP Logging `HttpLoggingModule`	Yes	ASP.NET Core Logging Implementations: elmah.io, Loggr, NLog, Serilog
HTTP Redirection `HttpRedirectionModule`	Yes	URL Rewriting Middleware
IIS Client Certificate Mapping Authentication `IISCertificateMappingAuthenticationModule`	Yes	
IP and Domain Restrictions `IpRestrictionModule`	Yes	
ISAPI Filters `IsapiFilterModule`	Yes	Middleware
ISAPI `IsapiModule`	Yes	Middleware
Protocol Support `ProtocolSupportModule`	Yes	
Request Filtering `RequestFilteringModule`	Yes	URL Rewriting Middleware IRule
Request Monitor `RequestMonitorModule`	Yes	
URL Rewriting `RewriteModule`	Yes†	URL Rewriting Middleware
Server Side Includes `ServerSideIncludeModule`	No	
Static Compression `StaticCompressionModule`	No	Response Compression Middleware
Static Content `StaticFileModule`	No	Static File Middleware
Token Caching `TokenCacheModule`	Yes	
URI Caching `UriCacheModule`	Yes	
URL Authorization `UrlAuthorizationModule`	Yes	ASP.NET Core Identity
Windows Authentication `WindowsAuthenticationModule`	Yes	

†The URL Rewrite Module's `isFile` and `isDirectory` do not work with ASP.NET Core applications due to the changes in

directory structure.

Managed Modules

MODULE	.NET CORE ACTIVE	ASP.NET CORE OPTION
AnonymousIdentification	No	
DefaultAuthentication	No	
FileAuthorization	No	
FormsAuthentication	No	Cookie Authentication Middleware
OutputCache	No	Response Caching Middleware
Profile	No	
RoleManager	No	
ScriptModule-4.0	No	
Session	No	Session Middleware
UrlAuthorization	No	
UrlMappingsModule	No	URL Rewriting Middleware
UrlRoutingModule-4.0	No	ASP.NET Core Identity
WindowsAuthentication	No	

IIS Manager application changes

When you use IIS Manager to configure settings, you're directly changing the *web.config* file of the app. If you deploy your app and include *web.config*, any changes you made with IIS Manger will be overwritten by the deployed *web.config file*. Therefore if you make changes to the server's *web.config* file, copy the updated *web.config* file to your local project immediately.

Disabling IIS modules

If you have an IIS module configured at the server level that you would like to disable for an application, you can do so with an addition to your *web.config* file. Either leave the module in place and deactivate it using a configuration setting (if available) or remove the module from the app.

Module deactivation

Many modules offer a configuration setting that will allow you to disable them without removing them from the application. This is the simplest and quickest way to deactivate a module. For example if you wish to disable the IIS URL Rewrite Module, use the `<httpRedirect>` element as shown below. For more information on disabling modules with configuration settings, follow the links in the *Child Elements* section of IIS `<system.webServer>`.

```
<configuration>
  <system.webServer>
     <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

Module removal

If you opt to remove a module with a setting in *web.config*, you must unlock the module and unlock the `<modules>` section of *web.config* first. The steps are outlined below:

Unlock the module at the server level. Click on the IIS server in the IIS Manager **Connections** sidebar. Open the **Modules** in the **IIS** area. Click on the module in the list. In the **Actions** sidebar on the right, click **Unlock**. Unlock as many modules as you plan to remove with *web.config* later.

Deploy your application without a `<modules>` section in *web.config*. If you deploy an app with a *web.config* containing the `<modules>` section without having unlocked the section first in the IIS Manager, the Configuration Manager will throw an exception when you try to unlock the section. Therefore, deploy your application without a `<modules>` section.

Unlock the `<modules>` section of *web.config*. In the **Connections** sidebar, click the website in **Sites**. In the **Management** area, open the **Configuration Editor**. Use the navigation controls to select the `system.webServer/modules` section. In the **Actions** sidebar on the right, click to **Unlock** the section.

At this point, you will be able to add a `<modules>` section to your *web.config* file with a `<remove>` element to remove the module from the application. You can add multiple `<remove>` elements to remove multiple modules. Don't forget that if you make *web.config* changes on the server to make them immediately in the project locally. Removing a module this way won't affect your use of the module with other apps on the server.

```
<configuration>
 <system.webServer>
   <modules>
     <remove name="MODULE_NAME" />
   </modules>
 </system.webServer>
</configuration>
```

For an IIS installation with the default modules installed, you can use the following `<module>` section to remove the default modules.

```
<modules>
  <remove name="CustomErrorModule" />
  <remove name="DefaultDocumentModule" />
  <remove name="DirectoryListingModule" />
  <remove name="HttpCacheModule" />
  <remove name="HttpLoggingModule" />
  <remove name="ProtocolSupportModule" />
  <remove name="RequestFilteringModule" />
  <remove name="StaticCompressionModule" />
  <remove name="StaticFileModule" />
</modules>
```

You can also remove an IIS module with *Appcmd.exe*. Provide the `MODULE_NAME` and `APPLICATION_NAME` in the command shown below:

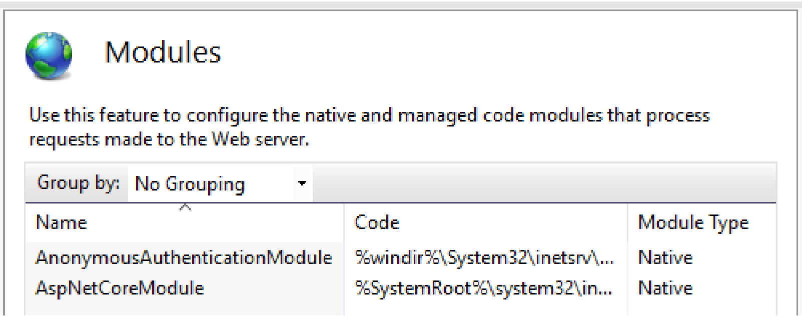
```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

Here's how to remove the `DynamicCompressionModule` from the Default Web Site:

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule /app.name:"Default Web Site"
```

Minimal module configuration

The only modules required to run an ASP.NET Core application are the Anonymous Authentication Module and the ASP.NET Core Module.



Resources

Publishing to IIS
IIS Modules Overview
Customizing IIS 7.0 Roles and Modules
IIS `<system.webServer>`

Host an ASP.NET Core app in a Windows Service

By Tom Dykstra

The recommended way to host an ASP.NET Core app on Windows when you don't use IIS is to run it in a Windows Service. That way it can automatically start after reboots and crashes, without waiting for someone to log in.

View or download sample code See the Next Steps section for instructions on how to run it.

Prerequisites

The app must run on the .NET framework runtime. In the *.csproj* file, specify appropriate values for TargetFramework and RuntimeIdentifier. Here's an example:

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <RuntimeIdentifier>win7-x86</RuntimeIdentifier>
</PropertyGroup>
```

When creating a project in Visual Studio, use the **ASP.NET Core Application (.NET Framework)** template.

If the app will get requests from the internet (not just from an internal network), it must use the WebListener web server rather than Kestrel. Kestrel must be used with IIS for edge deployments. For more information, see When to use Kestrel with a reverse proxy.

Getting started

This section explains the minimum changes required to set up an existing ASP.NET Core project to run in a service.

Install the NuGet package Microsoft.AspNetCore.Hosting.WindowsServices.

Make the following changes in `Program.Main`:

Call `host.RunAsService` instead of `host.Run`.

If your code calls `UseContentRoot`, use a path to the publish location instead of `Directory.GetCurrentDirectory()`

```
public static void Main(string[] args)
{
    var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
    var pathToContentRoot = Path.GetDirectoryName(pathToExe);

    var host = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(pathToContentRoot)
    .UseIISIntegration()
    .UseStartup<Startup>()
    .UseApplicationInsights()
    .Build();

    host.RunAsService();
}
```

Publish the application to a folder.

Use dotnet publish or a Visual Studio publish profile that publishes to a folder.

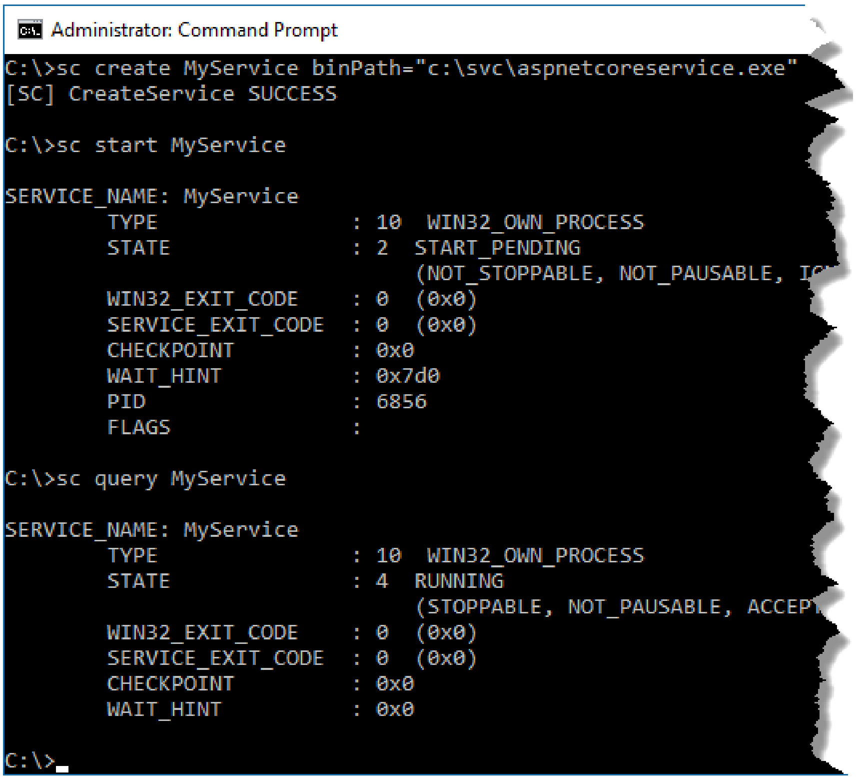
Test by creating and starting the service.

Open an administrator command prompt window to use the sc.exe command-line tool to create and start a service.

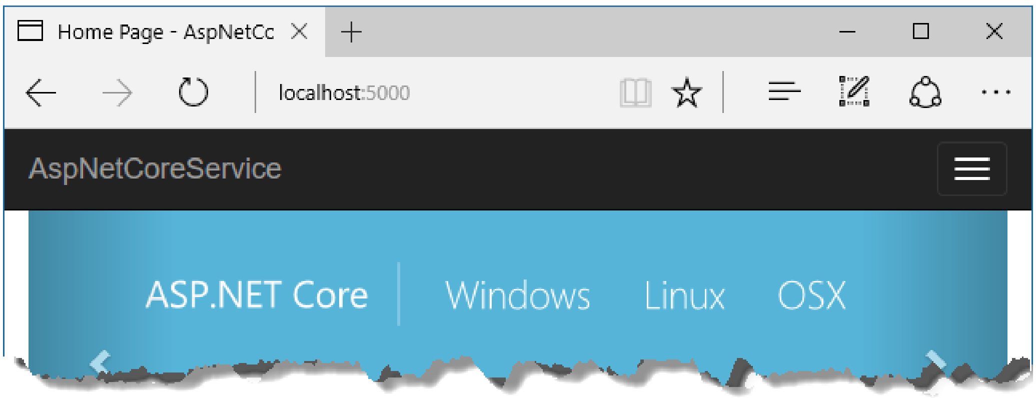
If you name your service MyService, you publish your app to `c:\svc`, and the app itself is named AspNetCoreService, the commands would look like this:

```
sc create MyService binPath="C:\Svc\AspNetCoreService.exe"
sc start MyService
```

The `binPath` value is the path to your app's executable, including the executable filename itself.



When these commands finish, you can browse to the same path as when you run as a console app (by default, `http://localhost:5000`)



Provide a way to run outside of a service

It's easier to test and debug when you're running outside of a service, so it's customary to add code that calls `host.RunAsService` only under certain conditions. For example, you could run as a console app if you get a `--console` command-line argument or if the debugger is attached.

```
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 = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(pathToContentRoot)
    .UseIISIntegration()
    .UseStartup<Startup>()
    .UseApplicationInsights()
    .Build();

    if (isService)
    {
        host.RunAsService();
    }
    else
    {
        host.Run();
    }
}
```

Handle stopping and starting events

If you want to handle `OnStarting`, `OnStarted`, and `OnStopping` events, make the following additional changes:

Create a class that derives from `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();
    }
}
```

Create an extension method for `IWebHost` that passes your custom `WebHostService` to `ServiceBase.Run`.

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

In `Program.Main` change call the new extension method instead of `host.RunAsService`.

```
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 = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(pathToContentRoot)
    .UseIISIntegration()
    .UseStartup<Startup>()
    .UseApplicationInsights()
    .Build();

    if (isService)
    {
        host.RunAsCustomService();
    }
    else
    {
        host.Run();
    }
}
```

If your custom `WebHostService` code needs to get a service from dependency injection (such as a logger), you can get it from the `Services` property of `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();
    }
}
```

Next steps

The sample application that accompanies this article is a simple MVC web app that has been modified as shown in preceding code examples. To run it in a service, do the following steps:

Publish to *c:\svc*.

Open an administrator window.

Enter the following commands:

```
sc create MyService binPath="c:\svc\aspnetcoreservice.exe"
sc start MyService
```

In a browser, go to http://localhost:5000 to verify that it's running.

If the app doesn't start up as expected when running in a service, a quick way to make error messages accessible is to add a logging provider such as the Windows EventLog provider.

Acknowledgments

This article was written with the help of sources that were already published. The earliest and most useful of them were these:

Hosting ASP.NET Core as Windows service
How to host your ASP.NET Core in a Windows Service

Set up a hosting environment for ASP.NET Core on Linux with Nginx, and deploy to it

By Sourabh Shirhatti

In this guide, we will cover setting up a production-ready ASP.NET environment on an Ubuntu 16.04 Server.

Note: For Ubuntu 14.04 supervisord is recommended as a solution for monitoring the Kestrel process. systemd is not available on Ubuntu 14.04. See previous version of this document

We will take an existing ASP.NET Core application and place it behind a reverse-proxy server. We will then setup the reverse-proxy server to forward requests to our Kestrel web server.

Additionally we will ensure our web application runs on startup as a daemon and configure a process management tool to help restart our web application in the event of a crash to guarantee high availability.

Prerequisites

Access to an Ubuntu 16.04 Server with a standard user account with sudo privilege.

An existing ASP.NET Core application.

Copy over your app

Run `dotnet publish` from your dev environment to package your app into a self-contained directory that can run on your server.

Copy your ASP.NET Core app to your server using whatever tool (SCP, FTP, etc) integrates into your workflow. Test your app, for example:

From the command line, run `dotnet yourapp.dll`
In a browser, navigate to `http://<serveraddress>:<port>` to verify you app works on Linux.

Note: You can use Yeoman to create a new ASP.NET Core application for a new project.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web applications. The reverse proxy terminates the HTTP request and forwards it to the ASP.NET application.

Why use a reverse-proxy server?

Kestrel is great for serving dynamic content from ASP.NET, however the web serving parts aren't as feature rich as full-featured servers like IIS, Apache or Nginx. A reverse proxy-server can allow you to offload work like serving static content, caching requests, compressing requests, and SSL termination from the HTTP server. The reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, we are going to use a single instance of Nginx that runs on the same server alongside your HTTP server. However, based on your requirements you may choose a different setup.

When setting up a reverse-proxy server other than IIS, you must call `app.UseIdentity` (in `Configure`) before any other external providers.

Because requests are forwarded by reverse-proxy, use `ForwardedHeaders` middleware (from `Microsoft.AspNetCore.HttpOverrides` package) in order to set the redirect URI with the `X-Forwarded-For` and `X-Forwarded-Proto` headers instead of `Request.Scheme` and `Request.Host`."

Add `UseForwardedHeaders` to `Configure` before calling `app.UseFacebookAuthentication` or similar:

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});
```

Install Nginx

```
sudo apt-get install nginx
```

Note

If you plan to install optional Nginx modules you may be required to build Nginx from source.

We are going to `apt-get` to install Nginx. The installer also creates a System V init script that runs Nginx as daemon on system startup. Since we just installed Nginx for the first time, we can explicitly start it by running

```
sudo service nginx start
```

At this point you should be able to navigate to your browser and see the default landing page for Nginx.

Configure Nginx

We will now configure Nginx as a reverse proxy to forward requests to our ASP.NET application

We will be modifying the `/etc/nginx/sites-available/default`, so open it up in your favorite text editor and replace the contents with the following.

```
server {
    listen 80;
    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 $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

This is one of the simplest configuration files for Nginx that forwards incoming public traffic on your port `80` to a port `5000` that your web application will listen on.

Once you have completed making changes to your nginx configuration you can run `sudo nginx -t` to verify the syntax of your configuration files. If the configuration file test is successful you can ask nginx to pick up the changes by running `sudo nginx -s reload`.

Monitoring our application

Nginx is now setup to forward requests made to `http://localhost:80` on to the ASP.NET Core application running on Kestrel at `http://127.0.0.1:5000`. However, Nginx is not set up to manage the Kestrel process. We will use *systemd* and create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping and managing processes.

Create the service file

Create the service definition file

```
sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

An example service file for our application.

```
[Unit]
Description=Example .NET Web API Application 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

[Install]
WantedBy=multi-user.target
```

Note

If the user *www-data* is not used by your configuration, the user defined here must be created first and given proper ownership for files

Save the file and enable the service.

```
systemctl enable kestrel-hellomvc.service
```

Start the service and verify that it is running.

```
systemctl start kestrel-hellomvc.service
systemctl status kestrel-hellomvc.service

● kestrel-hellomvc.service - Example .NET Web API Application 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
```

With the reverse proxy configured and Kestrel managed through systemd, the web application is fully configured and can be accessed from a browser on the local machine at `http://localhost`. Inspecting the response headers, the **Server** still shows the ASP.NET Core application being served by 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
```

Viewing logs

Since the web application using Kestrel is managed using systemd, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by systemd. To view the `kestrel-hellomvc.service` specific items, use the following command.

```
sudo journalctl -fu kestrel-hellomvc.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago` or a combination of these can reduce the amount of entries returned.

```
sudo journalctl -fu kestrel-hellomvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Securing our application

Enable `AppArmor`

Linux Security Modules (LSM) is a framework that is part of the Linux kernel since Linux 2.6 that supports different implementations of security modules. `AppArmor` is a LSM that implements a Mandatory Access Control system which allows you to confine the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configuring our firewall

Close off all external ports that are not in use. Uncomplicated firewall (ufw) provides a frontend for `iptables` by providing a command-line interface for configuring the firewall. Verify that `ufw` is configured to allow traffic on any ports you need.

```
sudo apt-get install ufw
sudo ufw enable

sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

Securing Nginx

The default distribution of Nginx doesn't enable SSL. To enable all the security features we require, we will build from source.

Download the source and install the build dependencies

```
# Install the build dependencies
sudo apt-get update
sudo apt-get install build-essential zlib1g-dev libpcre3-dev libssl-dev libxslt1-dev libxml2-dev libgd2-xpm-dev libgeoip-dev libgoogle-perftools-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
```

Change the Nginx response name

Edit *src/http/ngx_http_header_filter_module.c*

```
static char ngx_http_server_string[] = "Server: Your Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Your Web Server" CRLF;
```

Configure the options and build

The PCRE library is required for regular expressions. Regular expressions are used in the location directive for the ngx_http_rewrite_module. The http_ssl_module adds HTTPS protocol support.

Consider using a web application firewall like *ModSecurity* to harden your application.

```
./configure
--with-pcre=../pcre-8.38
--with-zlib=../zlib-1.2.8
--with-http_ssl_module
--with-stream
--with-mail=dynamic
```

Configure SSL

Configure your server to listen to HTTPS traffic on port `443` by specifying a valid certificate issued by a trusted Certificate Authority (CA).

Harden your security by employing some of the practices suggested below like choosing a stronger cipher and redirecting all traffic over HTTP to HTTPS.

Adding an `HTTP Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS only.

Do not add the Strict-Transport-Security header or chose an appropriate `max-age` if you plan to disable SSL in the future.

Add `/etc/nginx/proxy.conf` configuration file.

```
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;
```

Edit `/etc/nginx/nginx.conf` configuration file. The example contains both http and server sections in one configuration file.

```
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;
        }
    }
}
```

Secure Nginx from clickjacking

Clickjacking is a malicious technique to collect an infected user's clicks. Clickjacking tricks the victim (visitor) into clicking on an infected site. Use X-FRAME-OPTIONS to secure your site.

Edit the nginx.conf file.

```
sudo nano /etc/nginx/nginx.conf
```

Add the the line `add_header X-Frame-Options "SAMEORIGIN";` and save the file, then restart Nginx.

MIME-type sniffing

This header prevents Internet Explorer from MIME-sniffing a response away from the declared content-type as the header instructs the browser not to override the response content type. With the nosniff option, if the server says the content is text/html, the browser will render it as text/html.

Edit the nginx.conf file.

```
sudo nano /etc/nginx/nginx.conf
```

Add the the line `add_header X-Content-Type-Options "nosniff"` and save the file, then restart Nginx.

Set up a hosting environment for ASP.NET Core on Linux with Apache, and deploy to it

By Shayne Boyer

Apache is a very popular HTTP server and can be configured as a proxy to redirect HTTP traffic similar to nginx. In this guide, we will learn how to set up Apache on CentOS 7 and use it as a reverse proxy to welcome incoming connections and redirect them to the ASP.NET Core application running on Kestrel. For this purpose, we will use the *mod_proxy* extension and other related Apache modules.

Prerequisites

A server running CentOS 7, with a standard user account with sudo privilege.
An existing ASP.NET Core application.

Publish your application

Run `dotnet publish -c Release` from your development environment to package your application into a self-contained directory that can run on your server. The published application must then be copied to the server using SCP, FTP or other file transfer method.

Note

Under a production deployment scenario, a continuous integration workflow does the work of publishing the application and copying the assets to the server.

Configure a proxy server

A reverse proxy is a common setup for serving dynamic web applications. The reverse proxy terminates the HTTP request and forwards it to the ASP.NET application.

A proxy server is one which forwards client requests to another server instead of fulfilling them itself. A reverse proxy forwards to a fixed destination, typically on behalf of arbitrary clients. In this guide, Apache is being configured as the reverse proxy running on the same server that Kestrel is serving the ASP.NET Core application.

Each piece of the application can exist on separate physical machines, Docker containers, or a combination of configurations depending on your architectural needs or restrictions.

Install Apache

Installing the Apache web server on CentOS is a single command, but first let's update our packages.

```
sudo yum update -y
```

This ensures that all of the installed packages are updated to their latest version. Install Apache using `yum`

```
sudo yum -y install httpd mod_ssl
```

The output should reflect something similar to the following.

```
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!
```

Note

In this example the output reflects httpd.86_64 since the CentOS 7 version is 64 bit. The output may be different for your server. To verify where Apache is installed, run `whereis httpd` from a command prompt.

Configure Apache for reverse proxy

Configuration files for Apache are located within the `/etc/httpd/conf.d/` directory. Any file with the **.conf** extension will be processed in alphabetical order in addition to the module configuration files in `/etc/httpd/conf.modules.d/`, which contains any configuration files necessary to load modules.

Create a configuration file for your app, for this example we'll call it `hellomvc.conf`

```
<VirtualHost *:80>
    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
</VirtualHost>
```

The *VirtualHost* node, of which there can be multiple in a file or on a server in many files, is set to listen on any IP address using port 80. The next two lines are set to pass all requests received at the root to the machine 127.0.0.1 port 5000 and in reverse. For there to be bi-directional communication, both settings *ProxyPass* and *ProxyPassReverse* are required.

Logging can be configured per VirtualHost using *ErrorLog* and *CustomLog* directives. *ErrorLog* is the location where the server will log errors and *CustomLog* sets the filename and format of log file. In our case this is where request information will be logged. There will be one line for each request.

Save the file, and test the configuration. If everything passes, the response should be `Syntax [OK]`.

```
sudo service httpd configtest
```

Restart Apache.

```
sudo systemctl restart httpd
sudo systemctl enable httpd
```

Monitoring our application

Apache is now setup to forward requests made to `http://localhost:80` on to the ASP.NET Core application running on Kestrel at `http://127.0.0.1:5000`. However, Apache is not set up to manage the Kestrel process. We will use *systemd* and create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping and managing processes.

Create the service file

Create the service definition file

```
    sudo nano /etc/systemd/system/kestrel-hellomvc.service
```

An example service file for our application.

```
[Unit]
    Description=Example .NET Web API Application running on CentOS 7

    [Service]
    WorkingDirectory=/var/aspnetcore/hellomvc
    ExecStart=/usr/local/bin/dotnet /var/aspnetcore/hellomvc/hellomvc.dll
    Restart=always
    RestartSec=10                                    # Restart service after 10 seconds if dotnet
service crashes
    SyslogIdentifier=dotnet-example
    User=apache
    Environment=ASPNETCORE_ENVIRONMENT=Production

    [Install]
    WantedBy=multi-user.target
```

Note

User -- If the user *apache* is not used by your configuration, the user defined here must be created first and given proper ownership for files

Save the file and enable the service.

```
    systemctl enable kestrel-hellomvc.service
```

Start the service and verify that it is running.

```
    systemctl start kestrel-hellomvc.service
    systemctl status kestrel-hellomvc.service

    ● kestrel-hellomvc.service - Example .NET Web API Application 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
```

With the reverse proxy configured and Kestrel managed through systemd, the web application is fully configured and can be accessed from a browser on the local machine at `http://localhost`. Inspecting the response headers, the **Server** still shows the ASP.NET Core application being served by 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
```

Viewing logs

Since the web application using Kestrel is managed using systemd, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by systemd. To view the `kestrel-hellomvc.service` specific items, use the following command.

```
sudo journalctl -fu kestrel-hellomvc.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago` or a combination of these can reduce the amount of entries returned.

```
sudo journalctl -fu kestrel-hellomvc.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Securing our application

Configure firewall

Firewalld is a dynamic daemon to manage firewall with support for network zones, although you can still use iptables to manage ports and packet filtering. Firewalld should be installed by default, `yum` can be used to install the package or verify.

```
sudo yum install firewalld -y
```

Using `firewalld` you can open only the ports needed for the application. In this case, port 80 and 443 are used. The following commands permanently sets these to open.

```
sudo firewall-cmd --add-port=80/tcp --permanent
sudo firewall-cmd --add-port=443/tcp --permanent
```

Reload the firewall settings, and check the available services and ports in the default zone. Options are available by inspecting `firewall-cmd -h`

```
sudo firewall-cmd --reload
sudo firewall-cmd --list-all
```

```
public (default, active)
interfaces: eth0
sources:
services: dhcpv6-client
ports: 443/tcp 80/tcp
masquerade: no
forward-ports:
icmp-blocks:
rich rules:
```

SSL configuration

To configure Apache for SSL, the mod_ssl module is used. This was installed initially when we installed the `httpd` module. If it was missed or not installed, use yum to add it to your configuration.

```
sudo yum install mod_ssl
```

To enforce SSL, install `mod_rewrite`

```
sudo yum install mod_rewrite
```

The `hellomvc.conf` file that was created for this example needs to be modified to enable the rewrite as well as adding the new **VirtualHost** section for HTTPS.

```
<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>
```

Note

This example is using a locally generated certificate. **SSLCertificateFile** should be your primary certificate file for your domain name. **SSLCertificateKeyFile** should be the key file generated when you created the CSR. **SSLCertificateChainFile** should be the intermediate certificate file (if any) that was supplied by your certificate authority

Save the file, and test the configuration.

```
sudo service httpd configtest
```

Restart Apache.

```
sudo systemctl restart httpd
```

Additional Apache suggestions

Additional Headers

In order to secure against malicious attacks there are a few headers that should either be modified or added. Ensure that the `mod_headers` module is installed.

```
sudo yum install mod_headers
```

Secure Apache from clickjacking

Clickjacking is a malicious technique to collect an infected user's clicks. Clickjacking tricks the victim (visitor) into clicking on an infected site. Use X-FRAME-OPTIONS to secure your site.

Edit the httpd.conf file.

```
sudo nano /etc/httpd/conf/httpd.conf
```

Add the the line `Header append X-FRAME-OPTIONS "SAMEORIGIN"` and save the file, then restart Apache.

MIME-type sniffing

This header prevents Internet Explorer from MIME-sniffing a response away from the declared content-type as the header instructs the browser not to override the response content type. With the nosniff option, if the server says the content is text/html, the browser will render it as text/html.

Edit the httpd.conf file.

```
sudo nano /etc/httpd/conf/httpd.conf
```

Add the the line `Header set X-Content-Type-Options "nosniff"` and save the file, then restart Apache.

Load Balancing

This example shows how to setup and configure Apache on CentOS 7 and Kestrel on the same instance machine. However, in order to not have a single point of failure; using *mod_proxy_balancer* and modifying the VirtualHost would allow for managing mutliple instances of the web applications behind the Apache proxy server.

```
sudo yum install mod_proxy_balancer
```

In the configuration file, an additional instance of the `hellomvc` app has been setup to run on port 5001 and the *Proxy* section has been set with a balancer configuration with two members to load balance *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/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>
```

Rate Limits

Using `mod_ratelimit`, which is included in the `htttpd` module you can limit the amount of bandwidth of clients.

```
sudo nano /etc/httpd/conf.d/ratelimit.conf
```

The example file limits bandwidth as 600 KB/sec under the root location.

```
<IfModule mod_ratelimit.c>
    <Location />
        SetOutputFilter RATE_LIMIT
        SetEnv rate-limit 600
    </Location>
</IfModule>
```

Host ASP.NET Core in Docker containers

The following articles are available for learning about hosting ASP.NET Core apps in Docker.

Building Docker Images for .NET Core Applications
Visual Studio Tools for Docker
Publish to a Docker Image

Publish an ASP.NET Core web app to Azure App Service using Visual Studio

By Rick Anderson and Cesar Blum Silveira

Set up the development environment

Install the latest Azure SDK for Visual Studio. The SDK installs Visual Studio if you don't already have it.

Note

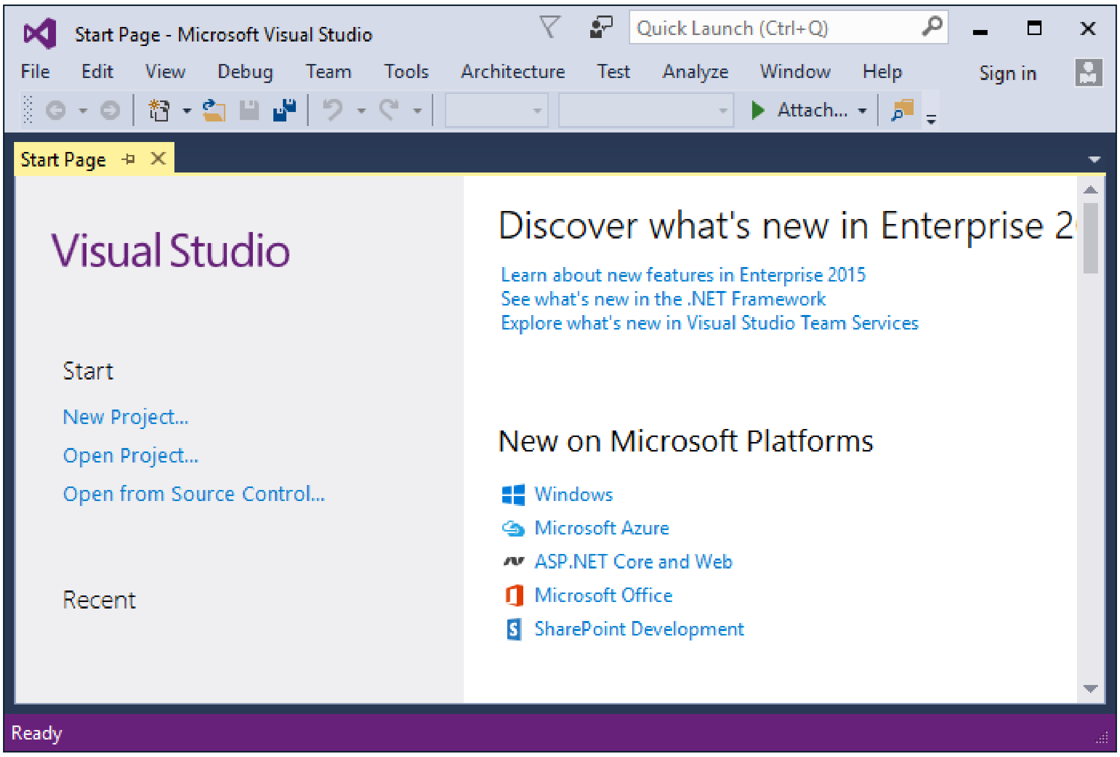
The SDK installation can take more than 30 minutes if your machine doesn't have many of the dependencies.

Install .NET Core + Visual Studio tooling

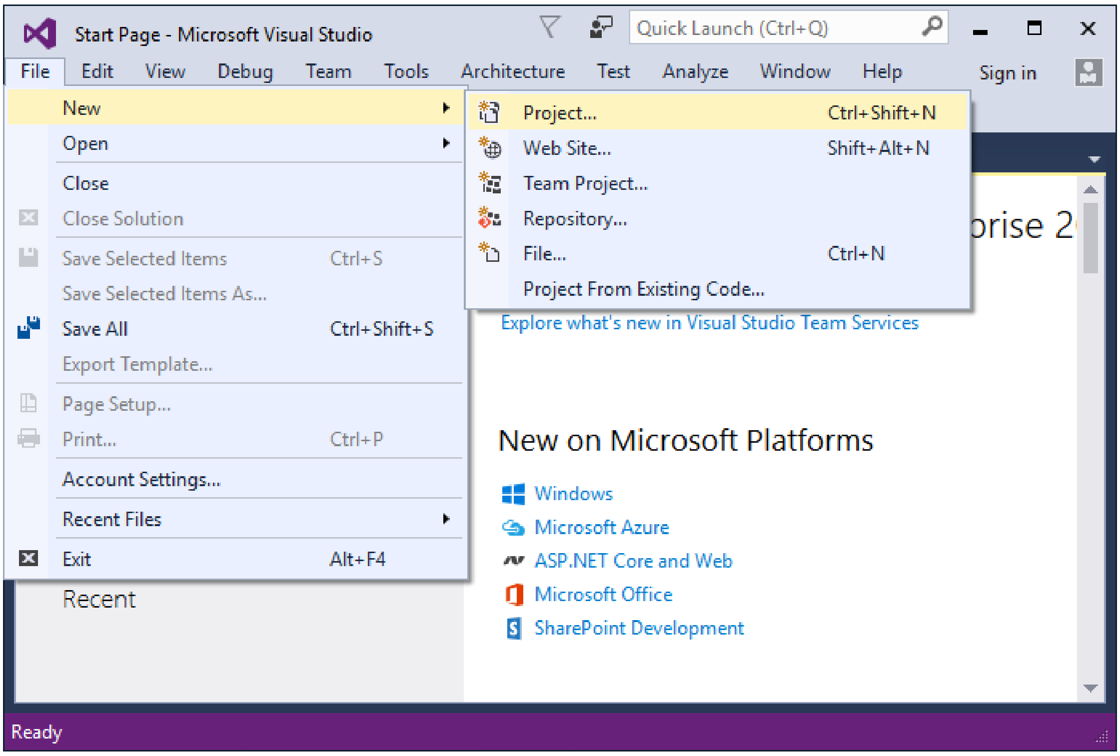
Verify your Azure account. You can open a free Azure account or Activate Visual Studio subscriber benefits.

Create a web app

In the Visual Studio Start Page, tap **New Project...**.



Alternatively, you can use the menus to create a new project. Tap **File > New > Project...**.

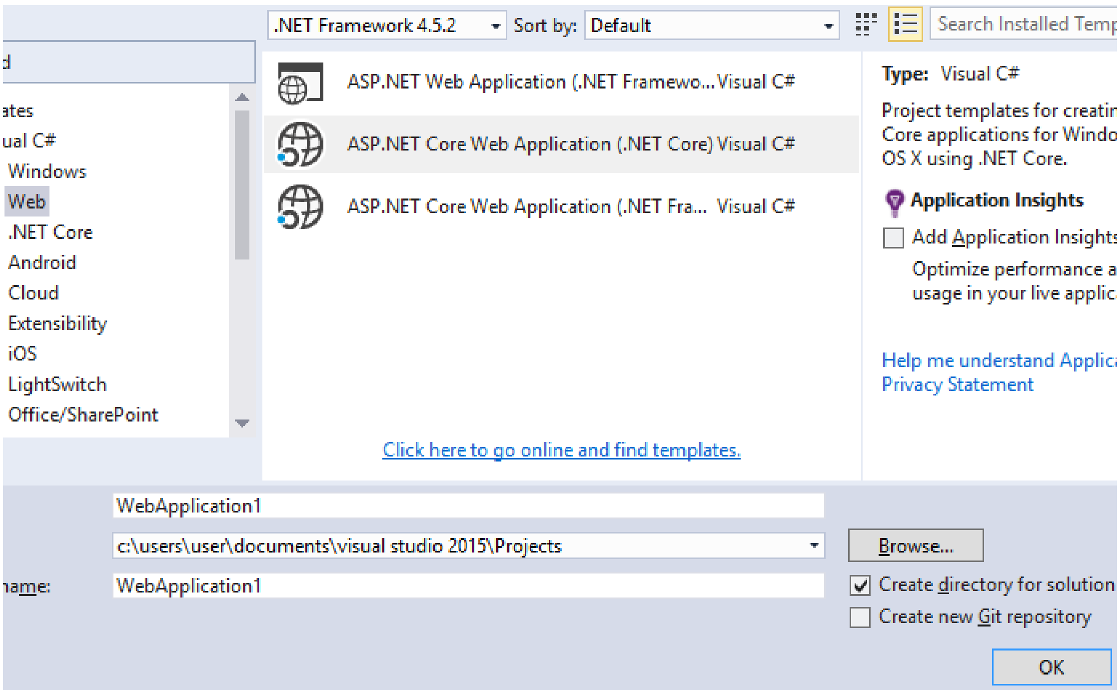


Complete the **New Project** dialog:

In the left pane, tap **Web**

In the center pane, tap **ASP.NET Core Web Application (.NET Core)**

Tap **OK**



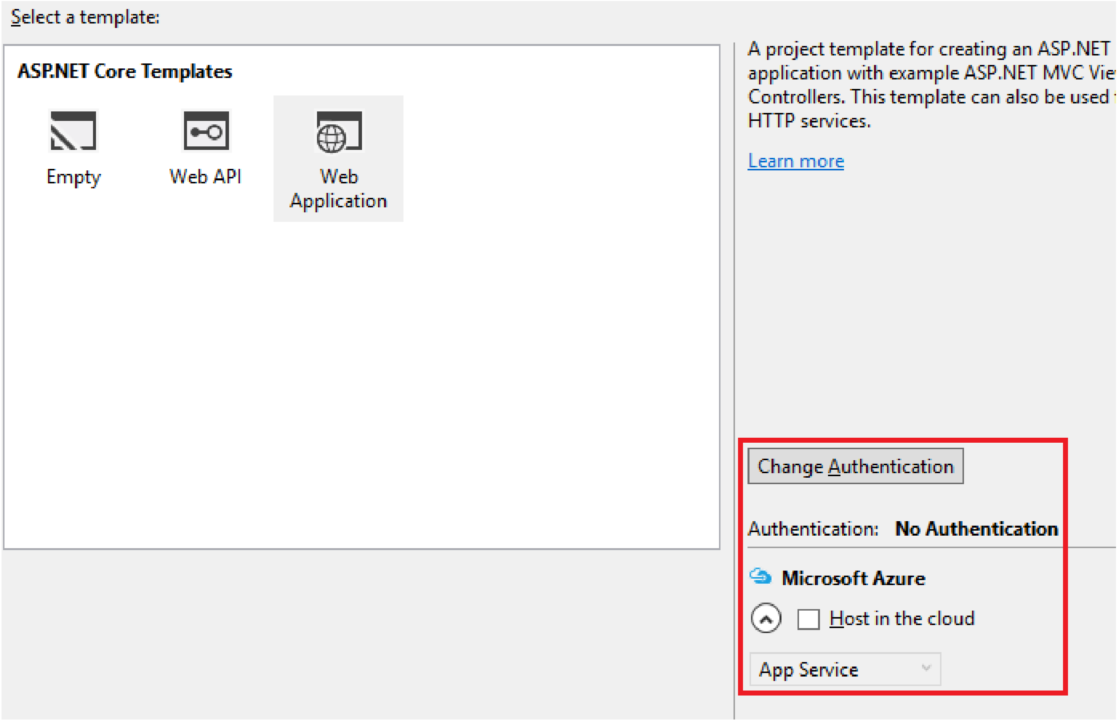
In the **New ASP.NET Core Web Application (.NET Core)** dialog:

Tap **Web Application**

Verify **Authentication** is set to **Individual User Accounts**

Verify **Host in the cloud** is **not** checked

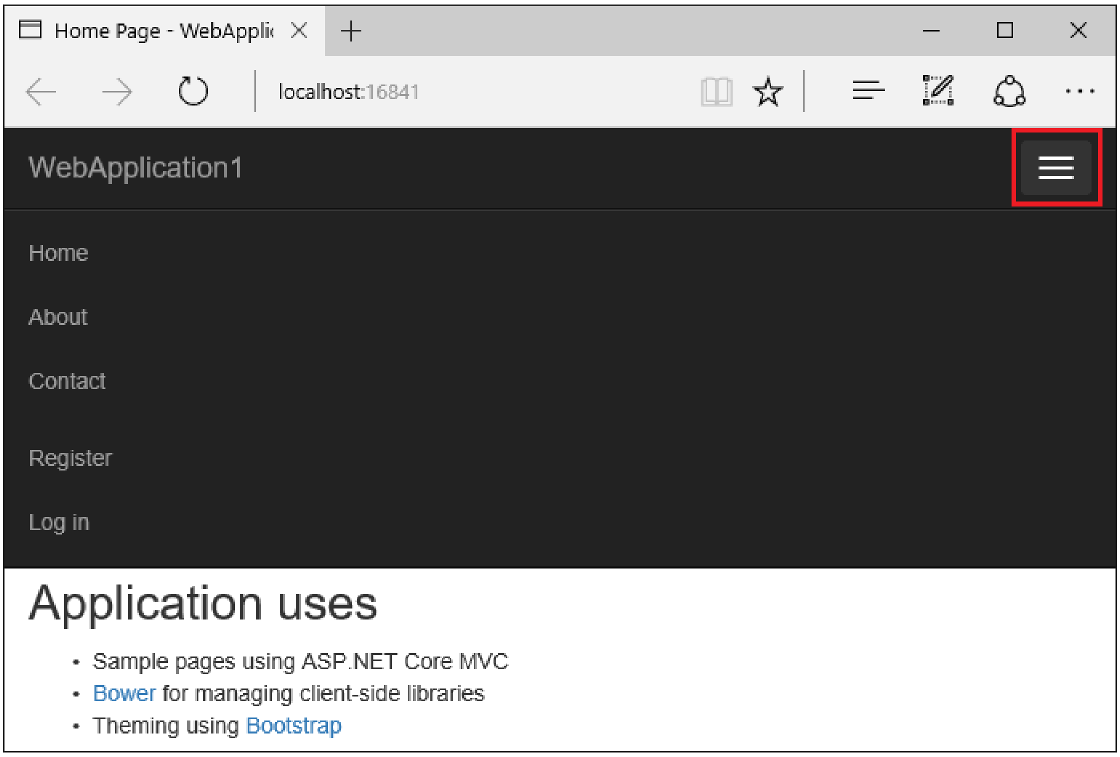
Tap **OK**



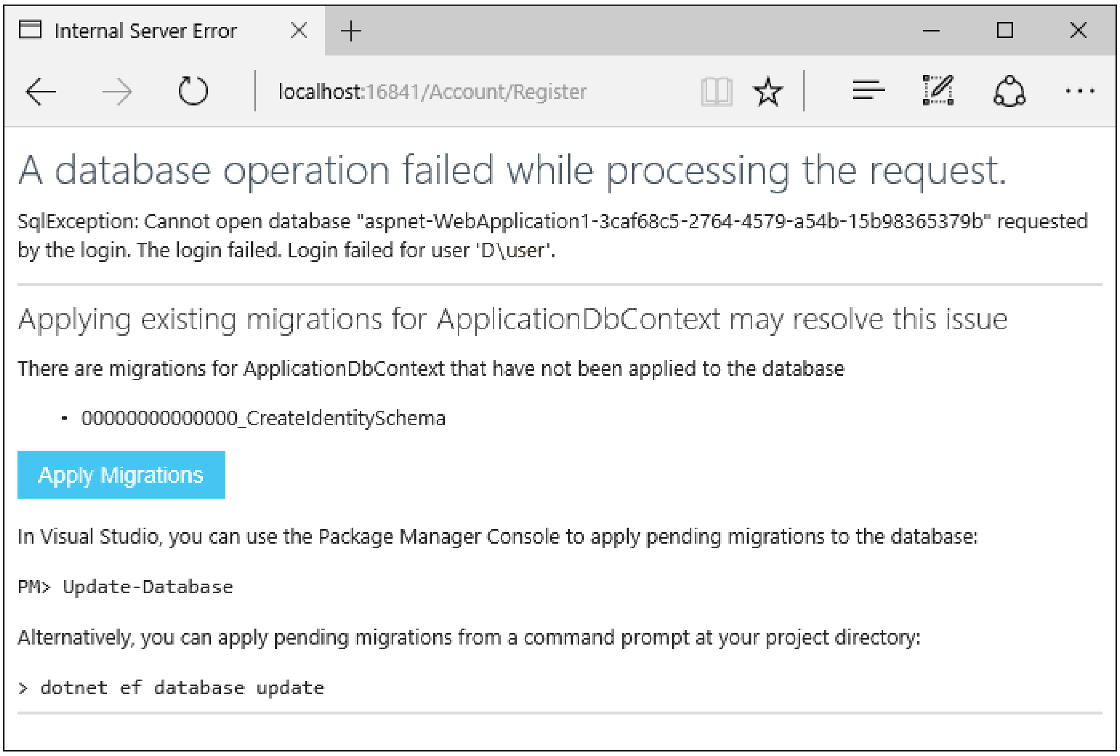
Test the app locally

Press **Ctrl-F5** to run the app locally

Tap the **About** and **Contact** links. Depending on the size of your device, you might need to tap the navigation icon to show the links



Tap **Register** and register a new user. You can use a fictitious email address. When you submit, you'll get the following error:



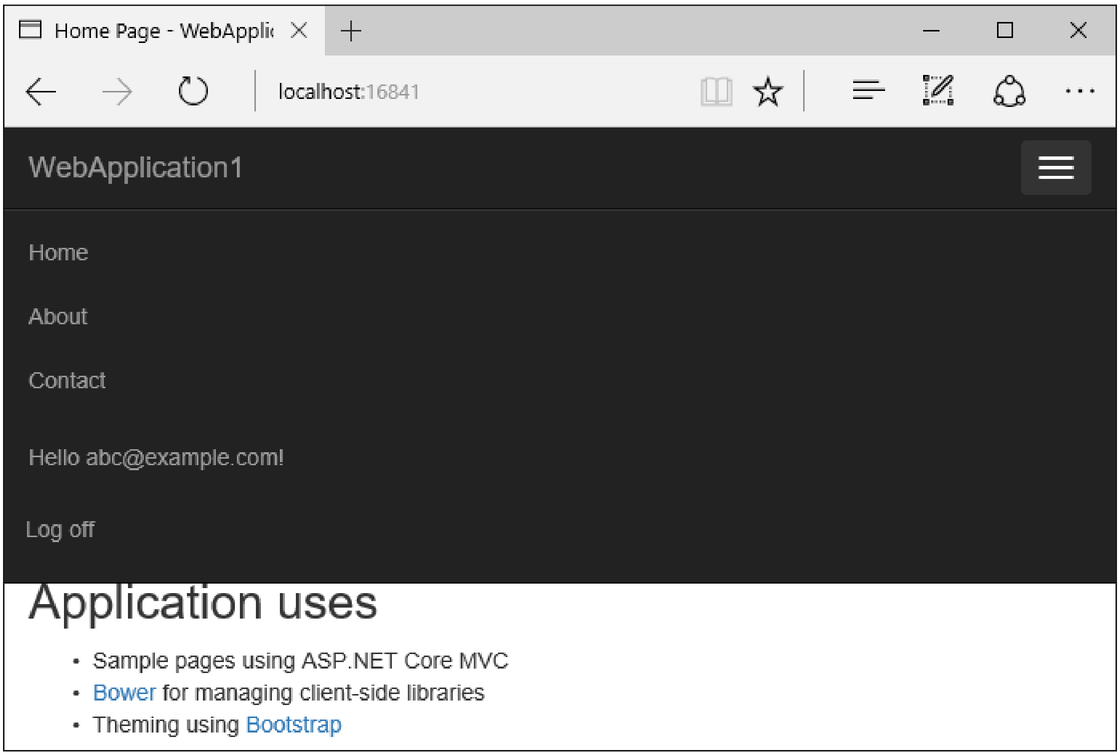
You can fix the problem in two different ways:

Tap **Apply Migrations** and, once the page updates, refresh the page; or

Run the following from a command prompt in the project's directory:

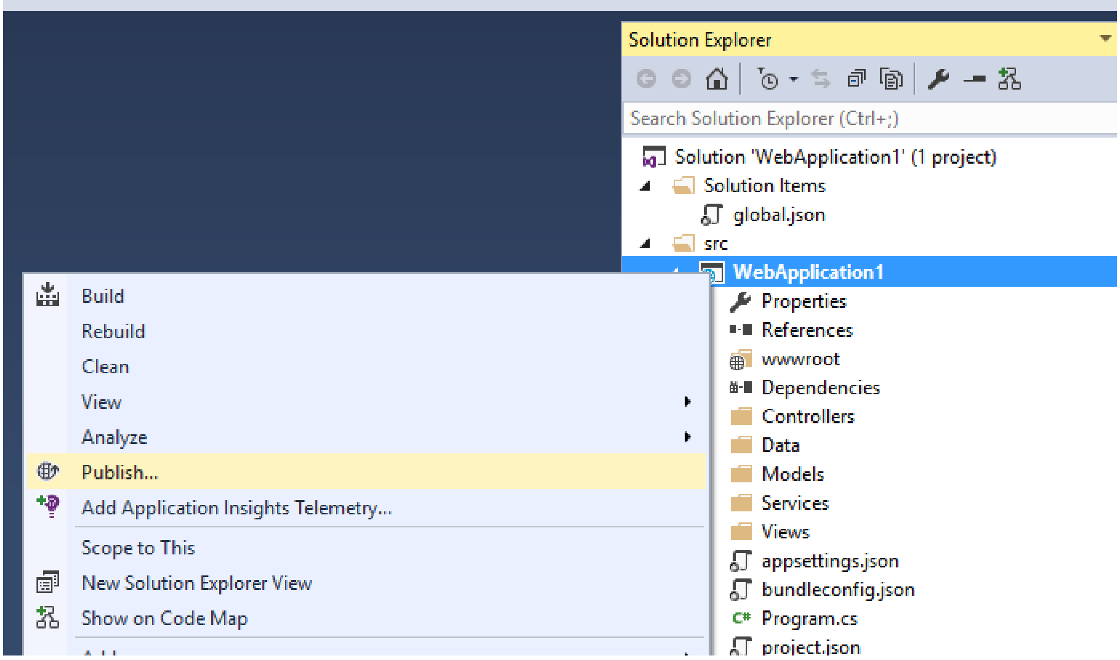
```
dotnet ef database update
```

The app displays the email used to register the new user and a **Log off** link.

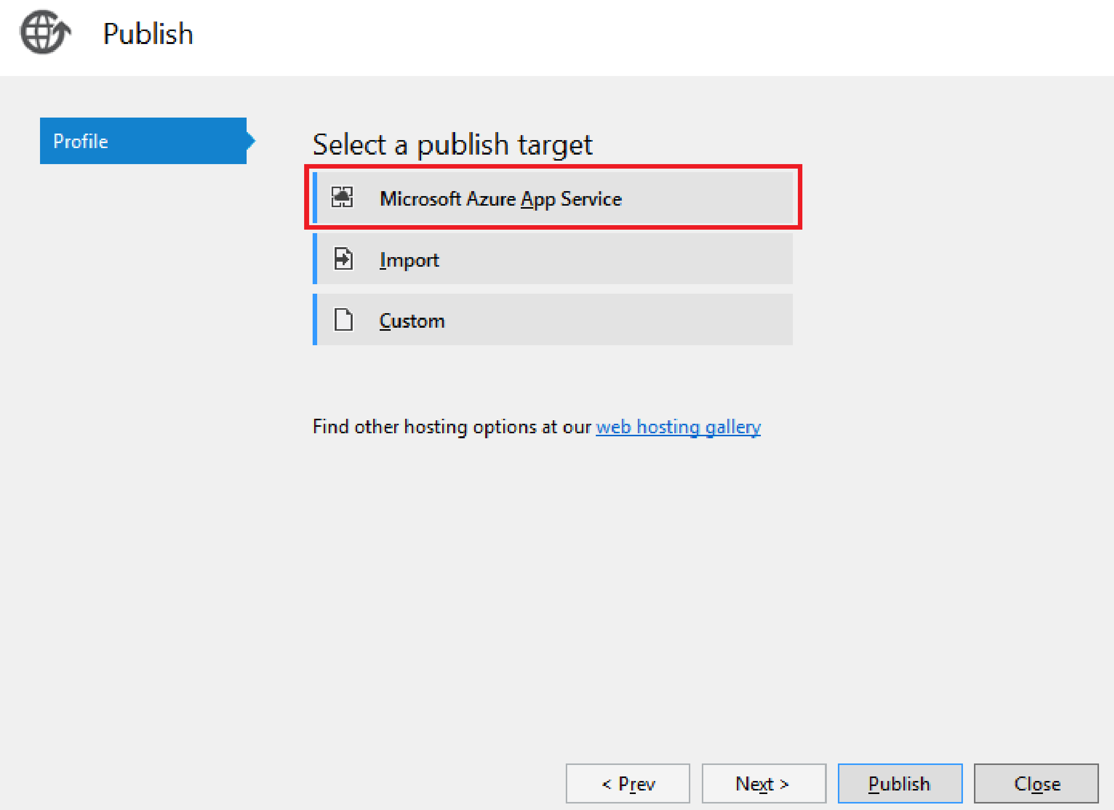


Deploy the app to Azure

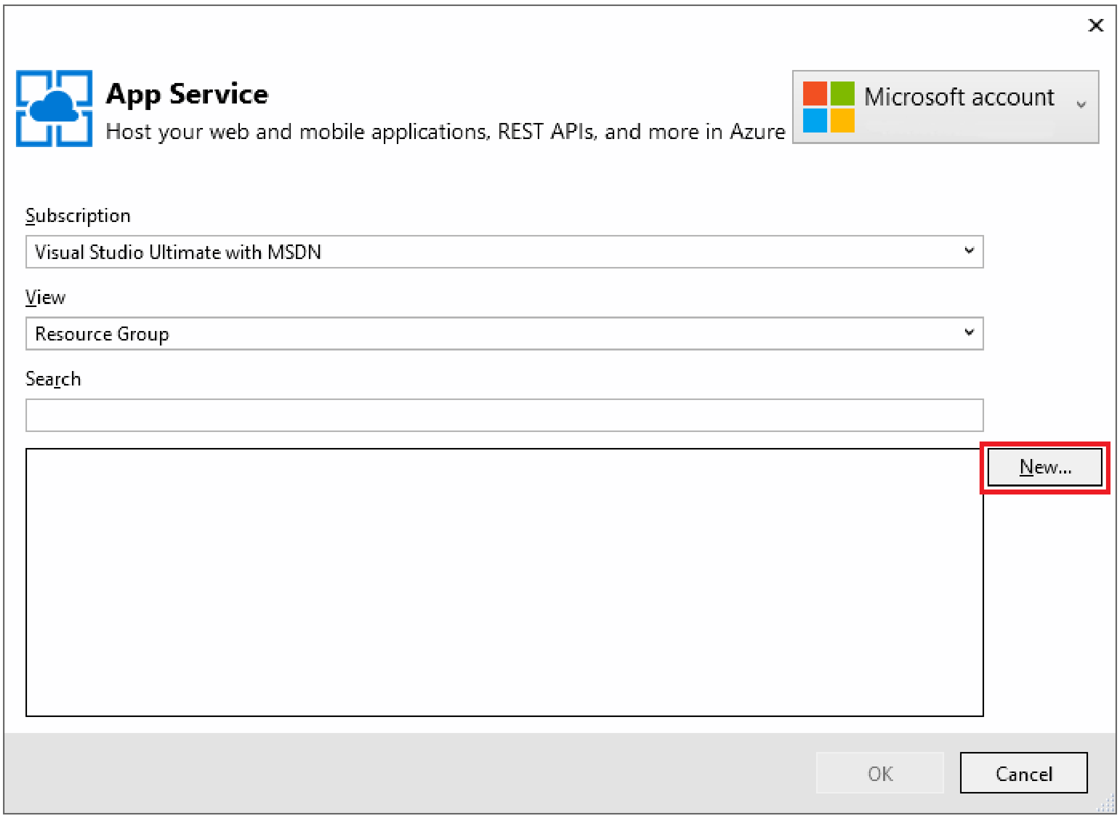
Right-click on the project in Solution Explorer and select **Publish...**.



In the **Publish** dialog, tap **Microsoft Azure App Service**.



Tap **New...** to create a new resource group. Creating a new resource group will make it easier to delete all the Azure resources you create in this tutorial.

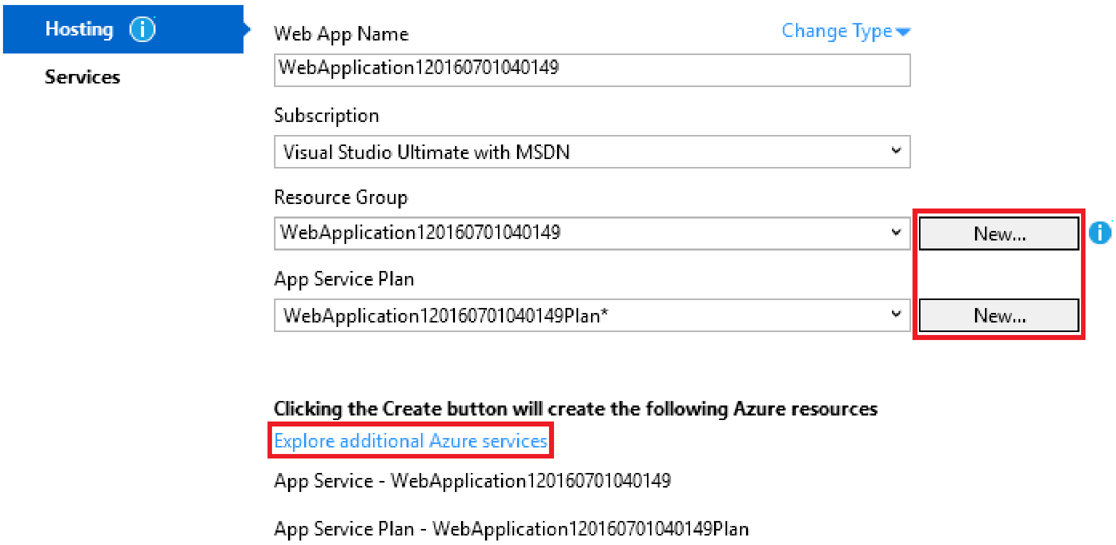


Create a new resource group and app service plan:

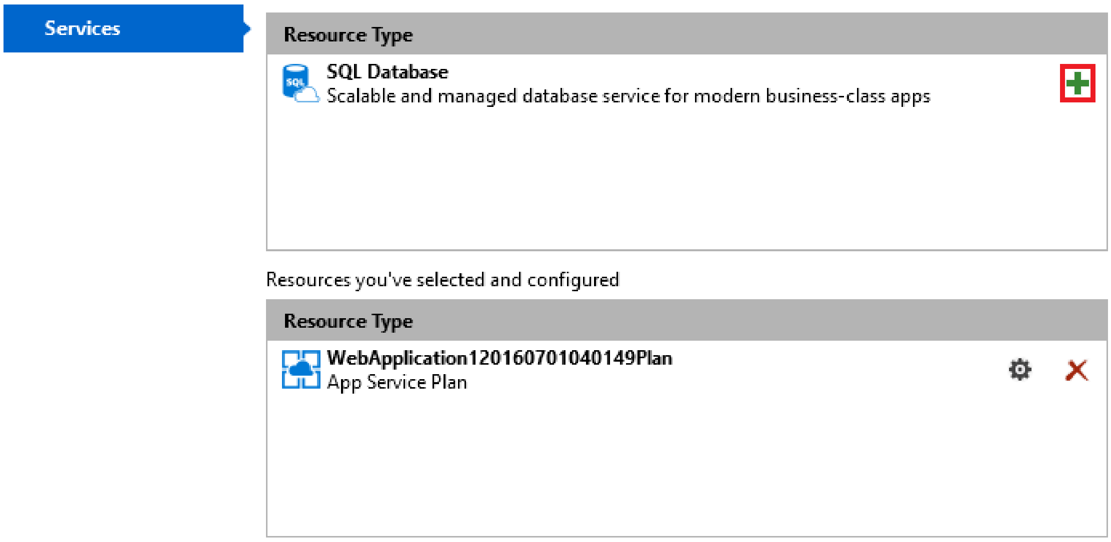
Tap **New...** for the resource group and enter a name for the new resource group

Tap **New...** for the app service plan and select a location near you. You can keep the default generated name

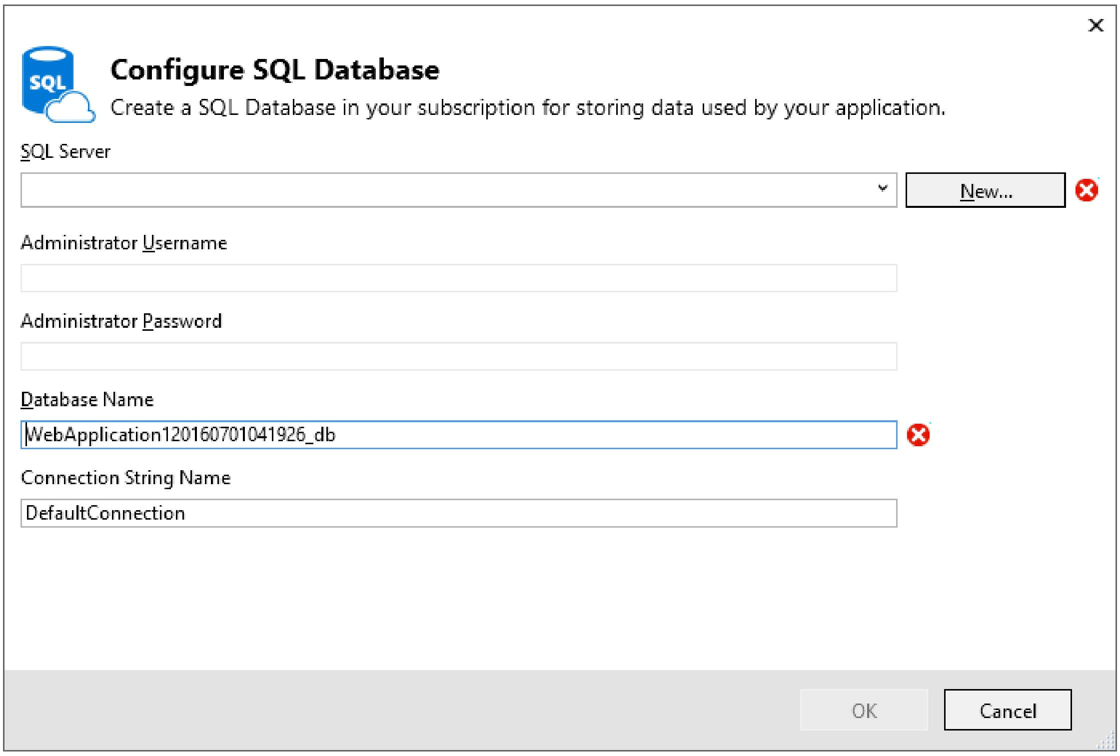
Tap **Explore additional Azure services** to create a new database



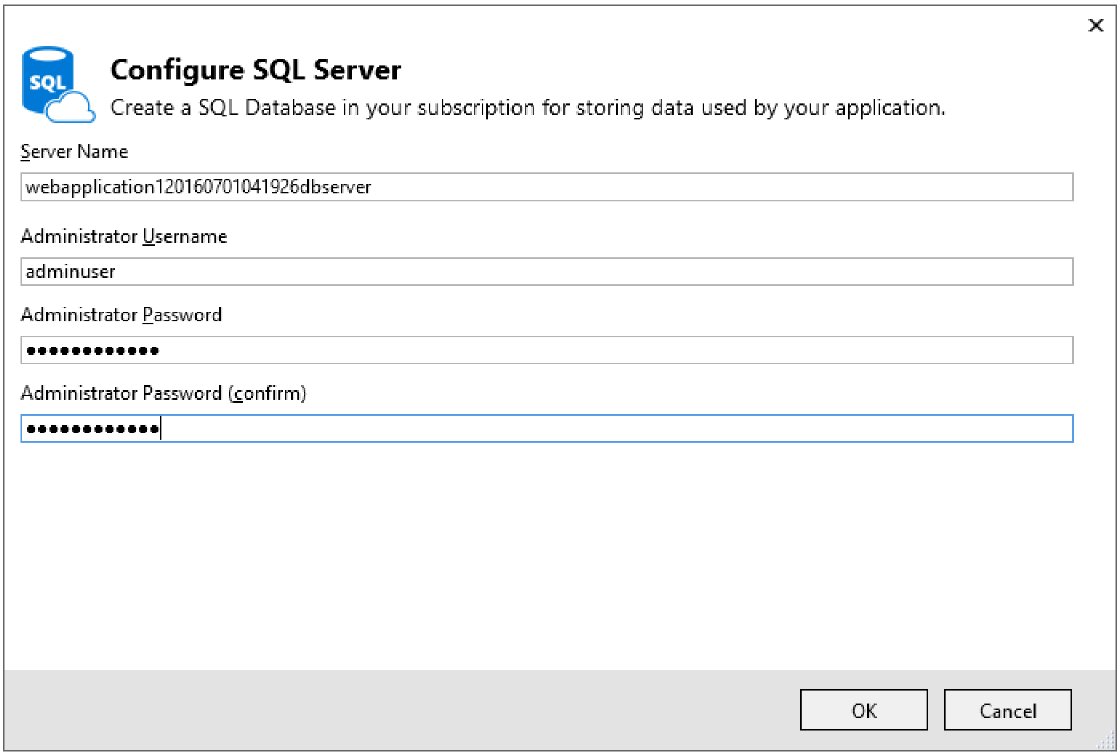
Tap the green **+** icon to create a new SQL Database



Tap **New...** on the **Configure SQL Database** dialog to create a new database server.



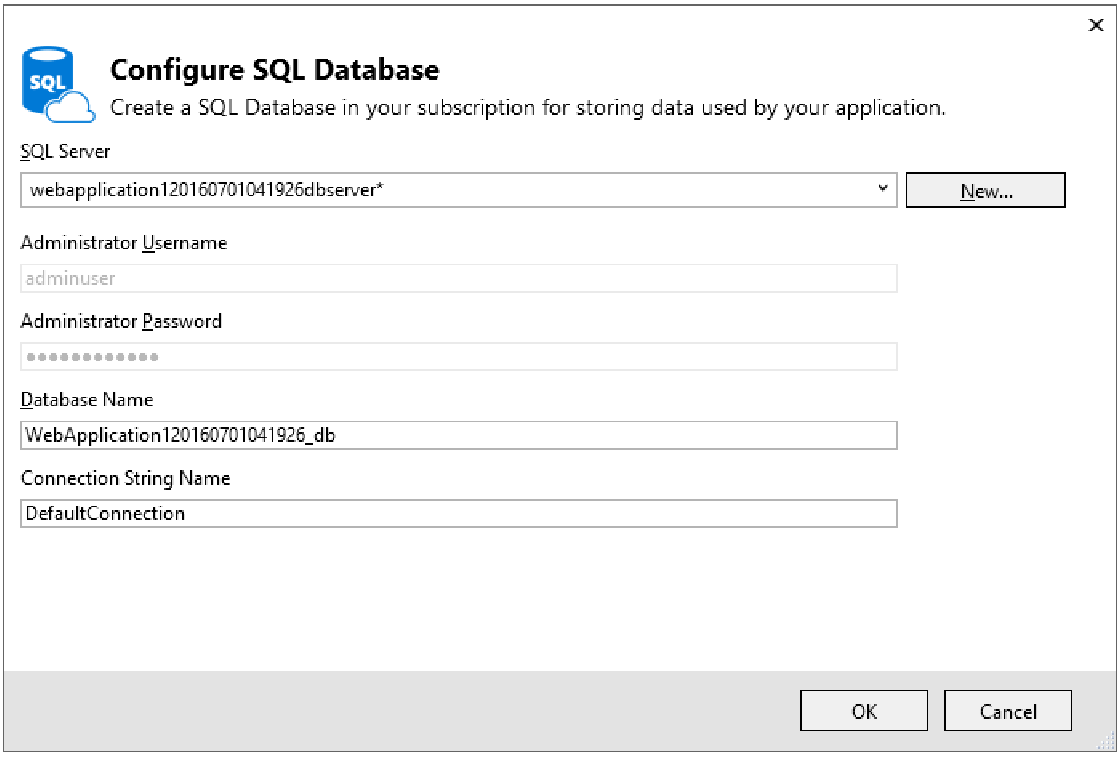
Enter an administrator user name and password, and then tap **OK**. Don't forget the user name and password you create in this step. You can keep the default **Server Name**



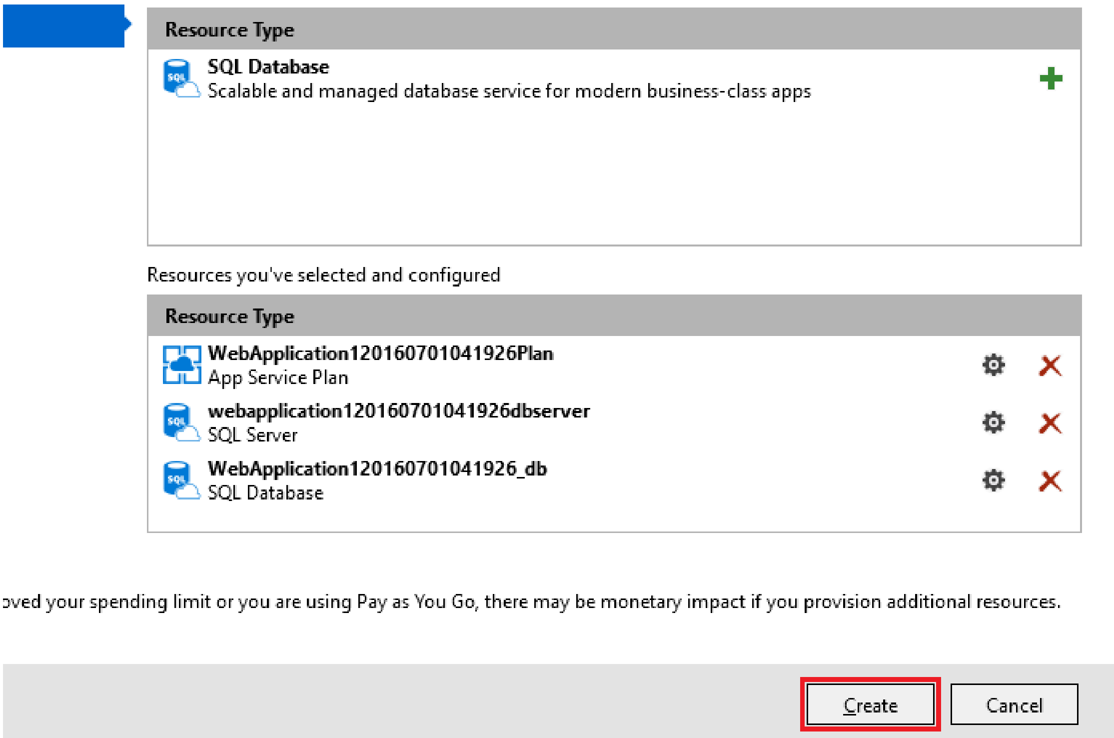
Note

"admin" is not allowed as the administrator user name.

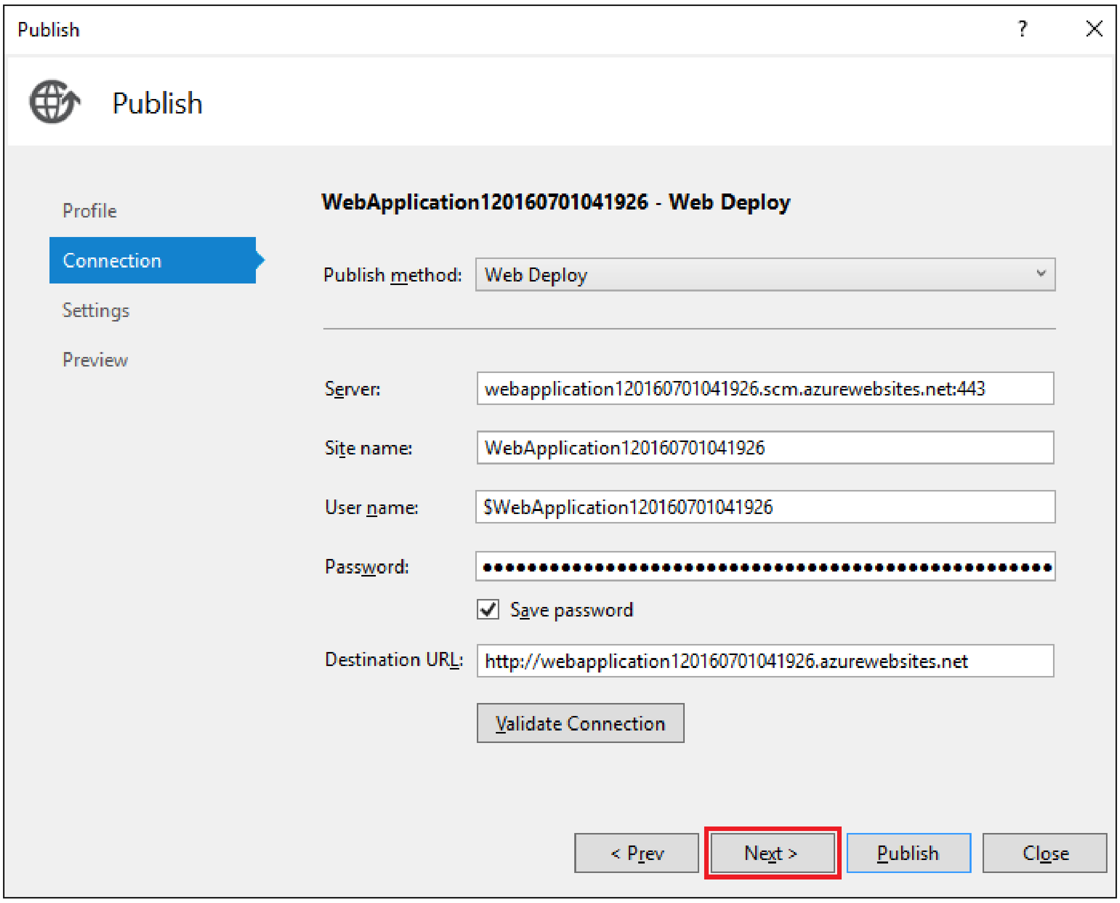
Tap **OK** on the **Configure SQL Database** dialog



Tap **Create** on the **Create App Service** dialog



Tap **Next** in the **Publish** dialog

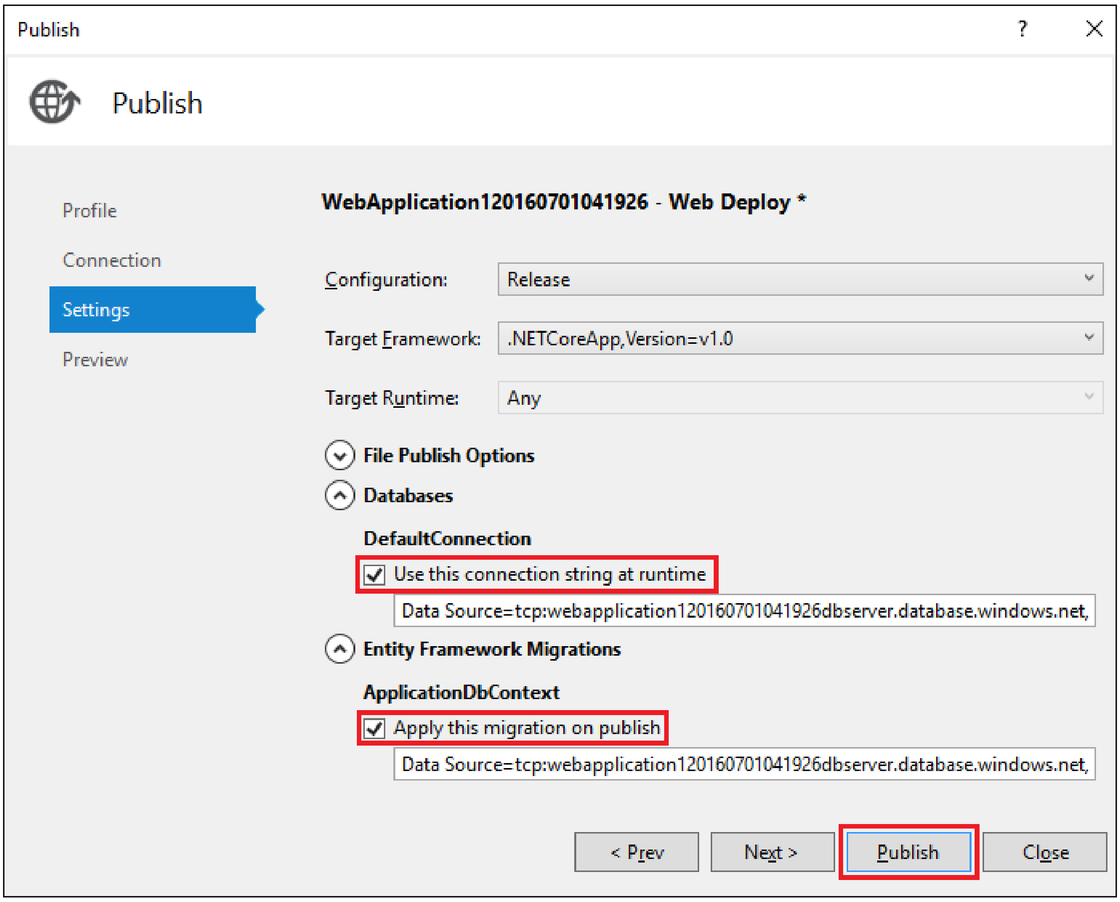


On the **Settings** stage of the **Publish** dialog:

Expand **Databases** and check **Use this connection string at runtime**

Expand **Entity Framework Migrations** and check **Apply this migration on publish**

Tap **Publish** and wait until Visual Studio finishes publishing your app

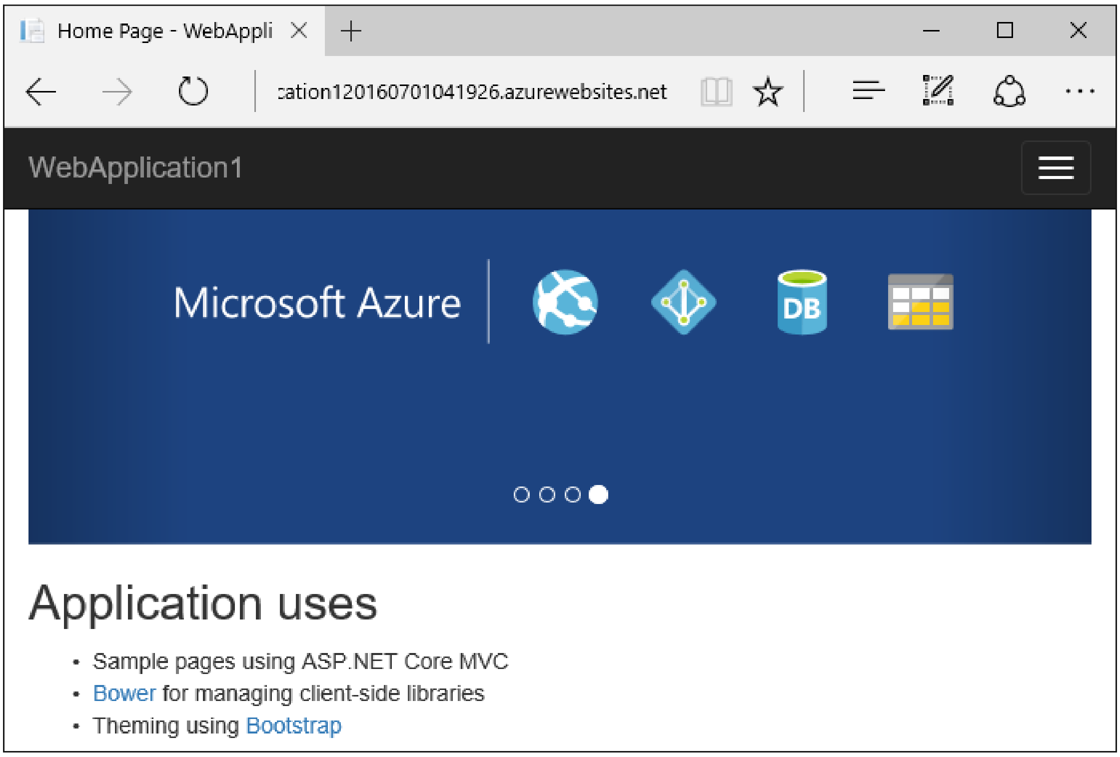


Visual Studio will publish your app to Azure and launch the cloud app in your browser.

Test your app in Azure

Test the **About** and **Contact** links

Register a new user

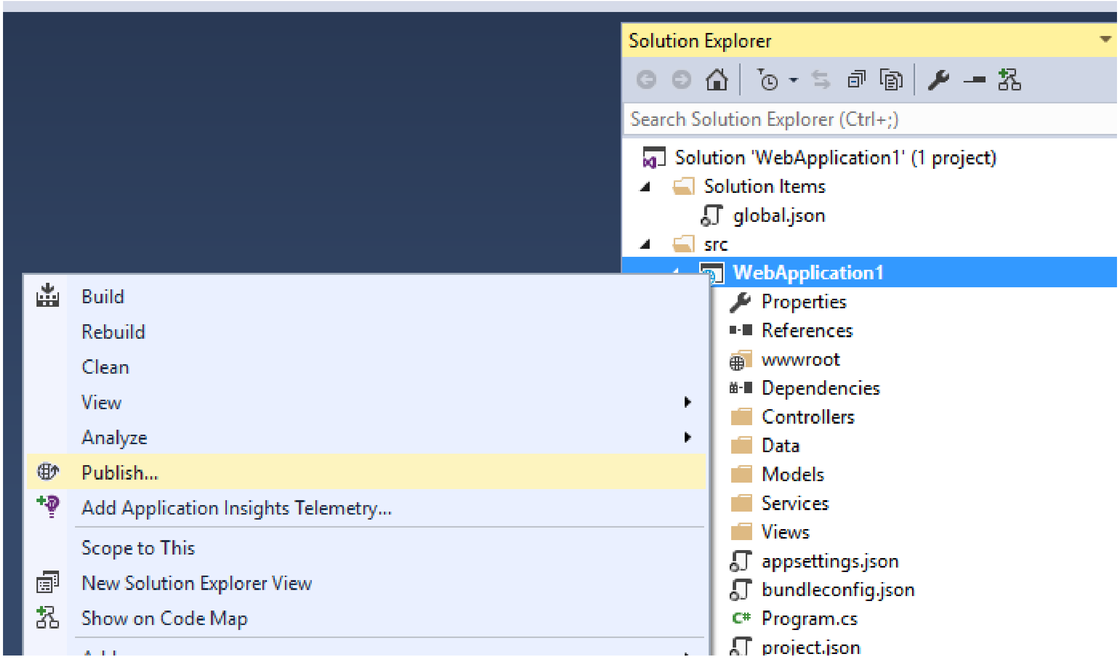


Update the app

Edit the `Views/Home/About.cshtml` Razor view file and change its contents. For example:

```
@{
       ViewData["Title"] = "About";
   }
   <h2>@ViewData["Title"].</h2>
   <h3>@ViewData["Message"]</h3>

   <p>My updated about page.</p>
```

Right-click on the project and tap **Publish...** again

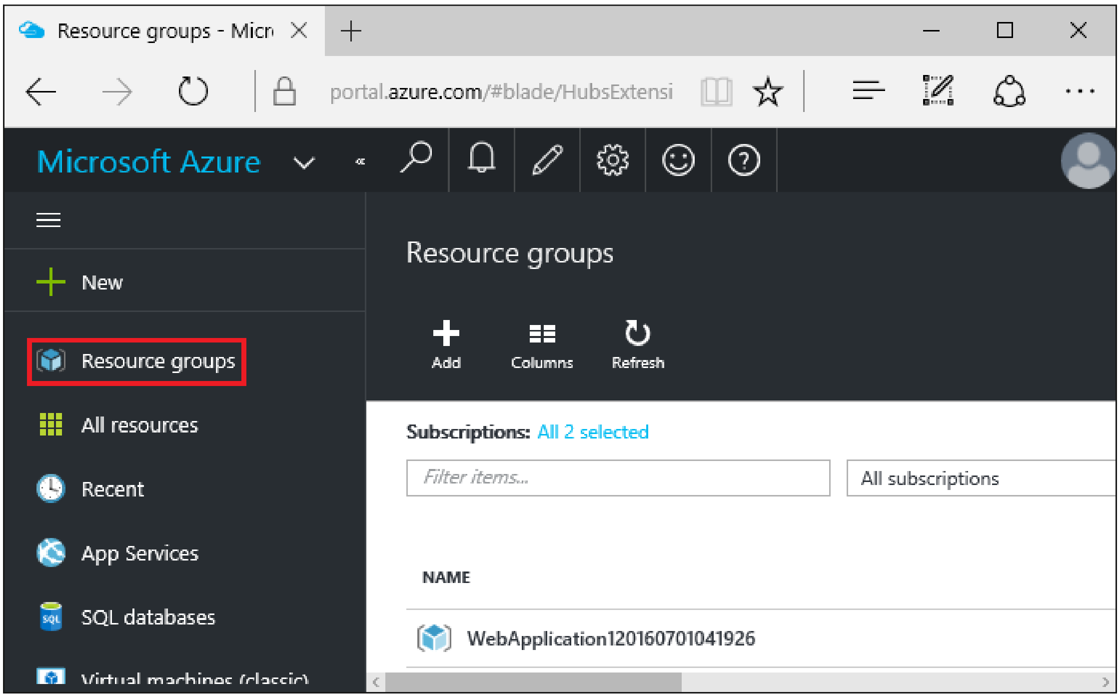


After the app is published, verify the changes you made are available on Azure

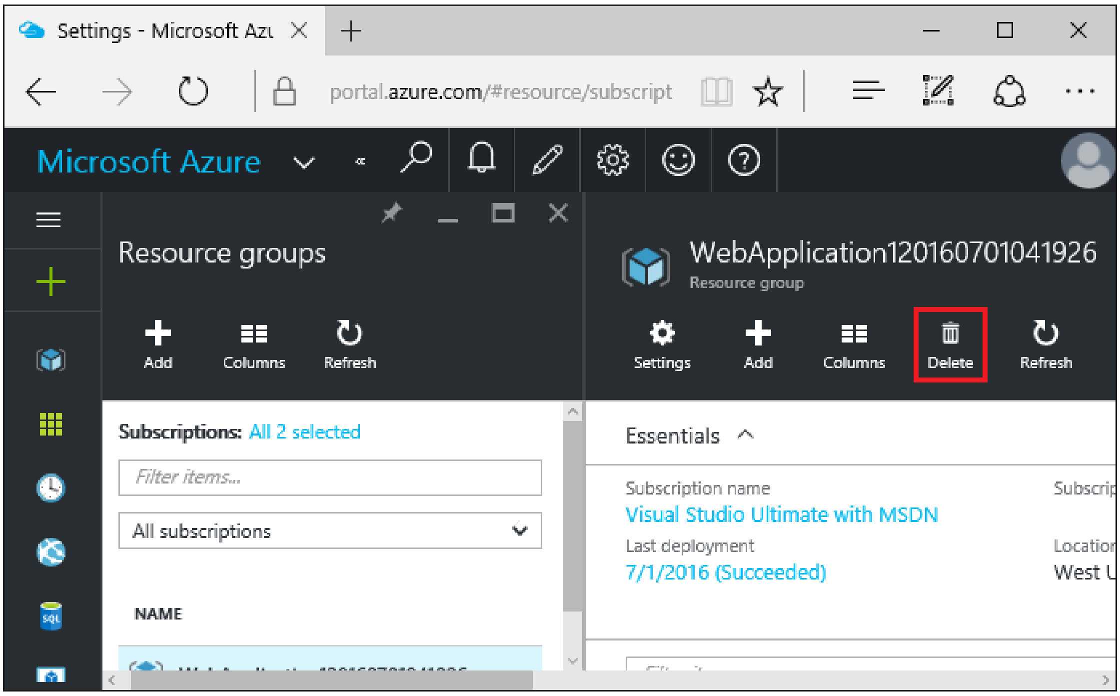
Clean up

When you have finished testing the app, go to the Azure portal and delete the app.

Select **Resource groups**, then tap the resource group you created



In the **Resource group** blade, tap **Delete**



Enter the name of the resource group and tap **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure

Next steps

Getting started with ASP.NET Core MVC and Visual Studio

Introduction to ASP.NET Core

Fundamentals

Continuous deployment to Azure for ASP.NET Core, with Visual Studio and Git

By Erik Reitan

This tutorial shows you how to create an ASP.NET Core web app using Visual Studio and deploy it from Visual Studio to Azure App Service using continuous deployment.

See also Use VSTS to Build and Publish to an Azure Web App with Continuous Deployment, which shows how to configure a continuous delivery (CD) workflow for Azure App Service using Visual Studio Team Services. Azure Continuous Delivery in Team Services simplifies setting up a robust deployment pipeline to publish updates for your app to Azure App Service. The pipeline can be configured from the Azure portal to build, run tests, deploy to a staging slot, and then deploy to production.

Note

To complete this tutorial, you need a Microsoft Azure account. If you don't have an account, you can activate your MSDN subscriber benefits or sign up for a free trial.

Prerequisites

This tutorial assumes you have already installed the following:

Visual Studio

ASP.NET Core (runtime and tooling)

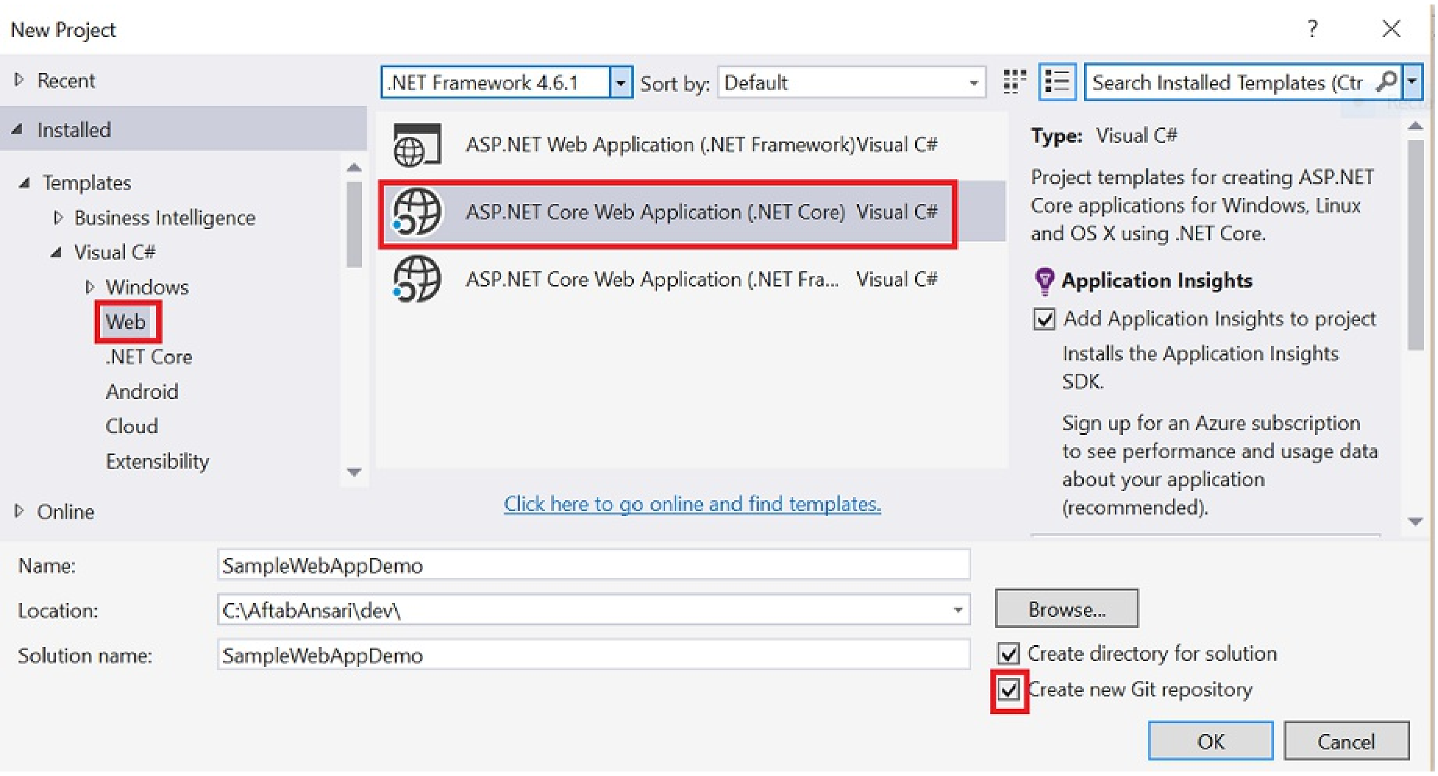
Git for Windows

Create an ASP.NET Core web app

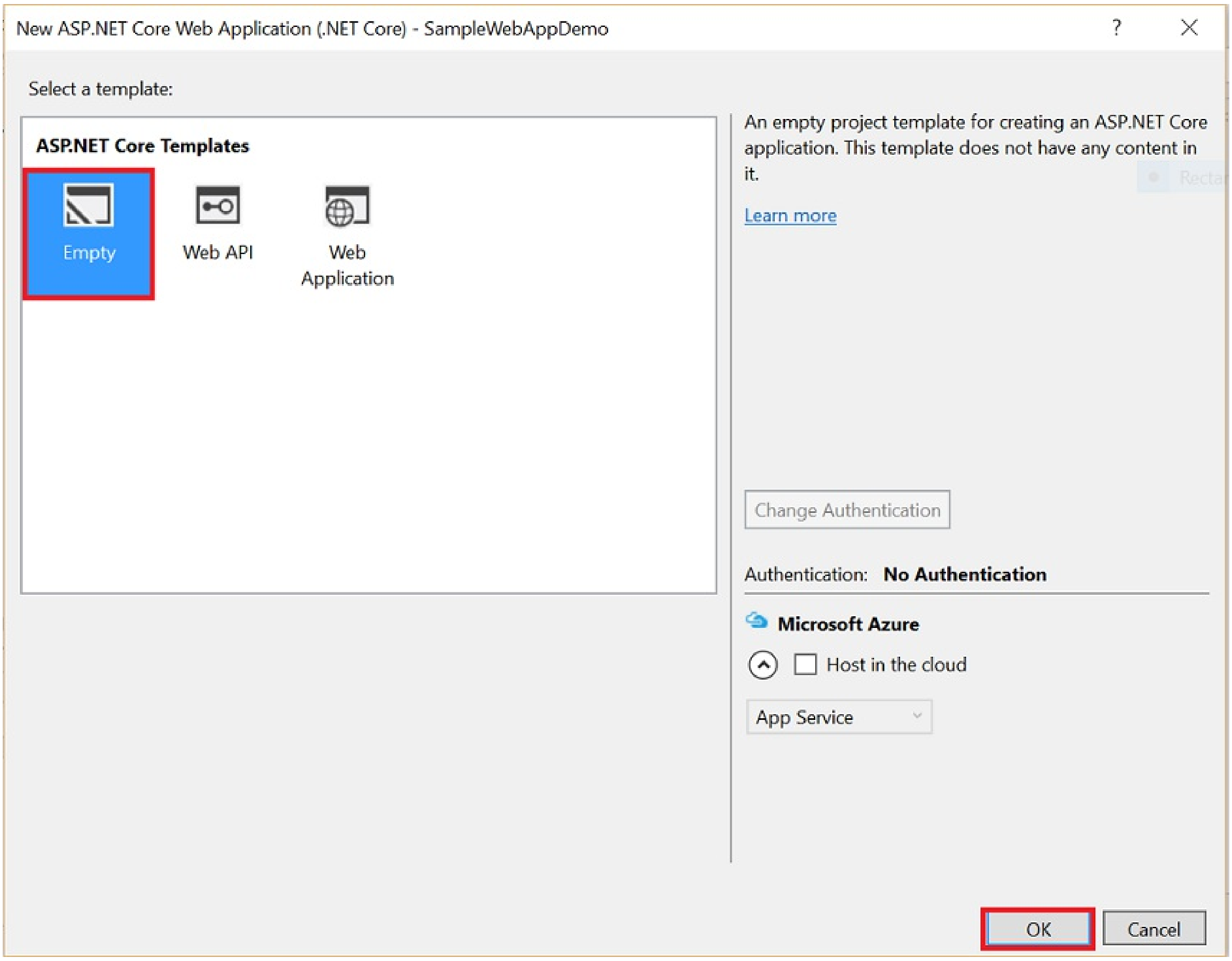
Start Visual Studio.

From the **File** menu, select **New** > **Project**.

Select the **ASP.NET Web Application** project template. It appears under **Installed** > **Templates** > **Visual C#** > **Web**. Name the project `SampleWebAppDemo`. Select the **Create new Git respository** option and click **OK**.



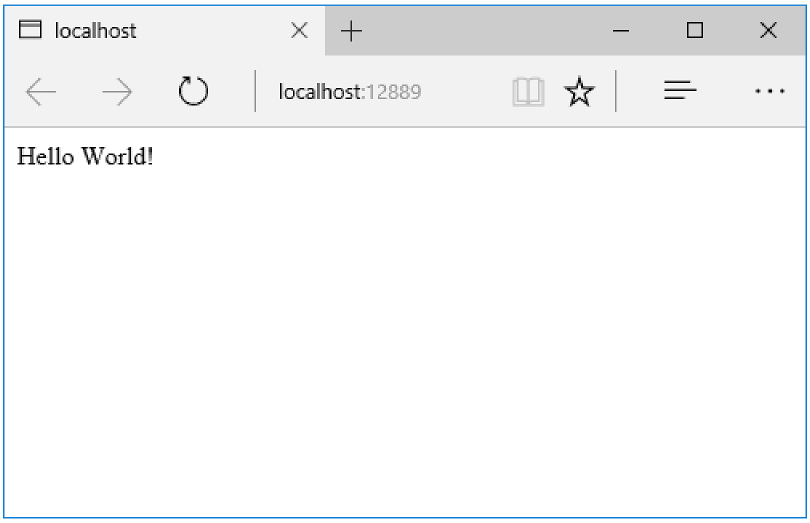
In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template, then click **OK**.



Running the web app locally

Once Visual Studio finishes creating the app, run the app by selecting **Debug** -> **Start Debugging**. As an alternative, you can press **F5**.

It may take time to initialize Visual Studio and the new app. Once it is complete, the browser will show the running app.



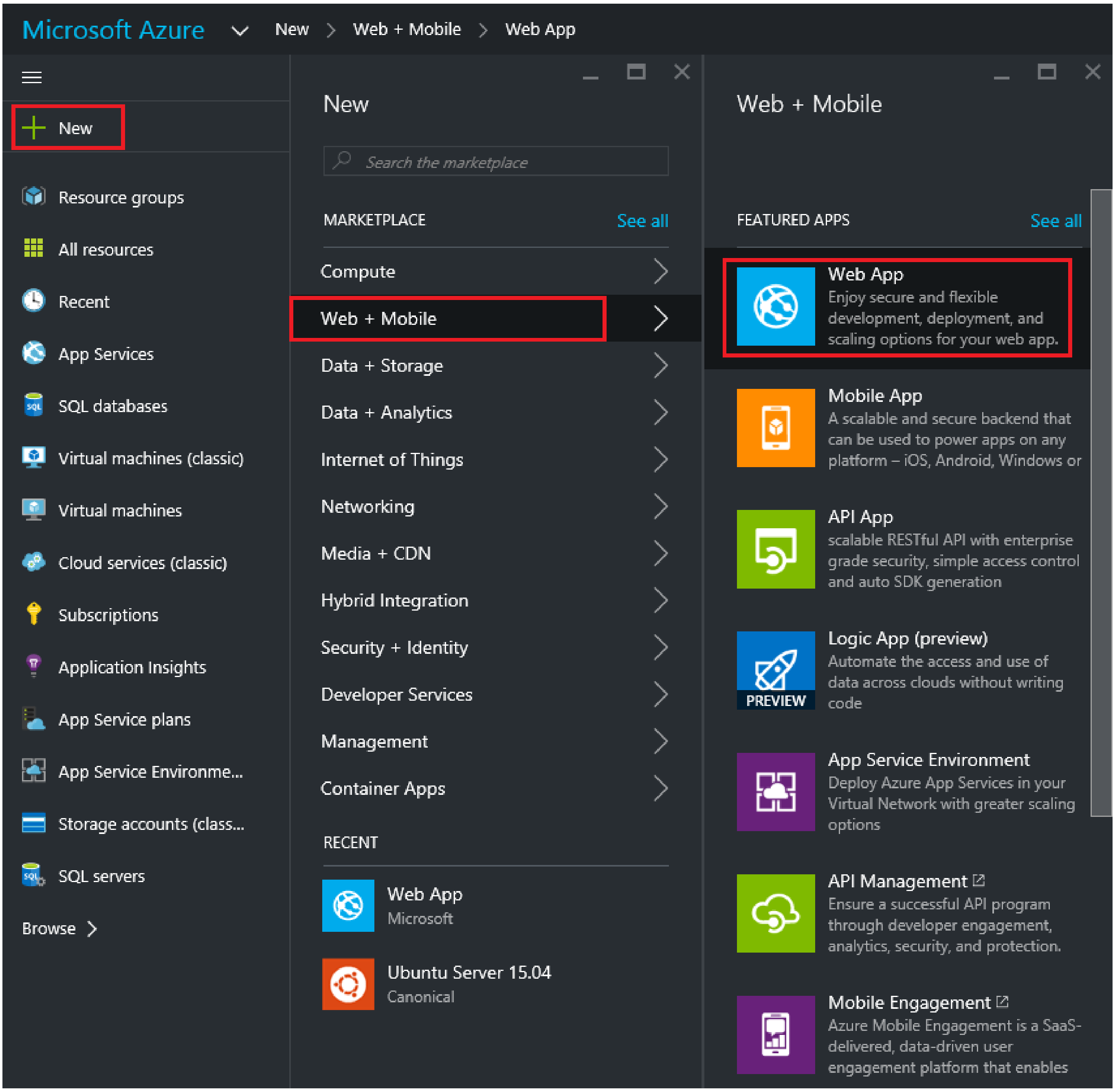
After reviewing the running Web app, close the browser and click the "Stop Debugging" icon in the toolbar of Visual Studio to stop the app.

Create a web app in the Azure Portal

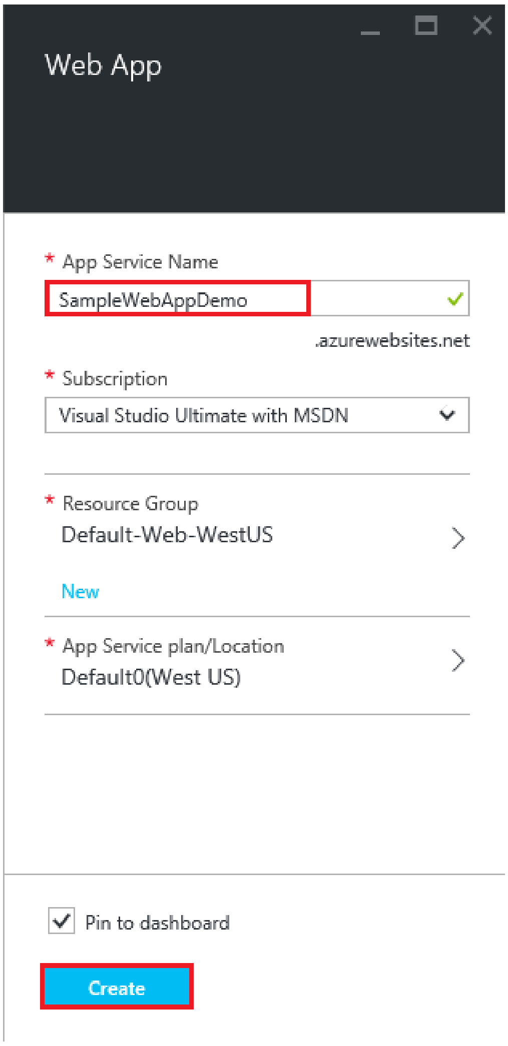
The following steps will guide you through creating a web app in the Azure Portal.

Log in to the Azure Portal
TAP **NEW** at the top left of the Portal

TAP **Web + Mobile** > **Web App**



In the **Web App** blade, enter a unique value for the **App Service Name**.

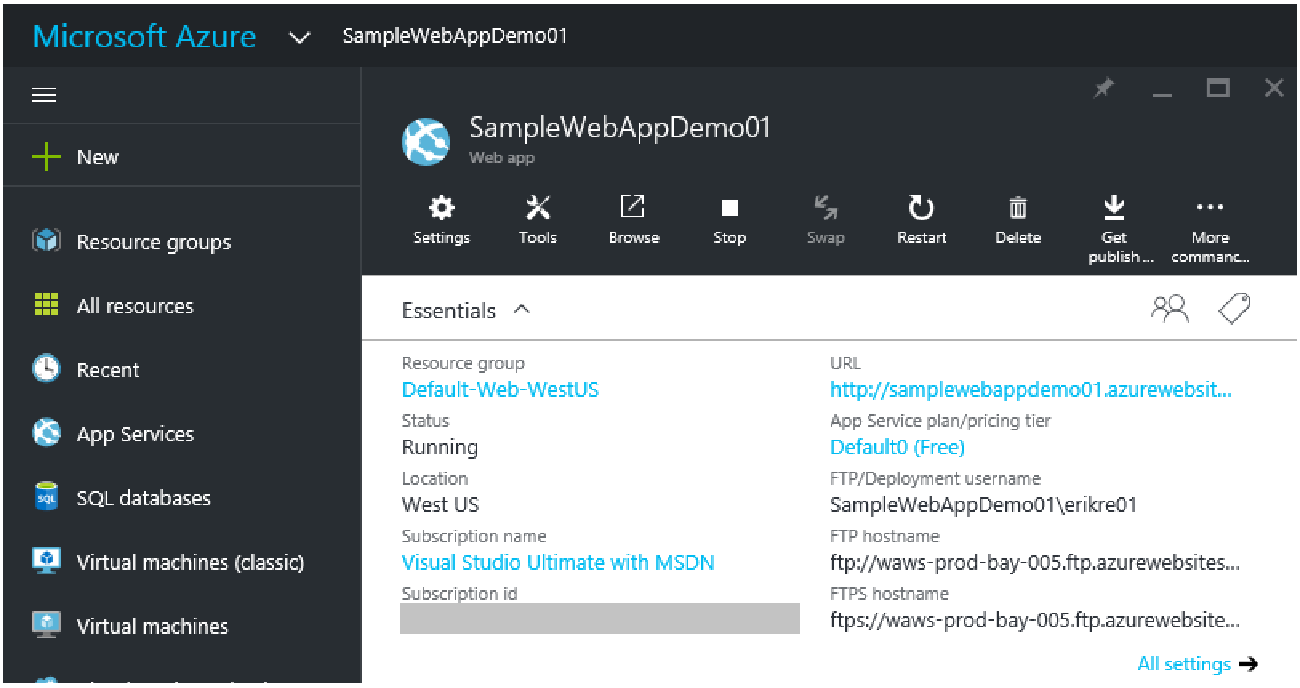


 Note

The **App Service Name** name needs to be unique. The portal will enforce this rule when you attempt to enter the name. After you enter a different value, you'll need to substitute that value for each occurrence of **SampleWebAppDemo** that you see in this tutorial.

Also in the **Web App** blade, select an existing **App Service Plan/Location** or create a new one. If you create a new plan, select the pricing tier, location, and other options. For more information on App Service plans, Azure App Service plans in-depth overview.

Click **Create**. Azure will provision and start your web app.



Enable Git publishing for the new web app

Git is a distributed version control system that you can use to deploy your Azure App Service web app. You'll store the code you write for your web app in a local Git repository, and you'll deploy your code to Azure by pushing to a remote repository.

Log into the Azure Portal, if you're not already logged in.

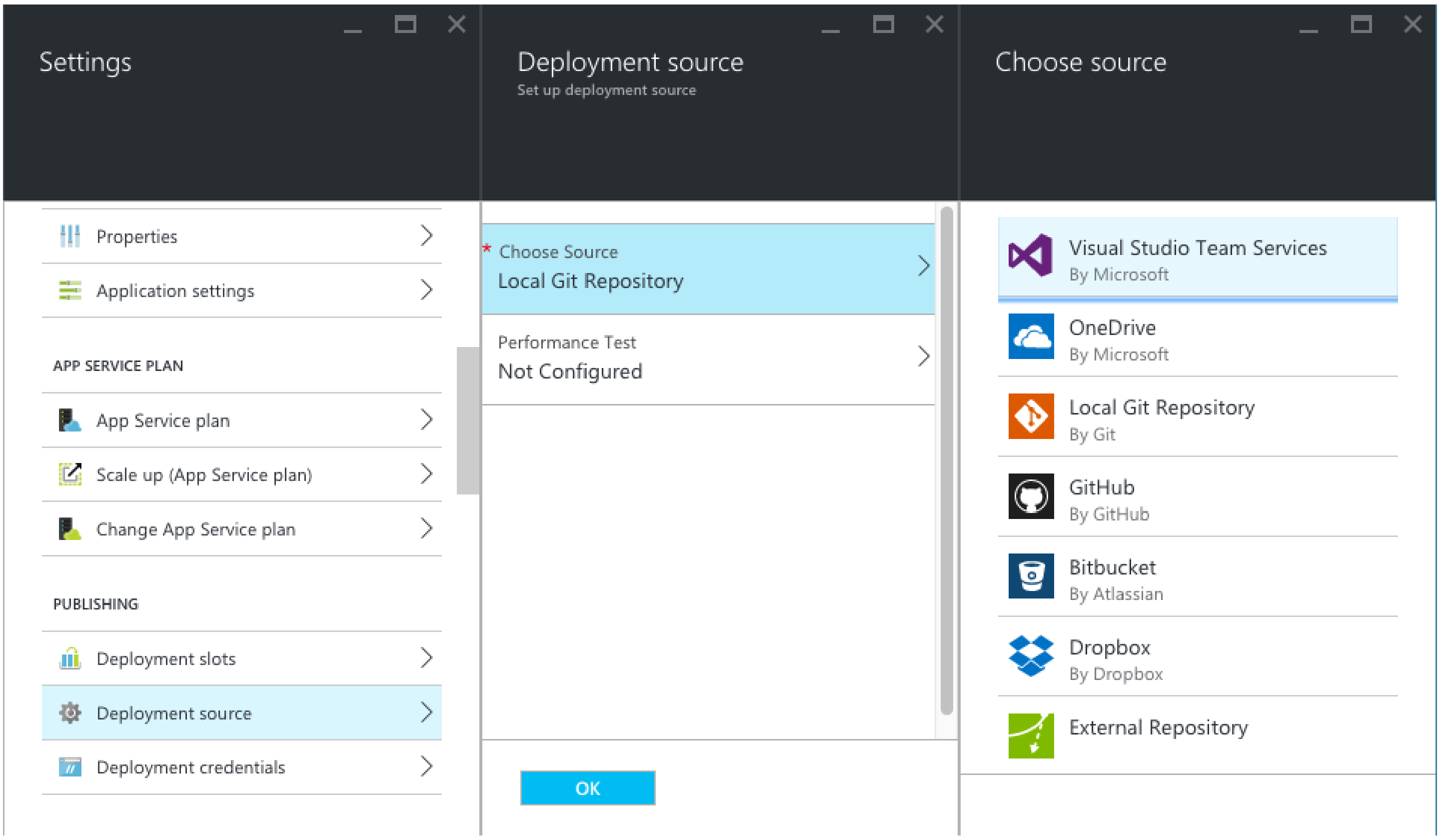
Click **Browse**, located at the bottom of the navigation pane.

Click **Web Apps** to view a list of the web apps associated with your Azure subscription.

Select the web app you created in the previous section of this tutorial.

If the **Settings** blade is not shown, select **Settings** in the **Web App** blade.

In the **Settings** blade, select **Deployment source** > **Choose Source** > **Local Git Repository**.



Click **OK**.

If you have not previously set up deployment credentials for publishing a web app or other App Service app, set them up now:

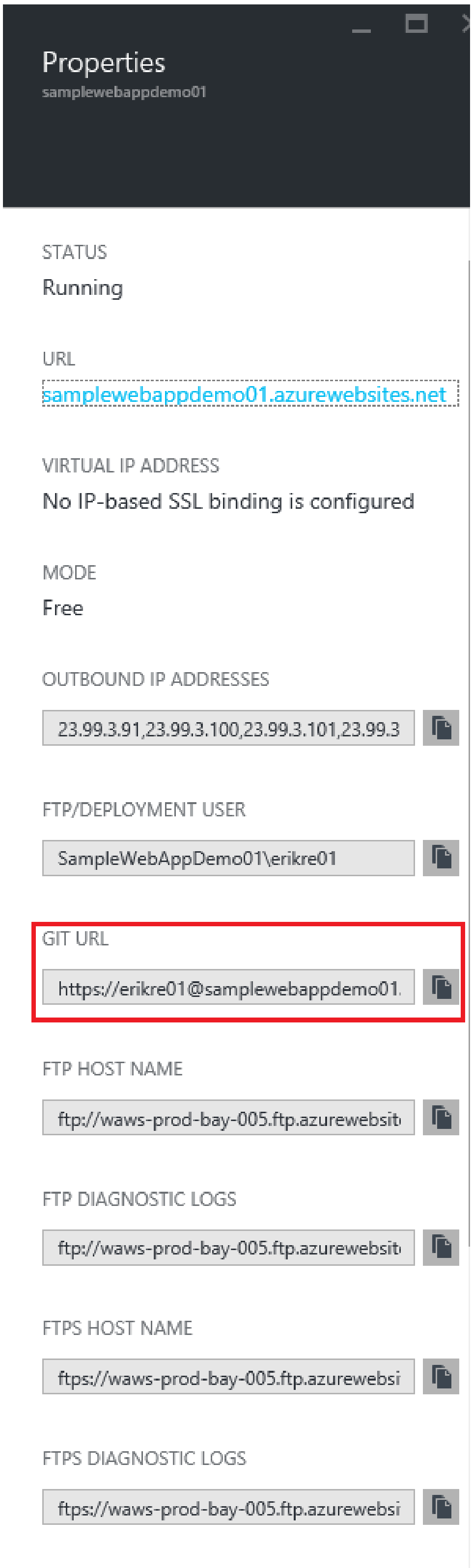
Click **Settings** > **Deployment credentials**. The **Set deployment credentials** blade will be displayed.

Create a user name and password. You'll need this password later when setting up Git.

Click **Save**.

In the **Web App** blade, click **Settings** > **Properties**. The URL of the remote Git repository that you'll deploy to is shown under **GIT URL**.

Copy the **GIT URL** value for later use in the tutorial.



Publish your web app to Azure App Service

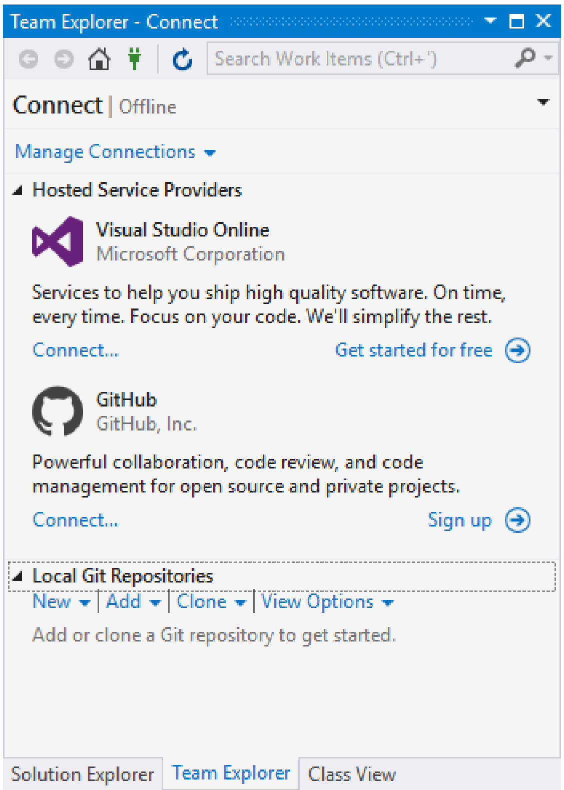
In this section, you will create a local Git repository using Visual Studio and push from that repository to Azure to deploy your web app. The steps involved include the following:

Add the remote repository setting using your GIT URL value, so you can deploy your local repository to Azure.

Commit your project changes.

Push your project changes from your local repository to your remote repository on Azure.

In **Solution Explorer** right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** will be displayed.

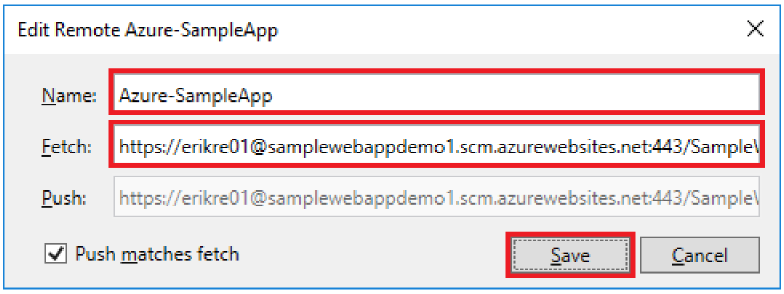


In **Team Explorer**, select the **Home** (home icon) > **Settings** > **Repository Settings**.

In the **Remotes** section of the **Repository Settings** select **Add**. The **Add Remote** dialog box will be displayed.

Set the **Name** of the remote to **Azure-SampleApp**.

Set the value for **Fetch** to the **Git URL** that you copied from Azure earlier in this tutorial. Note that this is the URL that ends with **.git**.



Note

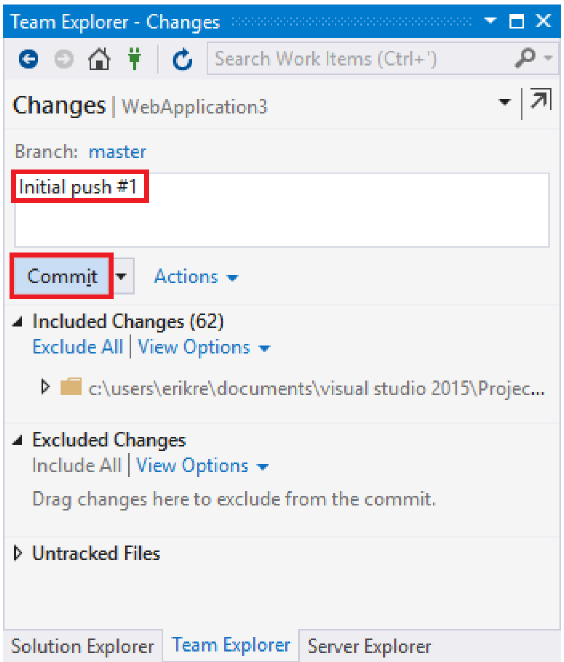
As an alternative, you can specify the remote repository from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering the command. For example:

```
git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp.git
```

Select the **Home** (home icon) > **Settings** > **Global Settings**. Make sure you have your name and your email address set. You may also need to select **Update**.

Select **Home** > **Changes** to return to the **Changes** view.

Enter a commit message, such as **Initial Push #1** and click **Commit**. This action will create a *commit* locally. Next, you need to *sync* with Azure.



 Note

As an alternative, you can commit your changes from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering the git commands. For example:

`git add .`

`git commit -am "Initial Push #1"`

Select **Home** > **Sync** > **Actions** > **Open Command Prompt**. The command prompt will open to your project directory.

Enter the following command in the command window:

`git push -u Azure-SampleApp master`

Enter your Azure **deployment credentials** password that you created earlier in Azure.

 Note

Your password will not be visible as you enter it.

This command will start the process of pushing your local project files to Azure. The output from the above command ends with a message that deployment was successful.

```
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.
```

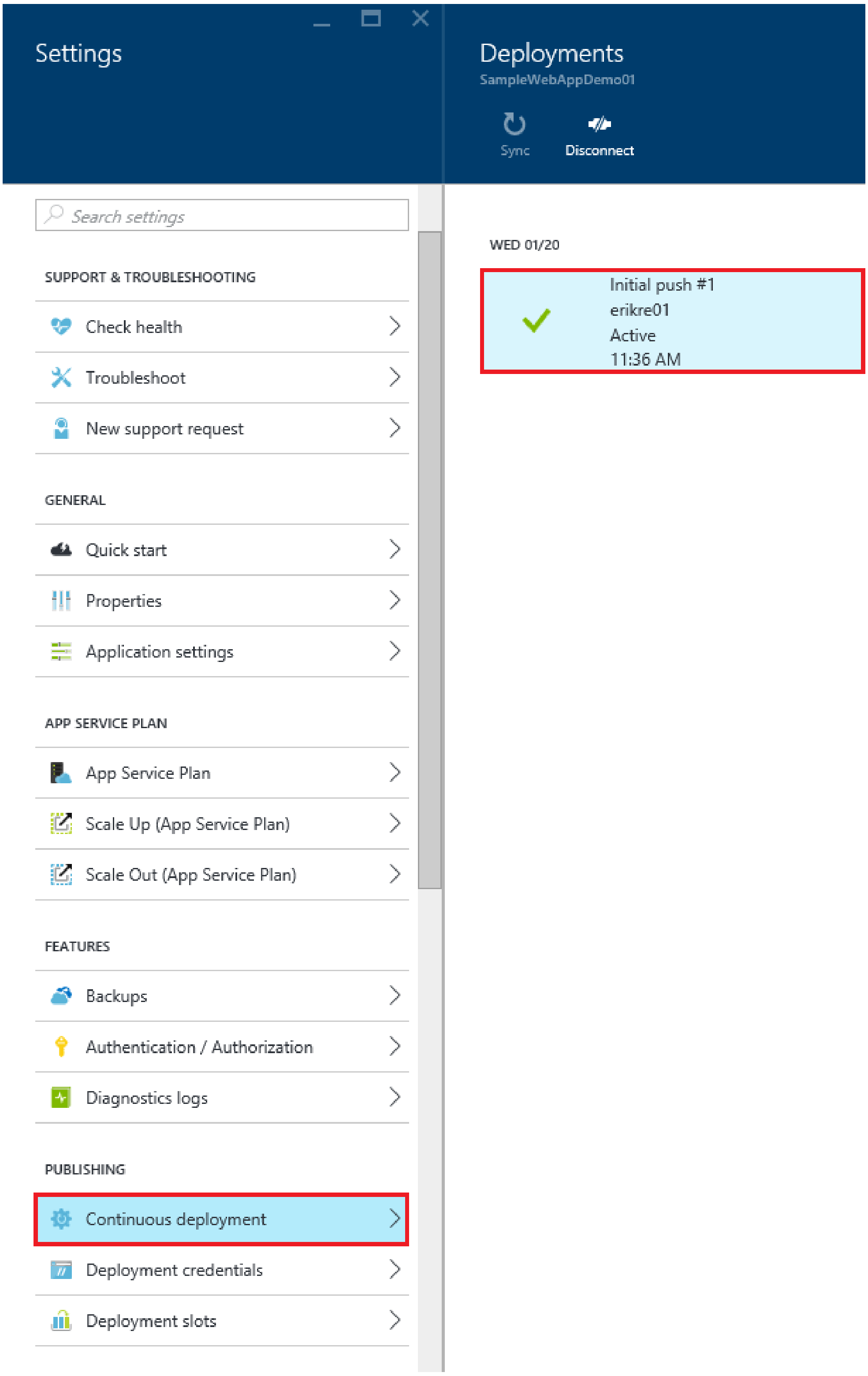
 Note

If you need to collaborate on a project, you should consider pushing to GitHub in between pushing to Azure.

Verify the Active Deployment

You can verify that you successfully transferred the web app from your local environment to Azure. You'll see the listed successful deployment.

In the Azure Portal, select your web app. Then, select **Settings** > **Continuous deployment**.



Run the app in Azure

Now that you have deployed your web app to Azure, you can run the app.

This can be done in two ways:

In the Azure Portal, locate the web app blade for your web app, and click **Browse** to view your app in your default browser.

Open a browser and enter the URL for your web app. For example:

```
http://SampleWebAppDemo.azurewebsites.net
```

Update your web app and republish

After you make changes to your local code, you can republish.

In **Solution Explorer** of Visual Studio, open the *Startup.cs* file.

In the `Configure` method, modify the `Response.WriteAsync` method so that it appears as follows:

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```

Save changes to *Startup.cs*.

In **Solution Explorer**, right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** will be displayed.

Enter a commit message, such as:

```
Update #2
```

Press the **Commit** button to commit the project changes.

Select **Home** > **Sync** > **Actions** > **Push**.

Note

As an alternative, you can push your changes from the **Command Window** by opening the **Command Window**, changing to your project directory, and entering a git command. For example:

`git push -u Azure-SampleApp master`

View the updated web app in Azure

View your updated web app by selecting **Browse** from the web app blade in the Azure Portal or by opening a browser and entering the URL for your web app. For example:

`http://SampleWebAppDemo.azurewebsites.net`

Additional Resources

Publishing and Deployment

Project Kudu

Continuous deployment to Azure for ASP.NET Core, with VSTS

By Damien Pontifex

This tutorial shows you how to create an ASP.NET Core web app using Visual Studio and deploy it to Azure App Service using continuous deployment with VSTS.

Note

To complete this tutorial, you need a Microsoft Azure account. If you don't have an account, you can activate your MSDN subscriber benefits or sign up for a free trial. You will also need a Visual Studio Team Services account. If you don't have an account, you can sign up for free.

Prerequisites

This tutorial assumes you already have the following:

ASP.NET Core (runtime and tooling). Hosted Build Pool servers in VSTS already have RC2 tooling installed.

Git

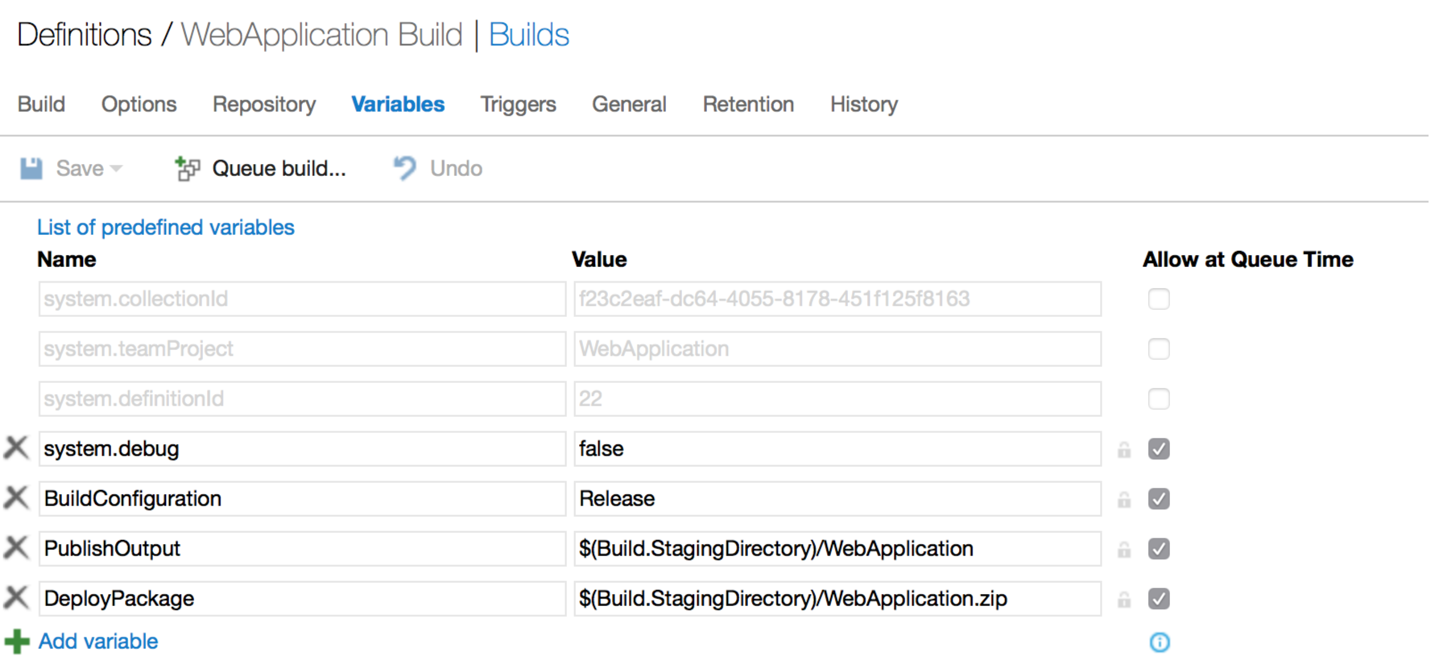
The Trackyon Advantage extension installed into your team services account. This adds an available zip task for later steps.

Setup VSTS Build

Setup some build variables to make later steps clearer and easier to retain consistent paths across build steps.

Create a variable for **PublishOutput** and set it to your desired path. We have used `$(Build.StagingDirectory)/WebApplication`

Create a variable for **DeployPackage** and set it to the path you would like the zipped web package to be at. We have used `$(Build.StagingDirectory)/WebApplication.zip` to have it alongside our published output.



Note

If you are using hosted build agents to build your ASP.NET Core application, the host will try to cache packages. As the hosted servers won't retain the cache, you can skip this step and reduce restore times by adding another variable here:

Name: `DOTNET_SKIP_FIRST_TIME_EXPERIENCE`
Value: `true`

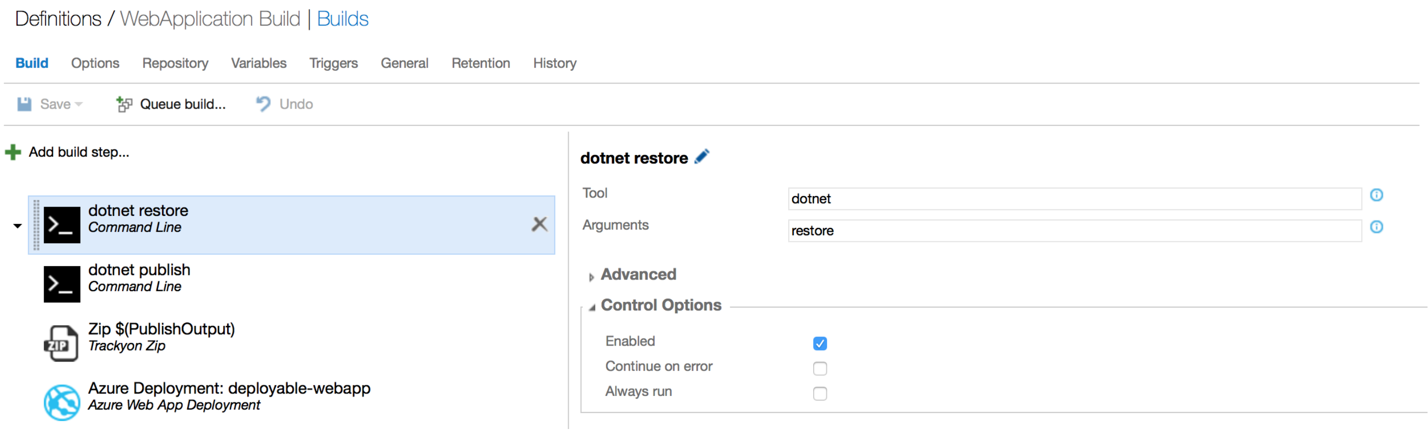
Use a Command Line build step to restore packages.

Click **Add build step...** and choose **Utility** > **Command Line** > **Add**

Set the arguments for the build step as:

Tool: `dotnet`

Arguments: `restore`



Use another Command Line build step to publish the project.

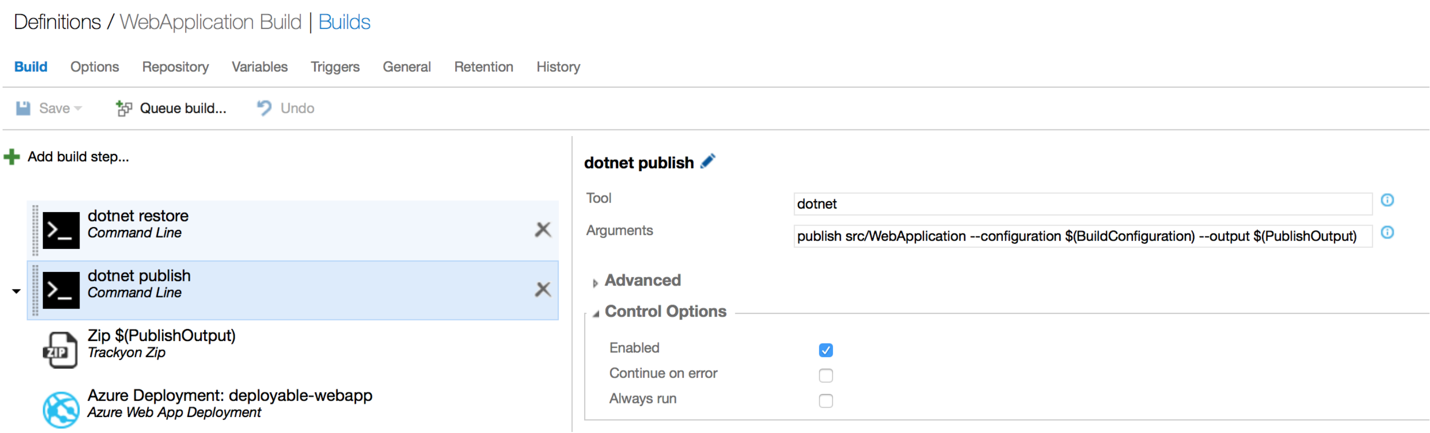
Click **Add build step...** and choose **Utility** > **Command Line** > **Add**

Set the arguments for the build step as:

Tool: `dotnet`

Arguments: `publish src/WebApplication --configuration $(BuildConfiguration) --output $(PublishOutput)`

Replace src/WebApplication to the path of your app to be deployed as appropriate



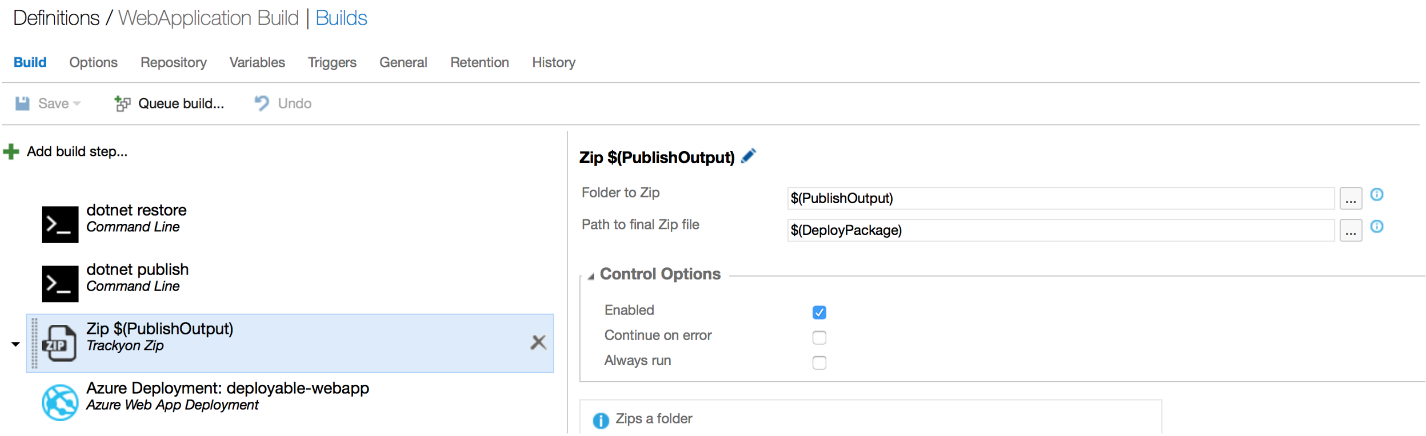
Compress the published output so it can be deployed to Azure App Service. We will use the Trackyon Advantage task we installed to zip the contents of our published output for deployment.

Click **Add build step...** and choose **Utility** > **Trackyon Zip** > **Add**

Set the arguments for the zip build step as:

Folder to Zip: `$(PublishOutput)`

Path to final Zip file: `$(DeployPackage)`



Use the Azure Web App Deployment build step to publish the compressed publish output to your Azure Web App. The Web Deploy Package will be the output of the contents compressed in step 4. In this case, we re-use the variable for it's path we setup earlier.

Click **Add build step...** and choose **Deploy** > **Azure Web App Deployment** > **Add**

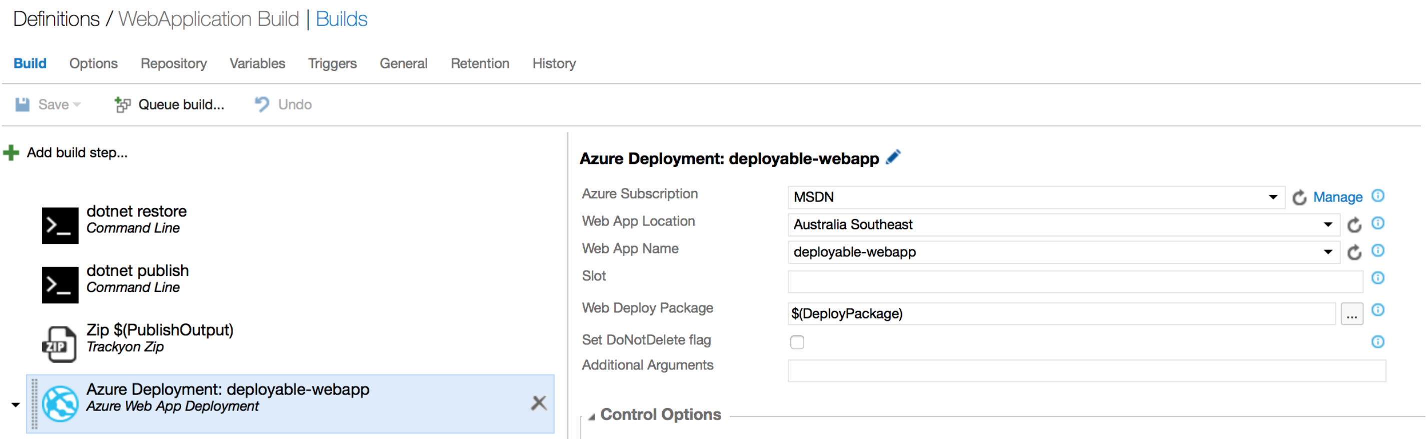
Set the arguments for the deployment step as:

Azure Subscription:

Web App Location:

Web App Name:

Web Deploy Package: `$(DeployPackage)`



Use VSTS Release

See Use VSTS to Build and Publish to an Azure Web App with Continuous Deployment, which shows how to configure a continuous delivery (CD) workflow for Azure App Service using Visual Studio Team Services. Azure Continuous Delivery in Team Services simplifies setting up a robust deployment pipeline to publish updates for your app to Azure App Service. The pipeline can be configured from the Azure portal to build, run tests, deploy to a staging slot, and then deploy to production.

VSTS Release management can alternatively be used to manage the release pipeline from the VSTS build. We require a small change to the build pipeline and setup of the release process.

If configured, remove the Azure Web App Deployment step from the VSTS build setup in the previous section.

Add a Copy and Publish Build Artifacts step to the build pipeline

Click **Add build step...** and choose **Utility** > **Copy and Publish Build Artifacts** > **Add**

Set the arguments for the copy and publish step as:

Contents: `$(DeployPackage)`

Artifact Name: `DeployPackage`

Artifact Type: `Server`

You will be able to create a release definition and link to the Build definition and utilise the artifacts copied from step 2 here for publishing.

Additional Resources

Publishing and Deployment

Team Services Build

Team Services Release

Create publish profiles for Visual Studio and MSBuild, to deploy ASP.NET Core apps

By Sayed Ibrahim Hashimi and Rick Anderson

This article focuses on using Visual Studio 2017 to create publish profiles. The publish profiles created with Visual Studio can be run from MSBuild and Visual Studio 2017.

The following *.csproj* file was created with the command `dotnet new mvc`:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
  </ItemGroup>

</Project>
```

The `Sdk` attribute in the `<Project>` element (in the first line) of the markup above does the following:

Imports the `props` file from *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.Props* at the beginning.
Imports the targets file from *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets* at the end.

The default location for `MSBuildSDKsPath` (with Visual Studio 2017 Enterprise) is the *%programfiles(x86)%\Microsoft Visual Studio\2017\Enterprise\MSBuild\Sdks* folder.

`"Microsoft.NET.Sdk.Web"` depends on:

Microsoft.NET.Sdk.Web.ProjectSystem
Microsoft.NET.Sdk.Publish

Which causes the following `props` and `targets` to be imported:

$(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

Publish targets will again import the right set of targets based on the publish method used.

When MSBuild or Visual Studio loads a project, the following high level actions are performed:

Build project
Compute files to publish
Publish files to destination

Compute project items

When the project is loaded, the project items (files) are computed. The `item type` attribute determines how the file is processed. By default, *.cs* files are included in the `Compile` item list. Files in the `Compile` item list are compiled.

The `Content` item list contains files that will be published in addition to the build outputs. By default, files matching the pattern wwwroot/** will be included in the `Content` item. wwwroot/** is a globbing pattern that specifies all files in the *wwwroot* folder

and subfolders. If you need to explicitly add a file to the publish list you can add the file directly in the *.csproj* file as shown in Including Files.

When you select the **Publish** button in Visual Studio or when you publish from command line:

The properties/items are computed (the files that are needed to build).
Visual Studio only: NuGet packages are restored. (Restore needs to be explicit by the user on the CLI.)
The project builds.
The publish items are computed (the files that are needed to publish).
The project is published. (The computed files are copied to the publish destination.)

Simple command line publishing

This section works on all .NET Core supported platforms and doesn't require Visual Studio. In the samples below, the `dotnet publish` command is run from the project directory (which contains the *.csproj* file). If you're not in the project folder, you can explicitly pass in the project file path. For example:

```
dotnet publish  c:/webs/web1
```

Run the following commands to create and publish a web app:

```
dotnet new mvc
dotnet restore
dotnet publish
```

The `dotnet publish` produces output similar to the following:

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version 15.1.548.43366
Copyright (C) Microsoft Corporation. All rights reserved.

  Web1 -> C:\Webs\Web1\bin\Debug\netcoreapp1.1\Web1.dll
```

The default publish folder is `bin\$(Configuration)\netcoreapp<version>\publish`. The default for `$(Configuration)` is Debug. In the sample above, the `<TargetFramework>` is `netcoreapp1.1`. The actual path in the sample above is *bin\Debug\netcoreapp1.1\publish*.

`dotnet publish -h` displays help information for publish.

The following command specifies a `Release` build and the publishing directory:

```
dotnet publish -c Release -o C:/MyWebs/test
```

The `dotnet publish` command calls `MSBuild` which invokes the `Publish` target. Any parameters passed to `dotnet publish` are passed to `MSBuild`. The `-c` parameter maps to the `Configuration` MSBuild property. The `-o` parameter maps to `OutputPath`.

You can pass MSBuild properties using either of the following formats:

`p:<NAME>=<VALUE>`
`/p:<NAME>=<VALUE>`

The following command publishes a `Release` build to a network share:

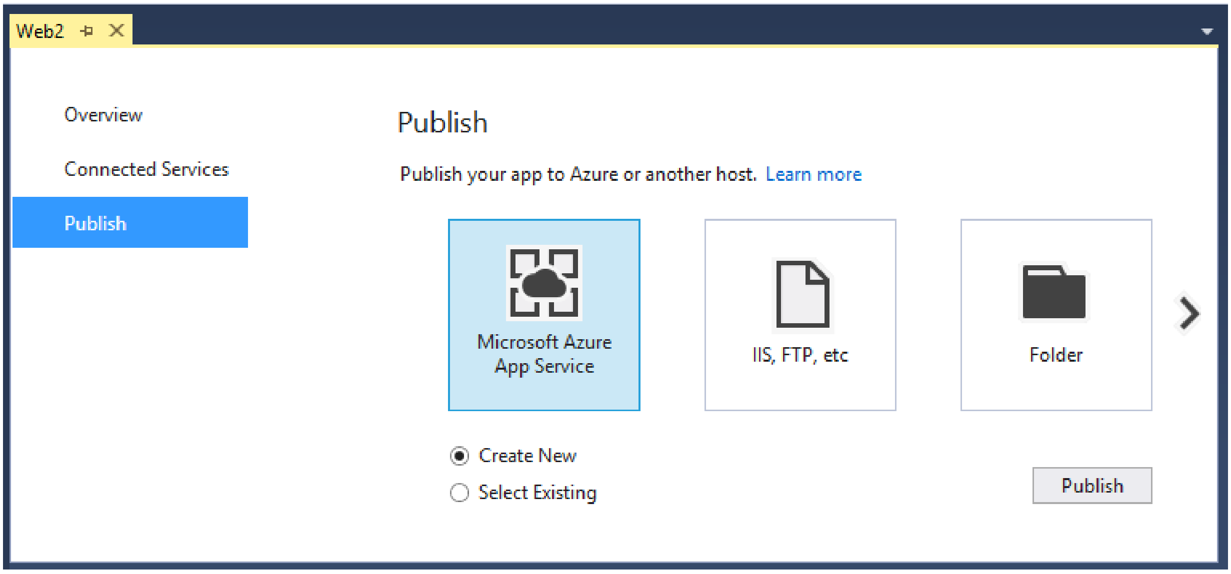
`dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb`

The network share is specified with forward slashes (*//r8/*) and works on all .NET Core supported platforms.

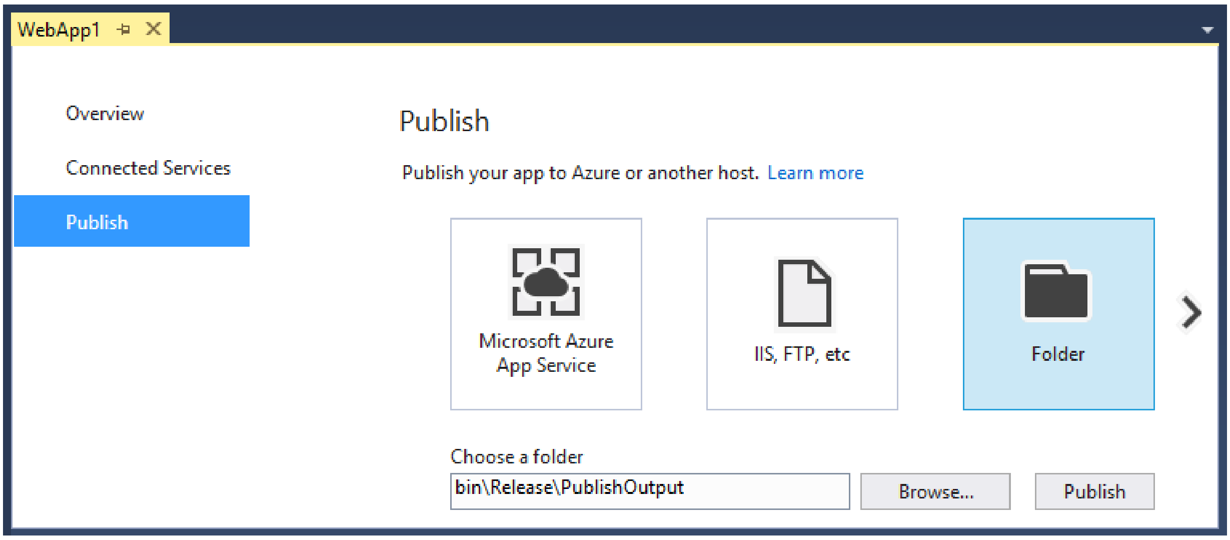
Publish profiles

This section uses Visual Studio 2017 and higher to create publishing profiles. Once created, you can publish from Visual Studio or the command line.

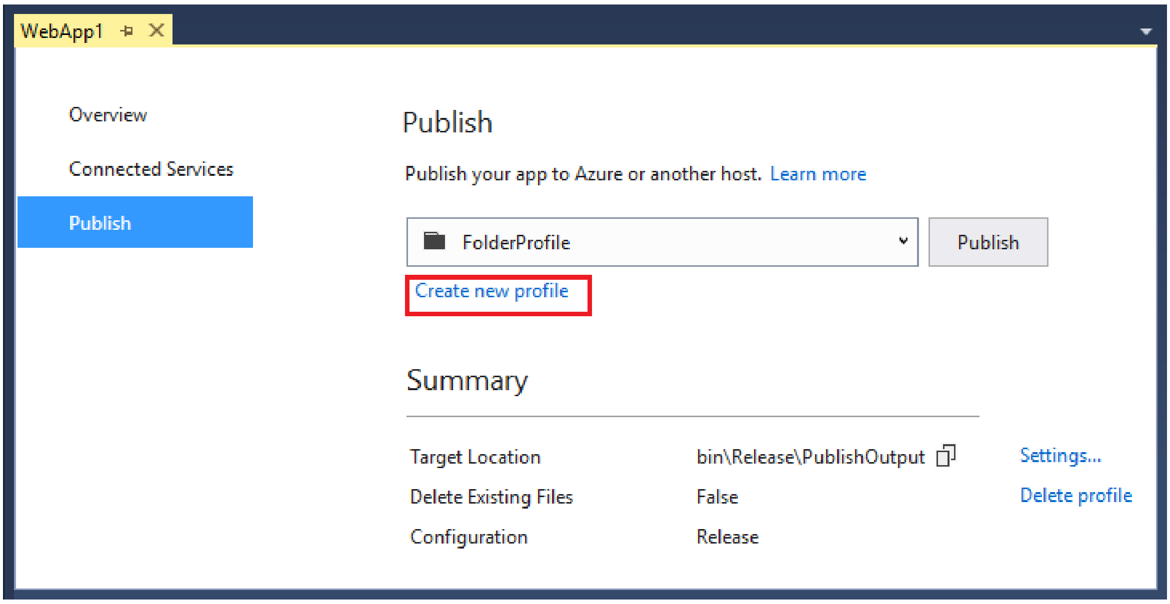
Publish profiles can simplify the publishing process. You can have multiple publish profiles. To create a publish profile in Visual Studio, right click on the project in Solution Explore and select **Publish**. Alternatively, you can select **Publish <project name>** from the build menu. The **Publish** tab of the application capacities page is displayed. If the project doesn't contain a publish profile, the following page is displayed:



When **Folder** is selected, the **Publish** button creates a folder publish profile and publishes.



Once a publish profile is created, the **Publish** tab changes, and you select **Create new profile** to create a new profile.



The Publish wizard supports the following publish targets:

Microsoft Azure App Service
IIS, FTP, etc (for any web server)
Folder
Import profile (allows you to import a profile).
Microsoft Azure Virtual Machines

See What publishing options are right for me? for more information.

When you create a publish profile with Visual Studio, a *Properties/PublishProfiles/<publish name>.pubxml* MSBuild file is created. This *.pubxml* file is a MSBuild file and contains publish configuration settings. You can change this file to customize the build and publish process. This file is read by the publishing process. `<LastUsedBuildConfiguration>` is special because it's a global property and shouldn't be in any file that's imported in the build. See MSBuild: how to set the configuration property for more info. The *.pubxml* file should not be checked into source control because it depends on the *.user* file. The *.user* file should never be checked into source control because it can contain sensitive information and it's only valid for one user and machine.

Sensitive information (like the publish password) is encrypted on a per user/machine level and stored in the *Properties/PublishProfiles/<publish name>.pubxml.user* file. Because this file can contain sensitive information, it should **not** be checked into source control.

For an overview of how to publish a web app on ASP.NET Core see Publishing and Deployment. Publishing and Deployment is an open source project at https://github.com/aspnet/websdk.

Currently `dotnet publish` doesn't have the ability to use publish profiles. To use publish profiles, use `dotnet build`. `dotnet build` invokes MSBuild on the project. Alternatively, call `msbuild` directly.

Set the following MSBuild properties when using a publish profile:

`DeployOnBuild=true`
`PublishProfile=<Publish profile name>`

For example, when publishing with a profile named *FolderProfile* you can execute either of the commands below.

`dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`
`msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile`

When you invoke `dotnet build` it will call `msbuild` to run the build and publish process. Calling `dotnet build` or `msbuild` is essentially equivalent when you pass in a folder profile. When calling MSBuild directly on Windows you get the full .NET Framework version of MSBuild. MSDeploy is currently limited to Windows machines for publishing. Calling `dotnet build` on a non-folder profile invokes MSBuild, and MSBuild uses MSDeploy on non-folder profiles. Calling `dotnet build` on a non-folder profile invokes MSBuild (using MSDeploy) and results in a failure (even when running on a Windows platform). To publish with a non-folder profile, call MSBuild directly.

The following folder publish profile was created with Visual Studio and publishes to a network share:

```
<?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>
```

Note `<LastUsedBuildConfiguration>` is set to `Release`. When publishing from Visual Studio, the `<LastUsedBuildConfiguration>` configuration property value is set using the value when the publish process is started. The `<LastUsedBuildConfiguration>` configuration property is special and shouldn't be overridden in an imported MSBuild file. You can override this property from the command line. For example:

```
dotnet build -c Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Using MSBuild:

```
msbuild /p:Configuration=Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Publish to an MSDeploy endpoint from the command line

As previously mentioned, you can publish using `dotnet publish` or the `msbuild` command. `dotnet publish` runs in the context for of .NET Core. `msbuild` requires the full .NET framework, and is therefore limited to Windows environments.

The easiest way to publish with MSDeploy is to first create a publish profile in Visual Studio 2017 and use the profile from the command line.

In the following sample, I created an ASP.NET Core web app (using `dotnet new mvc`) and added an Azure publish profile with Visual Studio.

You run `msbuild` from a **Developer Command Prompt for VS 2017**. The Developer Command Prompt will have the correct *msbuild.exe* in its path and set some MSBuild variables.

MSBuild uses the following syntax:

```
msbuild <path-to-project-file> /p:DeployOnBuild=true /p:PublishProfile=<Publish Profile> /p:Username=<USERNAME>
/p:Password=<PASSWORD>
```

You can get the `Password` from the *<Publish name>.PublishSettings* file. You can download the *.PublishSettings* file from:

Solution Explorer: Right click on the Web App and select **Download Publish Profile**.
The Azure Management Portal: Select **Get publish profile** from the Web App blade.

`Username` can be found in the publish profile.

The following sample uses the "Web11112 - Web Deploy" publish profile:

```
msbuild "C:\Webs\Web1\Web1.csproj" /p:DeployOnBuild=true
 /p:PublishProfile="Web11112 - Web Deploy"  /p:Username="$Web11112"
 /p:Password="<password removed>"
```

Excluding files

When publishing ASP.NET Core web apps, the build artifacts and contents of the *wwwroot* folder are included. `msbuild` supports globbing patterns. For example, the following `<Content>` element markup will exclude all text (*.txt*) files from the *wwwroot/content* folder and all its subfolders.

```
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never" />
</ItemGroup>
```

The markup above can be added to a publish profile or the *.csproj* file. When added to the *.csproj* file, the rule is added to all publish profiles in the project.

The following `<MsDeploySkipRules>` element markup exludes all files from the *wwwroot/content* folder:

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` will not delete the *skip* targets from the deployment site. `<Content>` targeted files and folders will be deleted from the deployment site. For example, suppose you had deployed a web app with the following files:

Views/Home/About1.cshtml
Views/Home/About2.cshtml
Views/Home/About3.cshtml

If you added the following `<MsDeploySkipRules>` markup, those files would not be deleted on the deployment site.

```
<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>
```

The `<MsDeploySkipRules>` markup shown above prevents the *skipped* files from being depoyed, but will not delete those files once they are deployed.

The following `<Content>` markup would delete the targeted files at the deployment site:

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never" />
</ItemGroup>
```

Using command line deployment with the `<Content>` markup above would result in output similar to the following:

```
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).
```

Including files

The following markup can be used to include an *images* folder outside the project directory to the *wwwroot/images* folder of the publish site.

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotnetPublishFiles Include="@(_CustomFiles)">    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%
(Filename)%(Extension)</DestinationRelativePath>
  </DotnetPublishFiles>
</ItemGroup>
```

The markup can be added to the *.csproj* file or the publish profile. If it's added to the *.csproj* file, it will be included in each publish profile in the project.

The following highlighted markup shows how to:

Copy a file from outside the project into the *wwwroot* folder.
Exclude the *wwwroot\Content* folder.
Exclude *Views\Home\About2.cshtml*.

```xml
<?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>
```

See the WebSDK Readme for more deployment samples.

Run a target before or after publishing

The builtin `BeforePublish` and `AfterPublish` targets can be used to execute a target before or after the publish target. The following markup can be added to the publish profile to log messages to the console output before and after publishing:

```xml
<Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
    <Message Text="Inside BeforePublish" Importance="high" />
  </Target>
  <Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
    <Message Text="Inside AfterPublish" Importance="high" />
</Target>
```

The Kudu service

To view the files on your Azure Web App, use the kudu service. Append the `scm` token to the name or your Web App. For example:

URL	RESULT
`http://mysite.azurewebsites.net/`	Web App
`http://mysite.scm.azurewebsites.net/`	Kudu sevice

Select the Debug Console menu item to view/edit/delete/add files.

Additional resources

Web Deploy (msdeploy) simplifies deployment of Web applications and Web sites to IIS servers.

https://github.com/aspnet/websdk: File issues and request features for deployment.

Directory structure of published ASP.NET Core apps

By Luke Latham

In ASP.NET Core, the application directory, *publish*, is comprised of application files, config files, static assets, packages, and the runtime (for self-contained apps). This is the same directory structure as previous versions of ASP.NET, where the entire application lives inside the web root directory.

APP TYPE	DIRECTORY STRUCTURE
Framework-dependent Deployment	• publish* ◦ logs* (if included in publishOptions) ◦ refs* ◦ runtimes* ◦ Views* (if included in publishOptions) ◦ wwwroot* (if included in publishOptions) ◦ .dll files ◦ myapp.deps.json ◦ myapp.dll ◦ myapp.pdb ◦ myapp.PrecompiledViews.dll (if precompiling Razor Views) ◦ myapp.PrecompiledViews.pdb (if precompiling Razor Views) ◦ myapp.runtimeconfig.json ◦ web.config (if included in publishOptions)
Self-contained Deployment	• publish* ◦ logs* (if included in publishOptions) ◦ refs* ◦ Views* (if included in publishOptions) ◦ wwwroot* (if included in publishOptions) ◦ .dll files ◦ myapp.deps.json ◦ myapp.exe ◦ myapp.pdb ◦ myapp.PrecompiledViews.dll (if precompiling Razor Views) ◦ myapp.PrecompiledViews.pdb (if precompiling Razor Views) ◦ myapp.runtimeconfig.json ◦ web.config (if included in publishOptions)

* Indicates a directory

The contents of the *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory in the deployment, its location serves as the server's physical path to the hosted application. The *wwwroot* directory, if present, only contains static assets. The *logs* directory may be included in the deployment by creating it in the project and adding the `<Target>` element shown below to your *.csproj* file or by physically creating the directory on the server.

```
<Target Name="CreateLogsFolder" AfterTargets="AfterPublish">
  <MakeDir Directories="$(PublishDir)logs" Condition="!Exists('$(PublishDir)logs')" />
  <MakeDir Directories="$(PublishUrl)Logs" Condition="!Exists('$(PublishUrl)Logs')" />
</Target>
```

The first `<MakeDir>` element, which uses the `PublishDir` property, is used by the .NET Core CLI to determine the target location for the publish operation. The second `<MakeDir>` element, which uses the `PublishUrl` property, is used by Visual Studio to determine the target location. Visual Studio uses the `PublishUrl` property for compatibility with non-.NET Core projects.

The deployment directory requires Read/Execute permissions, while the *logs* directory requires Read/Write permissions.

Additional directories where assets will be written require Read/Write permissions.

Community OSS authentication options

This page contains community provided open source authentication options for ASP.NET Core. This page will be periodically updated as new providers become available.

OSS Authentication Providers

The list below is sorted alphabetically.

NAME	DESCRIPTION
AspNet.Security.OpenIdConnect.Server (ASOS)	Low-level/protocol-first OpenID Connect server framework for ASP.NET Core and OWIN/Katana
IdentityServer4	OpenID Connect and OAuth 2.0 framework for ASP.NET Core - officially certified by the OpenID Foundation and under governance of the .NET Foundation
OpenIddict	Easy-to-use OpenID Connect server for ASP.NET Core

To get your provider added here edit this page.

Introduction to Identity

By Pranav Rastogi, Rick Anderson, Tom Dykstra, Jon Galloway, and Erik Reitan

ASP.NET Core Identity is a membership system which allows you to add login functionality to your application. Users can create an account and login with a user name and password or they can use an external login providers such as Facebook, Google, Microsoft Account, Twitter and more.

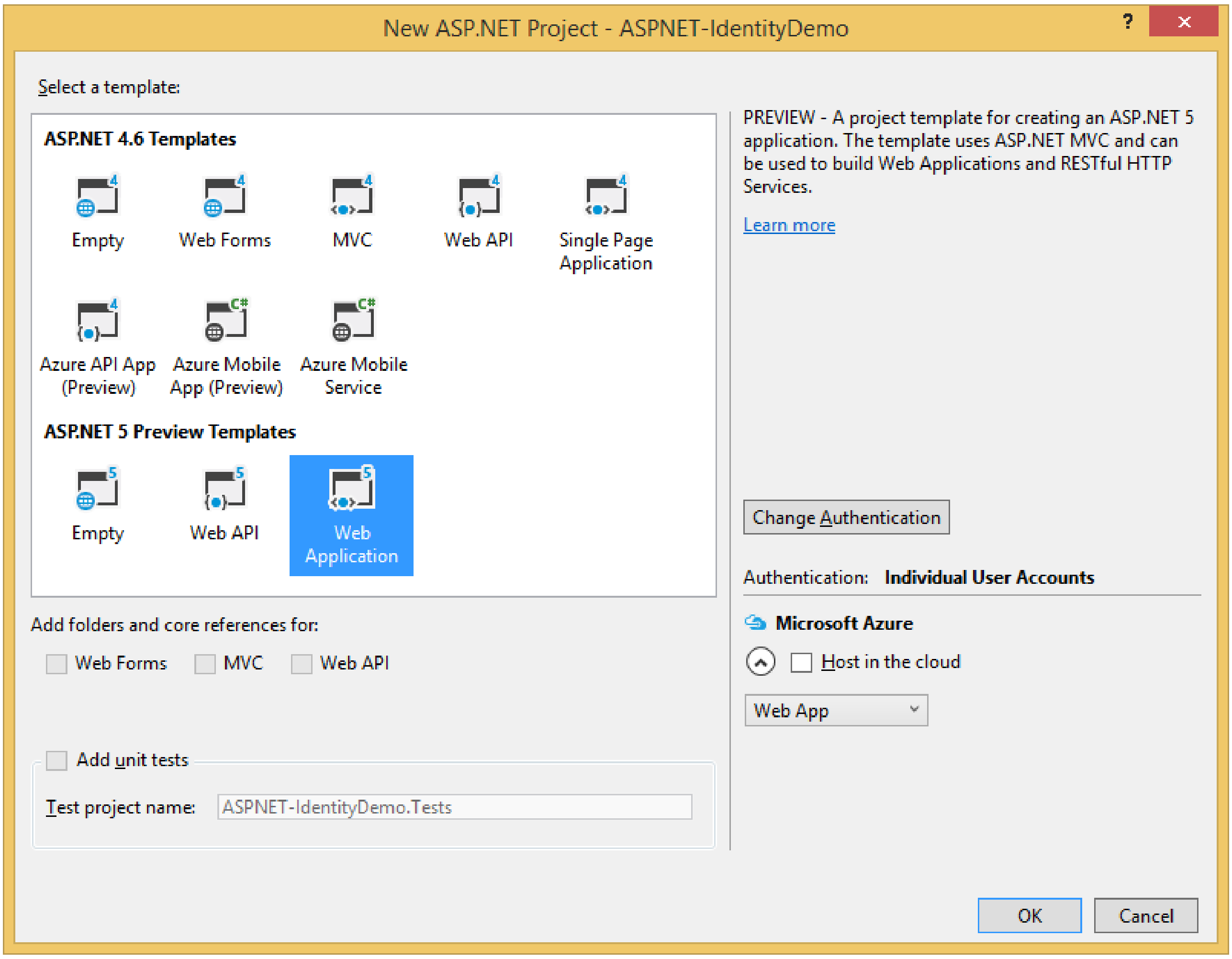
You can configure ASP.NET Core Identity to use a SQL Server database to store user names, passwords, and profile data. Alternatively, you can use your own persistent store to store data in another persistent storage, such as Azure Table Storage.

Overview of Identity

In this topic, you'll learn how to use ASP.NET Core Identity to add functionality to register, log in, and log out a user. You can follow along step by step or just read the details. For more detailed instructions about creating apps using ASP.NET Core Identity, see the Next Steps section at the end of this article.

Create an ASP.NET Core Web Application project in Visual Studio with Individual User Accounts.

In Visual Studio, select **File** -> **New** -> **Project**. Then, select the **ASP.NET Web Application** from the **New Project** dialog box. Continue by selecting an ASP.NET Core **Web Application** with **Individual User Accounts** as the authentication method.



The created project contains the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` package, which will persist the identity data and schema to SQL Server using Entity Framework Core.

Note

In Visual Studio, you can view NuGet packages details by selecting **Tools** -> **NuGet Package Manager** -> **Manage NuGet Packages for Solution**.

The identity services are added to the application in the `ConfigureServices` method in the `Startup` class:

```
// 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.AddEntityFramework()
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
```

These services are then made available to the application through dependency injection.

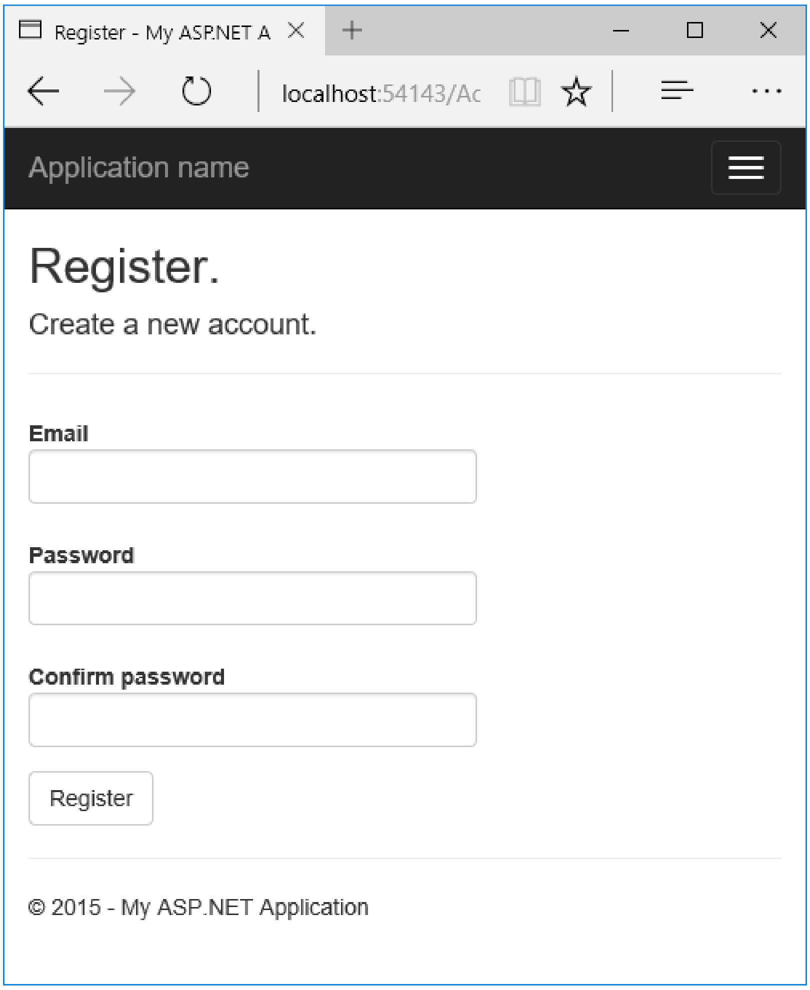
Identity is enabled for the application by calling `UseIdentity` in the `Configure` method of the `Startup` class. This adds cookie-based authentication to the request pipeline.

```
    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;

        // Lockout settings
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
        options.Lockout.MaxFailedAccessAttempts = 10;

        // Cookie settings
        options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
        options.Cookies.ApplicationCookie.LoginPath = "/Account/LogIn";
        options.Cookies.ApplicationCookie.LogoutPath = "/Account/LogOut";

        // User settings
        options.User.RequireUniqueEmail = true;
    });
}

// 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();

    if (env.IsDevelopment())
    {
        app.UseBrowserLink();
```

For more information about the application start up process, see Application Startup.

Creating a user.

Launch the application from Visual Studio (**Debug** -> **Start Debugging**) and then click on the **Register** link in the browser to create a user. The following image shows the Register page which collects the user name and password.



When the user clicks the **Register** link, the `UserManager` and `SignInManager` services are injected into the Controller:

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> _userManager;
    private readonly SignInManager<ApplicationUser> _signInManager;
    private readonly IEmailSender _emailSender;
    private readonly ISmsSender _smsSender;
    private static bool _databaseChecked;
    private readonly ILogger _logger;

    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>();
    }

    //
    // GET: /Account/Login
```

Then, the **Register** action creates the user by calling `CreateAsync` function of the `UserManager` object, as shown below:

```
[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);
}
```

Log in.

If the user was successfully created, the user is logged in by the `SignInAsync` method, also contained in the `Register` action. By signing in, the `SignInAsync` method stores a cookie with the user's claims.

```
[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);
}
```

The above `SignInAsync` method calls the below `SignInAsync` task, which is contained in the `SignInManager` class.

If needed, you can access the user's identity details inside a controller action. For instance, by setting a breakpoint inside the `HomeController.Index` action method, you can view the `User.claims` details. By having the user signed-in, you can make authorization decisions. For more information, see Authorization.

As a registered user, you can log in to the web app by clicking the **Log in** link. When a registered user logs in, the `Login` action of the `AccountController` is called. Then, the **Login** action signs in the user using the `PasswordSignInAsync` method contained in the `Login` action.

```
[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);
}
```

Log off.

Clicking the **Log off** link calls the `LogOut` action in the account controller.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> LogOut()
{
    await _signInManager.SignOutAsync();
    _logger.LogInformation(4, "User logged out.");
    return RedirectToAction(nameof(HomeController.Index), "Home");
}
```

The code above shows the `SignInManager.SignOutAsync` method. The `SignOutAsync` method clears the users claims stored in a cookie.

Configuration.

Identity has some default behaviors that you can override in your application's startup class.

```
// Configure Identity
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;

    // Lockout settings
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
    options.Lockout.MaxFailedAccessAttempts = 10;

    // Cookie settings
    options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
    options.Cookies.ApplicationCookie.LoginPath = "/Account/LogIn";
    options.Cookies.ApplicationCookie.LogoutPath = "/Account/LogOut";

    // User settings
    options.User.RequireUniqueEmail = true;
});
```

For more information about how to configure Identity, see Configure Identity.

You also can configure the data type of the primary key, see Configure Identity primary keys data type.

View the database.

After stopping the application, view the user database from Visual Studio by selecting **View** -> **SQL Server Object Explorer**. Then, expand the following within the **SQL Server Object Explorer**:

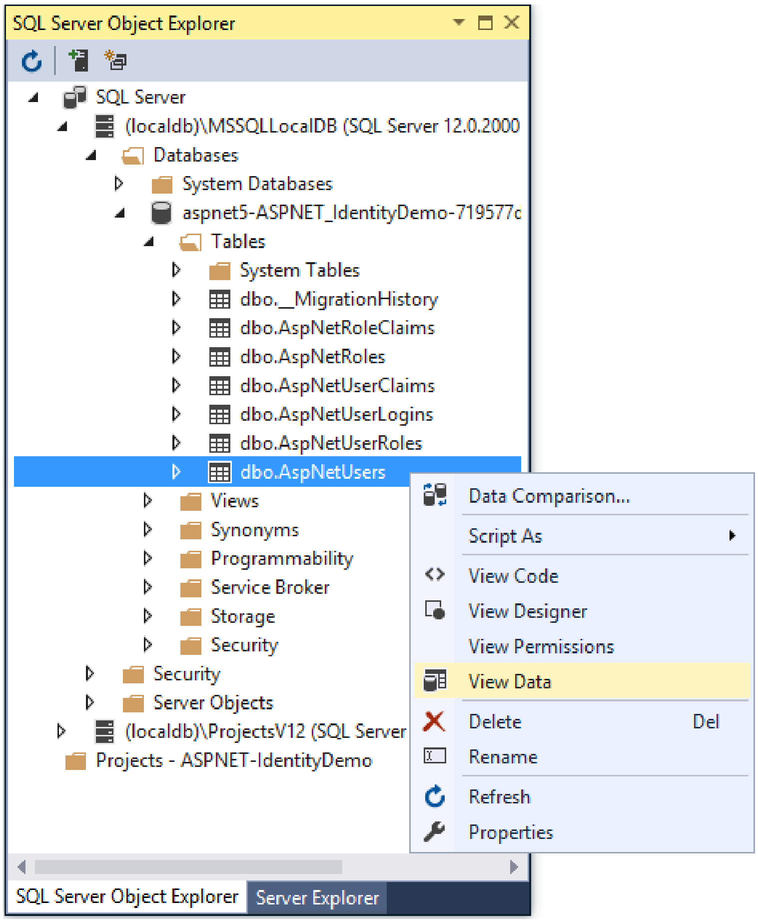
(localdb)\MSSQLLocalDB

Databases

aspnet5-<*the name of your application*>

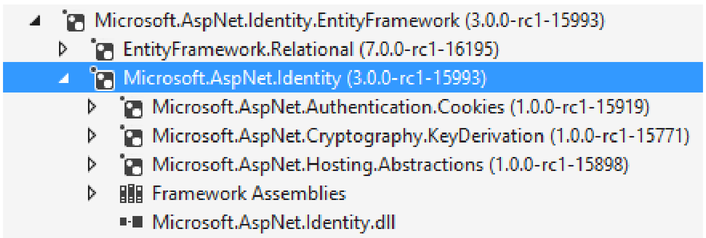
Tables

Next, right-click the **dbo.AspNetUsers** table and select **View Data** to see the properties of the user you created.



Identity Components

The primary reference assembly for the identity system is `Microsoft.AspNetCore.Identity`. This package contains the core set of interfaces for ASP.NET Core Identity.



These dependencies are needed to use the identity system in ASP.NET Core applications:

`EntityFramework.SqlServer` - Entity Framework is Microsoft's recommended data access technology for relational databases.

`Microsoft.AspNetCore.Authentication.Cookies` - Middleware that enables an application to use cookie based authentication, similar to ASP.NET's Forms Authentication.

`Microsoft.AspNetCore.Cryptography.KeyDerivation` - Utilities for key derivation.

`Microsoft.AspNetCore.Hosting.Abstractions` - Hosting abstractions.

Migrating to ASP.NET Core Identity

For additional information and guidance on migrating your existing identity store see Migrating Authentication and Identity

Next Steps

Migrating Authentication and Identity

Account Confirmation and Password Recovery

Two-factor authentication with SMS

Enabling authentication using Facebook, Google and other external providers

Configure Identity

ASP.NET Core Identity has some default behaviors that you can override easily in your application's startup class.

Password's policy

By default, Identity requires very secure passwords who have to contains uppercase character, lowercase character, digits and some others restrictions that you sometimes need to simplify. It's very simple to do that, all the configuration is accessible in the startup class of your application.

```
// Password settings
options.Password.RequireDigit = true;
options.Password.RequiredLength = 8;
options.Password.RequireNonAlphanumeric = false;
options.Password.RequireUppercase = true;
options.Password.RequireLowercase = false;
```

Application's cookie settings

With the same philosophy of the password's policy, all the settings about the application's cookie can be changed in the startup class.

```
            // Cookie settings
options.Cookies.ApplicationCookie.CookieName = "YouAppCookieName";
            options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
            options.Cookies.ApplicationCookie.LoginPath = "/Account/LogIn";
            options.Cookies.ApplicationCookie.LogoutPath = "/Account/LogOff";
options.Cookies.ApplicationCookie.AccessDeniedPath = "/Account/AccessDenied";
            options.Cookies.ApplicationCookie.AutomaticAuthenticate = true;
            options.Cookies.ApplicationCookie.AuthenticationScheme =
Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationDefaults.AuthenticationScheme;
            options.Cookies.ApplicationCookie.ReturnUrlParameter =
Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationDefaults.ReturnUrlParameter;
```

User's lockout

```
// Lockout settings
options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
options.Lockout.MaxFailedAccessAttempts = 10;
options.Lockout.AllowedForNewUsers = true;
```

Configure Identity primary keys data type

ASP.NET Core Identity allows you to easily configure the data type you want for the primary keys. By default, Identity uses string data type but you can very quickly override this behavior.

How to

The first step is to implement the Identity's model, and override the behavior to 's one.

```
namespace webapptemplate.Models
{
    // Add profile data for application users by adding properties to the ApplicationUser class
    public class ApplicationUser : IdentityUser<Guid>
    {
    }
}
```

```
namespace webapptemplate.Models
{
    public class ApplicationRole : IdentityRole<Guid>
    {
    }
}
```

Implement the database context of Identity with your models and the data type you want for primary keys

```
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);
        }
    }
}
```

Use your models and the data type you want for primary keys when you declare the identity service in your application's startup class

```
    public void ConfigureServices(IServiceCollection services)
    {
        // Add framework services.
        services.AddEntityFramework()
            .AddSqlServer()
            .AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));

        services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
            .AddDefaultTokenProviders();

        services.AddMvc();

        // Add application services.
        services.AddTransient<IEmailSender, AuthMessageSender>();
        services.AddTransient<ISmsSender, AuthMessageSender>();

        // Configure Identity
        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;

            // Lockout settings
            options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(30);
            options.Lockout.MaxFailedAccessAttempts = 10;
            options.Lockout.AllowedForNewUsers = true;

            // Cookie settings
options.Cookies.ApplicationCookie.CookieName = "YouAppCookieName";
            options.Cookies.ApplicationCookie.ExpireTimeSpan = TimeSpan.FromDays(150);
            options.Cookies.ApplicationCookie.LoginPath = "/Account/LogIn";
            options.Cookies.ApplicationCookie.LogoutPath = "/Account/LogOff";
options.Cookies.ApplicationCookie.AccessDeniedPath = "/Account/AccessDenied";
            options.Cookies.ApplicationCookie.AutomaticAuthenticate = true;
            options.Cookies.ApplicationCookie.AuthenticationScheme =
Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationDefaults.AuthenticationScheme;
```

Enabling authentication using Facebook, Google and other external providers

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

This tutorial demonstrates how to build an ASP.NET Core app that enables users to log in using OAuth 2.0/1.0a with credentials from external authentication providers.

Facebook, Twitter, Google, and Microsoft providers are covered in the following sections. See AspNet.Security.OAuth.Providers and AspNet.Security.OpenId.Providers for additional providers.

Enabling users to sign in with their existing credentials is convenient for the users and shifts many of the complexities of managing the sign-in process onto a third party. For examples of how social logins can drive traffic and customer conversions, see case studies by Facebook and Twitter.

Note

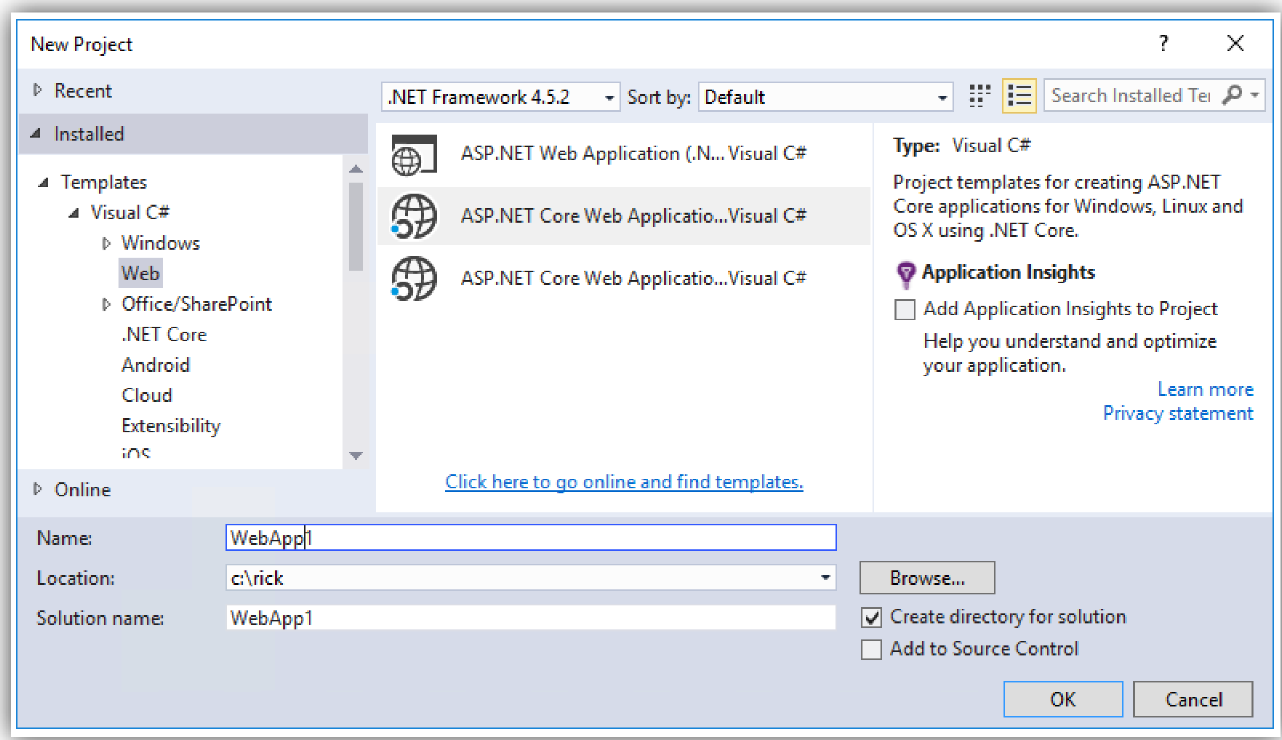
Packages presented here abstract a great deal of complexity of the OAuth authentication flow, but understanding the details may become necessary when troubleshooting. Many resources are available; for example, see Introduction to OAuth 2 or Understanding OAuth 2. Some issues can be resolved by looking at the ASP.NET Core source code for the provider packages.

Create a New ASP.NET Core Project

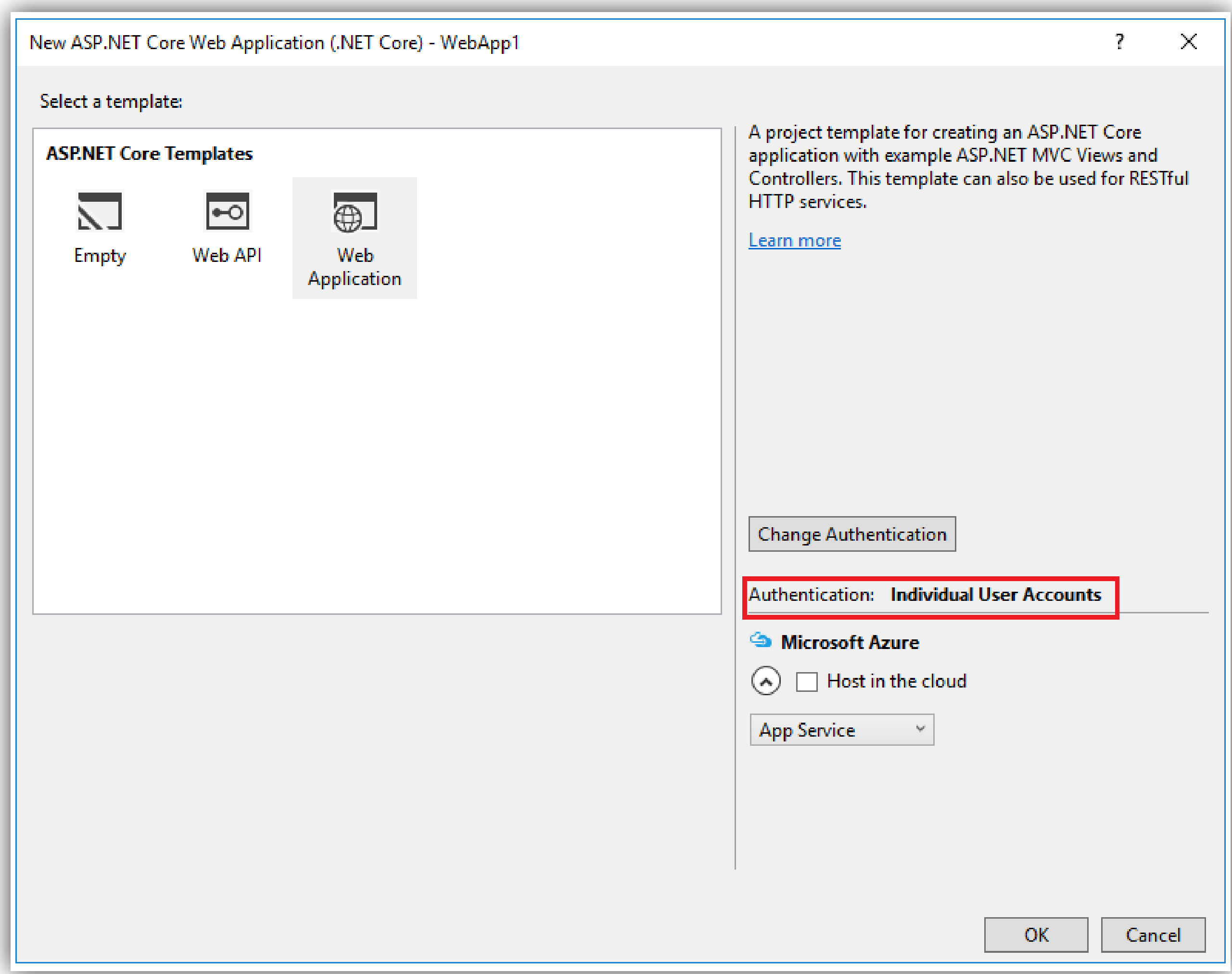
Note

This tutorial requires the latest update (**Update 3**) for Visual Studio 2015 and ASP.NET Core.

In Visual Studio, create a new project (from the Start Page, or via **File > New > Project**):



Tap **Web Application** and verify **Authentication** is set to **Individual User Accounts**:



Enable SSL

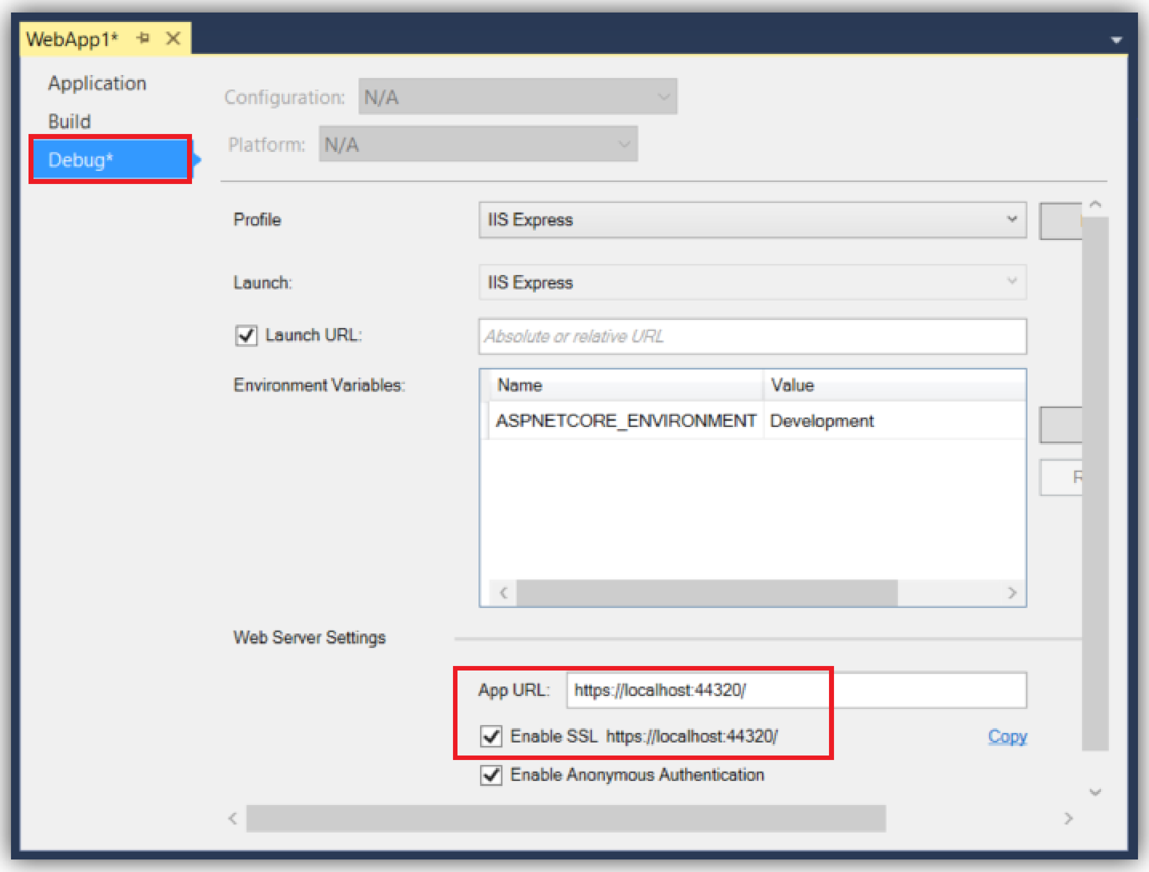
Some external authentication providers reject requests coming from origins that don't use the **https** protocol. This reflects the trend of major providers like Google moving their public API services to https and discontinuing the use of unencrypted endpoints. We encourage you to follow this trend and enable SSL for your entire site. For Visual Studio 2017 RC1, see instructions below.

In **Solution Explorer**, right-click the project and select **Properties**.

On the left pane, tap **Debug**.

Check **Enable SSL**.

Copy the SSL URL and paste it into the **App URL**:



Visual Studio 2017 RC1:

Edit the *properties/launchsettings.json* file. Add an sslPort between 44300 and 44399:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:62183/",
      "sslPort": 0
    }
  },
```

Modify the `services.AddMvc();` code in `Startup.cs` under `ConfigureServices` to reject all requests that are not coming over *https*:

```
services.AddMvc(options =>
{
    options.SslPort = 44321;
    options.Filters.Add(new RequireHttpsAttribute ());
});
```

Test the app to ensure that static files are still being served and publicly exposed routes are accessible.
There shouldn't be any warnings logged to the browser console in Developer Tools.
Attempting to navigate to the previous URL that used the *http* protocol should now result in **connection rejected** errors from the browser or a blank page.

Use SecretManager to store tokens assigned by login providers

The template used to create the sample project in this tutorial has code in `Startup.cs` which reads the configuration values from a secret store:

```
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);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=532709
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

As a best practice, it is not recommended to store the secrets in a configuration file in the application since they can be checked into source control which may be publicly accessible.

The **SecretManager** tool stores sensitive application settings in the user profile folder on the local machine. These settings are then seamlessly merged with settings from all other sources during application startup.

▯ Note

Most login providers assign **Application Id** and **Application Secret** during the registration process. These values are effectively the *user name* and *password* your application uses to access their API, and constitute the "secrets" linked to your application configuration with the help of **Secret Manager** instead of storing them in configuration files directly.

Install the Secret Manager tool so that you can use it to store tokens assigned by each login provider below.

Setup login providers required by your application

Use the following pages to configure your application to use the respective providers:

Facebook instructions
Twitter instructions
Google instructions
Microsoft instructions
Other provider instructions

▯ Note

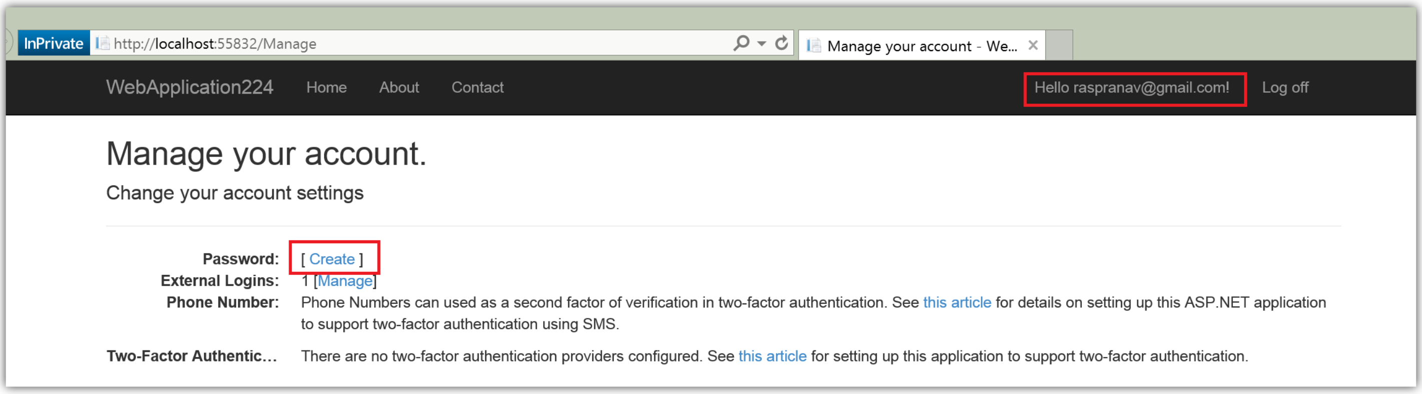
Call `app.UseIdentity` (in `Configure`) before any other external providers.

Optionally set password

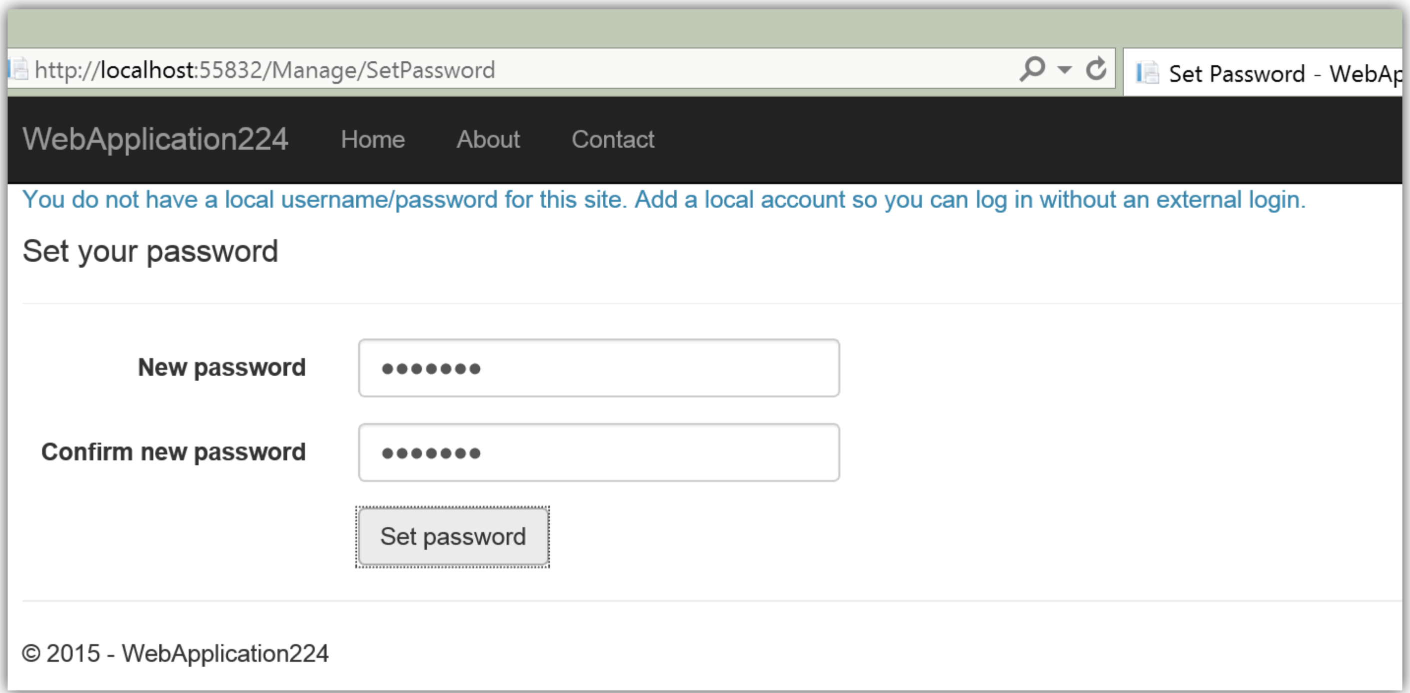
When you register with an external login provider, you do not have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to log in to the web site.

To create a password and sign in using your email that you set during the sign in process with external providers:

Tap the **Hello** link at the top right corner to navigate to the **Manage** view.



Tap **Create**



Set a valid password and you can use this to sign in with your email.

Next steps

This article introduced external authentication and explained the prerequisites required to add external logins to your ASP.NET Core app.

Reference provider-specific pages to configure logins for the providers required by your app.

Configuring Facebook authentication

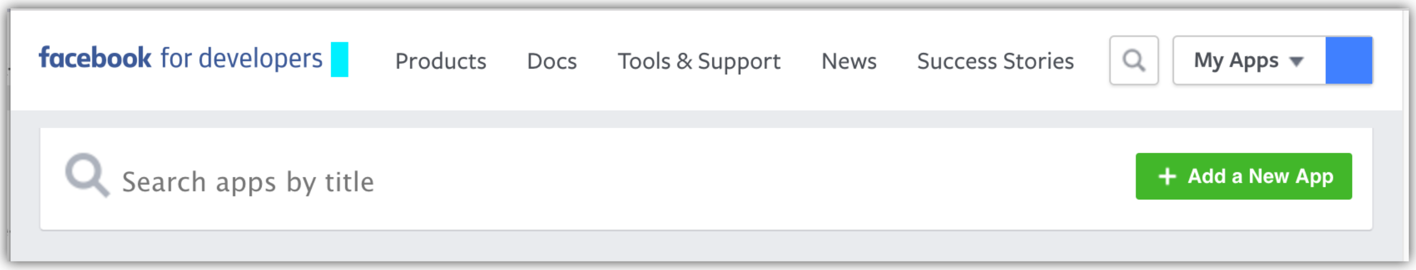
By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

This tutorial shows you how to enable your users to sign in with their Facebook account using a sample ASP.NET Core project created on the previous page. We start by creating a Facebook App ID by following the official steps.

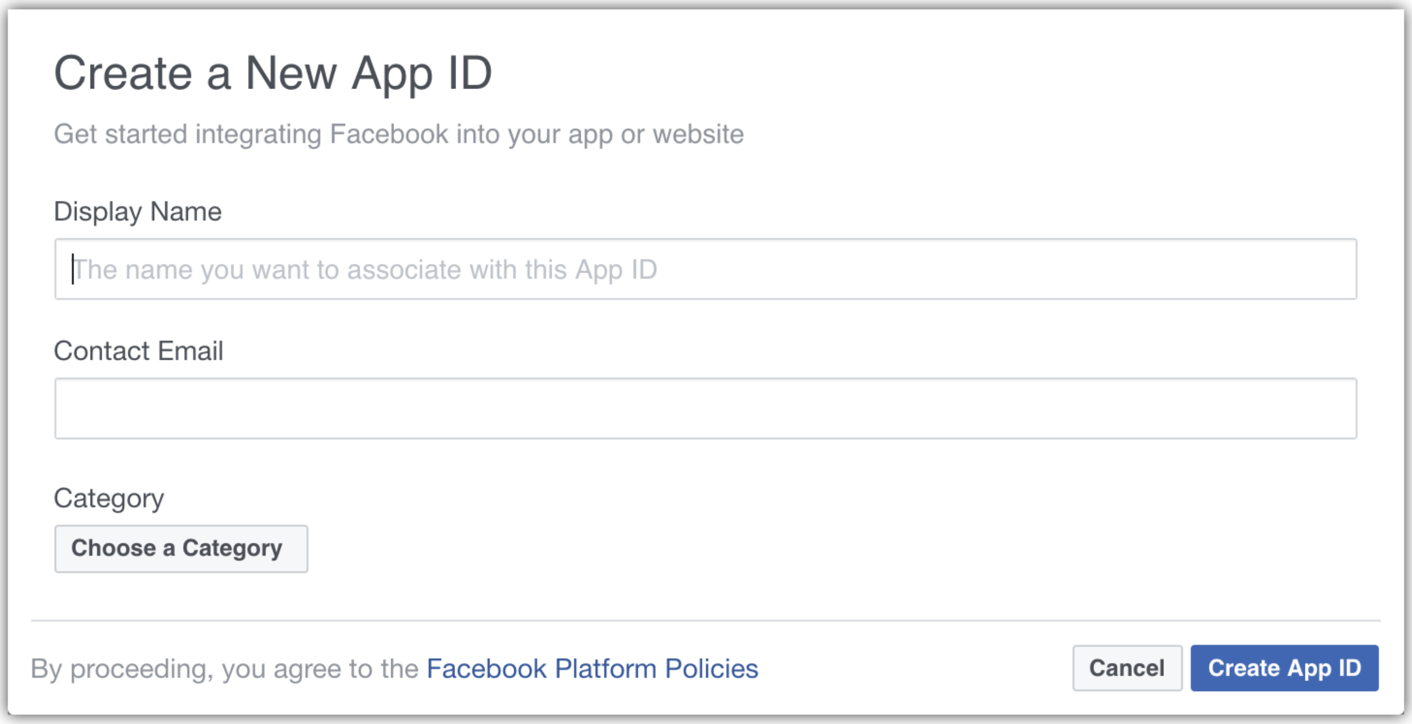
Creating the app in Facebook

Navigate to the Facebook for Developers page and sign in. If you don't already have a Facebook account, use the **Sign up for Facebook** link on the login page to create one.

Tap the **+ Add a New App** button in the upper right corner to create a new App ID. (If this is your first app with Facebook, the text of the button will be **Create a New App**.)



Fill out the form and tap the **Create App ID** button.

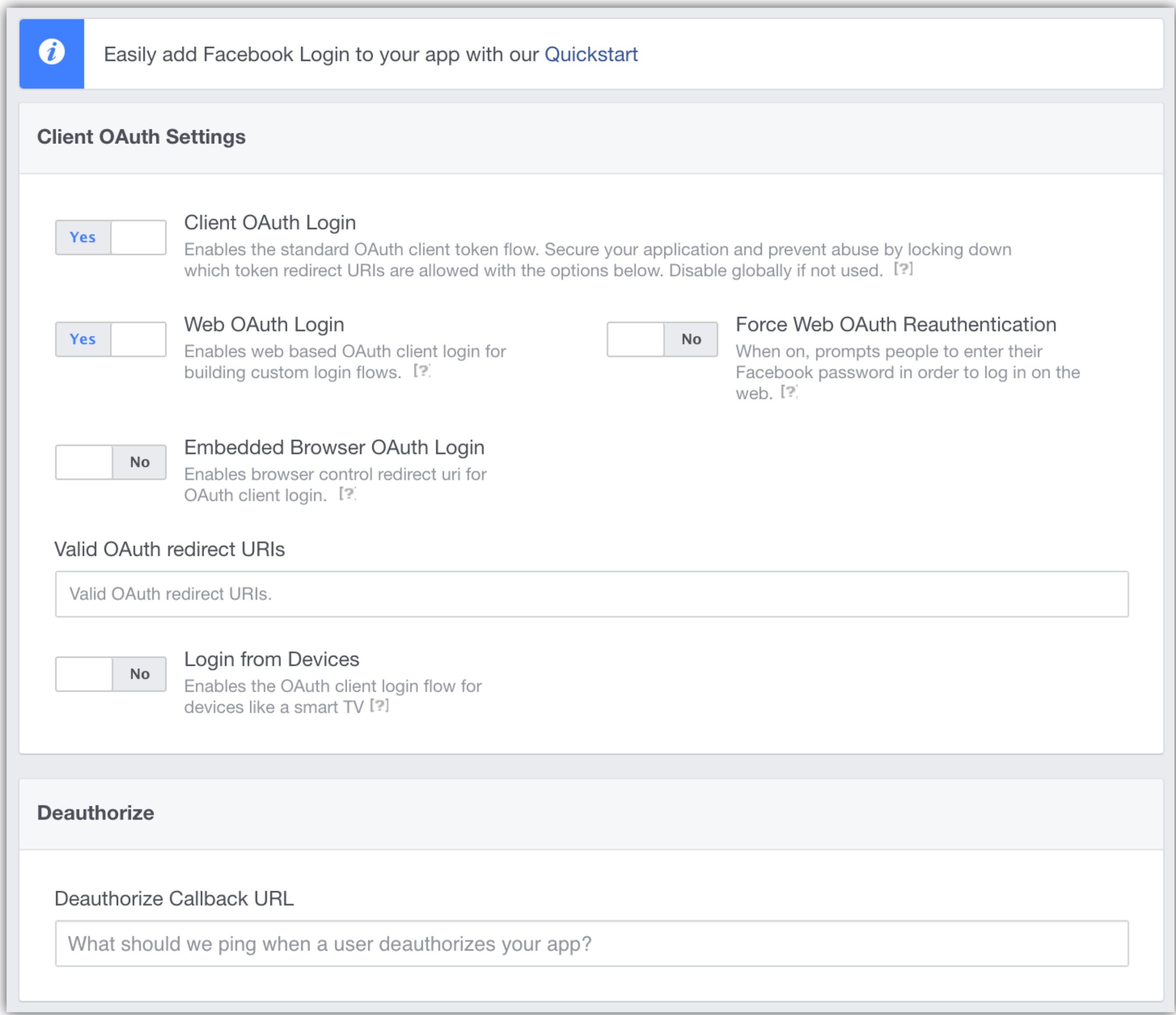


The **Product Setup** page is displayed, letting you select the features for your new app. Click **Get Started** on **Facebook Login**.

Next, a quick start process begins at the **Choose a Platform** screen. This will help you set up client-side login integration, which isn't covered in this tutorial.

To bypass this, click the **Settings** link in the menu at the left.

You are presented with the **Client OAuth Settings** page with some defaults already set.



Enter your base URI with *signin-facebook* appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-facebook`).

Click **Save Changes**.

 Note

When deploying the site you'll need to register a new public url.

Note

You don't need to configure **signin-facebook** as a route in your app. The Facebook middleware automatically intercepts requests at this route and handles them to implement the OAuth flow.

Click the **Dashboard** link in the left navigation.

On this page, you'll need to make a note of your `App ID` and your `App Secret`. Later in this tutorial, you will add both into your ASP.NET Core application.

Storing Facebook App ID and AppSecret

Link sensitive settings like Facebook `App ID` and `App Secret` to your application configuration by using the Secret Manager tool instead of storing them in your configuration file directly, as described in the social login overview page. Execute the following commands in your project working directory:

Set the Facebook AppId

```
dotnet user-secrets set Authentication:Facebook:AppId <app-Id>
```

Set the Facebook AppSecret

```
dotnet user-secrets set Authentication:Facebook:AppSecret <app-secret>
```

The following code reads the configuration values stored by the Secret Manager:

```
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);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=532709
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

Enable Facebook middleware

Note

You will need to use NuGet to install the Microsoft.AspNetCore.Authentication.Facebook package if it hasn't already been installed. Alternatively, execute the following commands in your project directory:

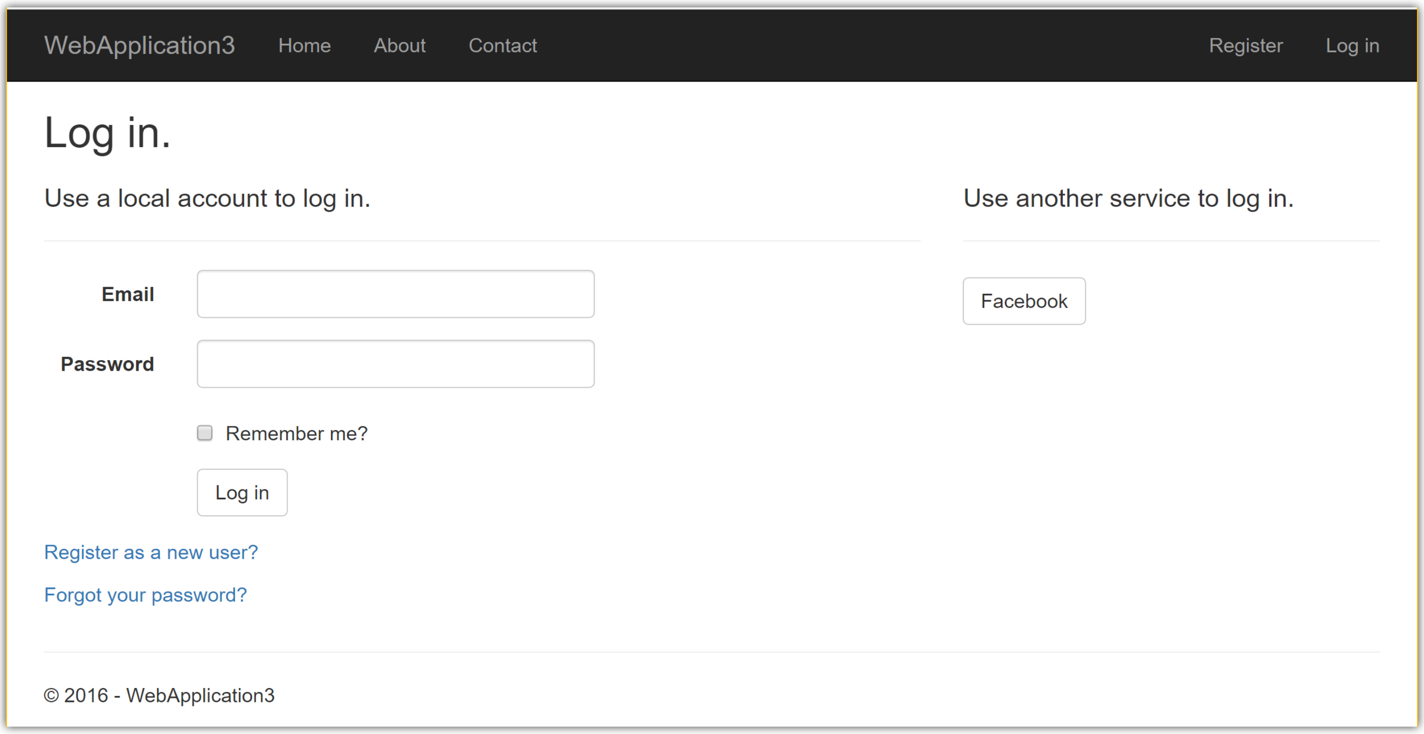
`dotnet add package Microsoft.AspNetCore.Authentication.Facebook`

Add the Facebook middleware in the `Configure` method in `Startup.cs`:

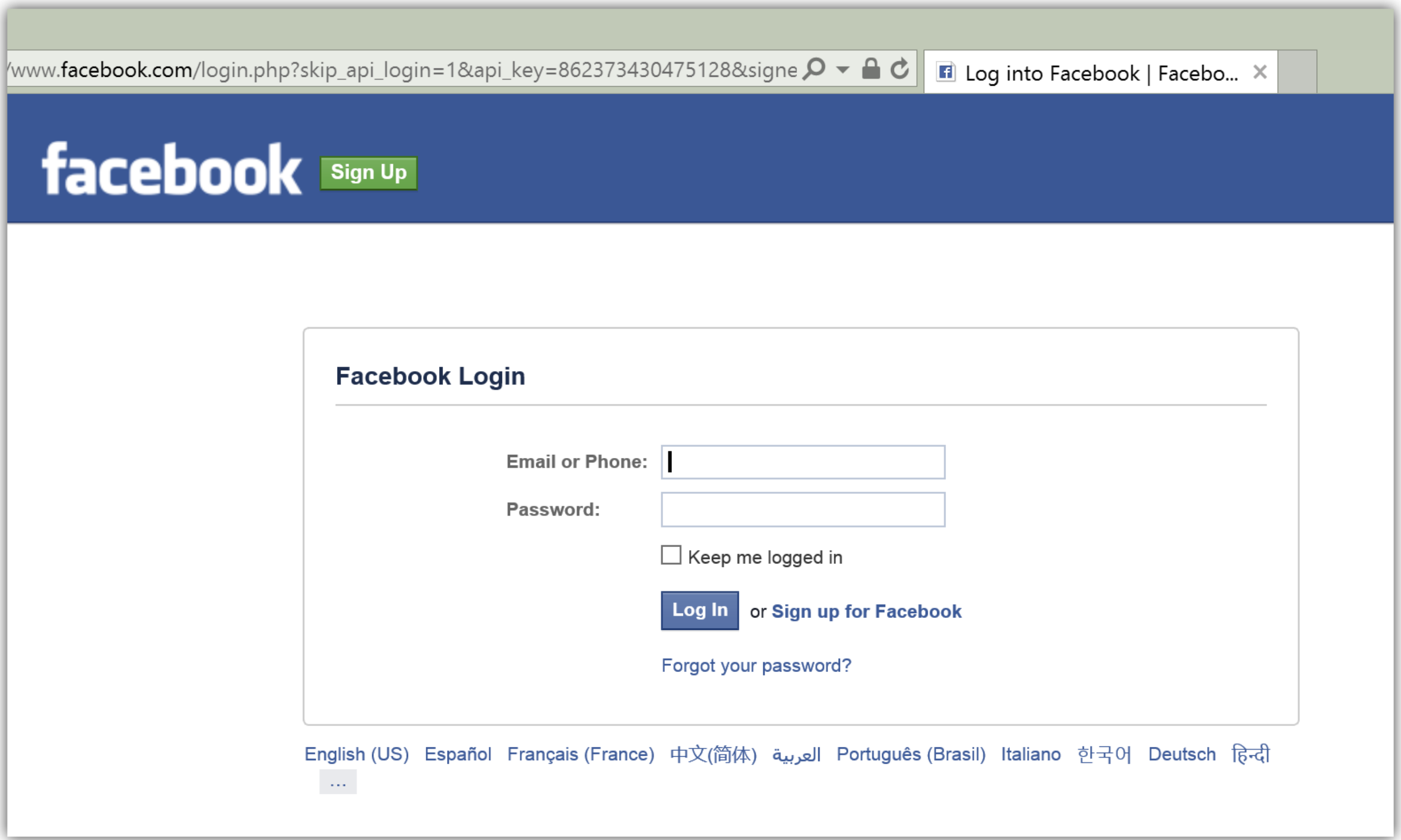
```
app.UseFacebookAuthentication(new FacebookOptions()
{
    AppId = Configuration["Authentication:Facebook:AppId"],
    AppSecret = Configuration["Authentication:Facebook:AppSecret"]
});
```

Sign in with Facebook

Run your application and click **Log in**. You will see an option to sign in with Facebook.

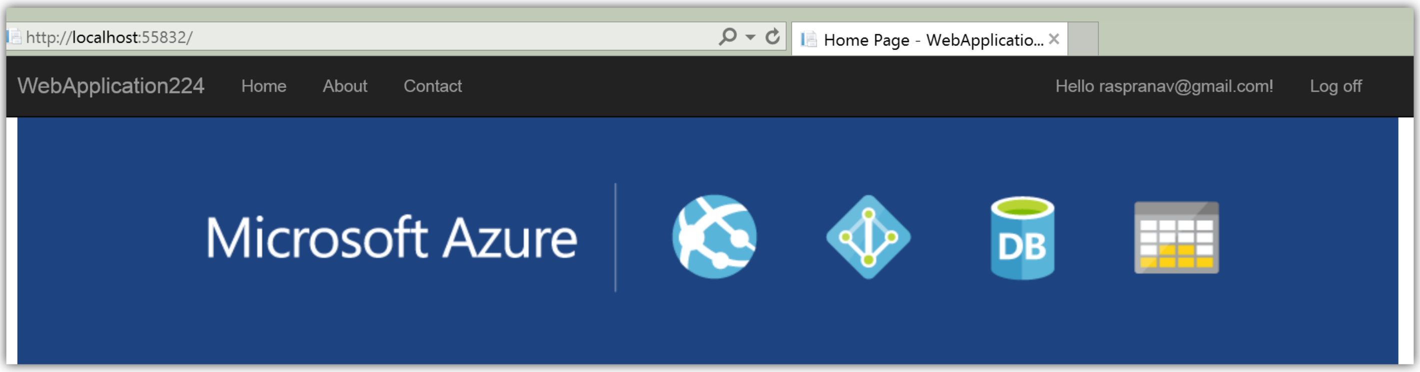


When you click on Facebook, you will be redirected to Facebook for authentication.



Once you enter your Facebook credentials, then you will be redirected back to the web site where you can set your email.

You are now logged in using your Facebook credentials:



Next steps

This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with other providers listed on the previous page.

Once you publish your web site to Azure web app, you should reset the `AppSecret` in the Facebook developer portal.

Set the `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Configuring Twitter authentication

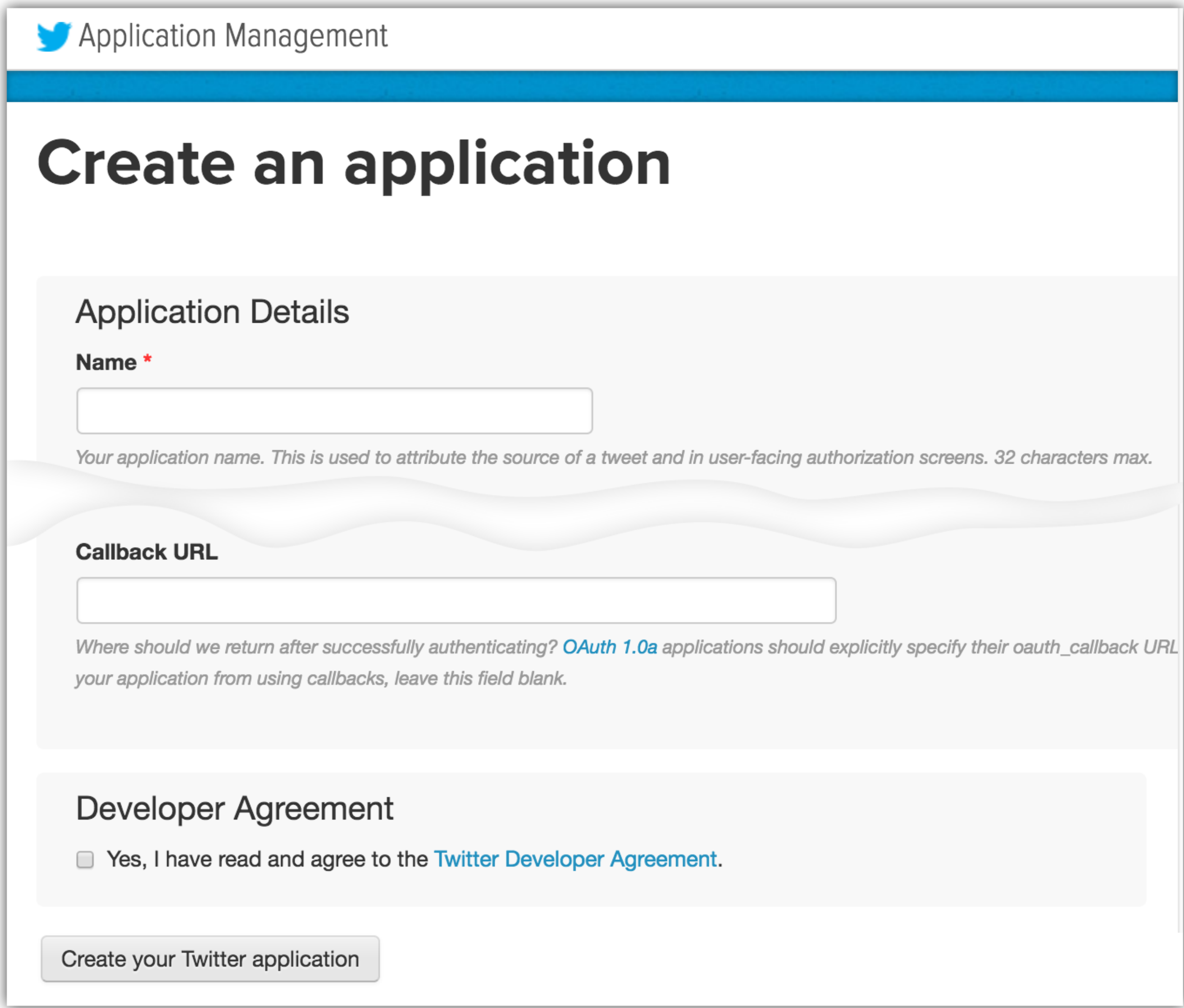
By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

This tutorial shows you how to enable your users to sign in with their Twitter account using a sample ASP.NET Core project created on the previous page.

Creating the app in Twitter

Navigate to https://apps.twitter.com/ and sign in. If you don't already have a Twitter account, use the **Sign up now** link to create one. After signing in, the **Application Management** page is shown:

Tap **Create New App** and fill out the application **Name**:



Enter your current site URL with *signin-twitter* appended into the **Callback URL** field. For example, `https://localhost:44320/signin-twitter`.

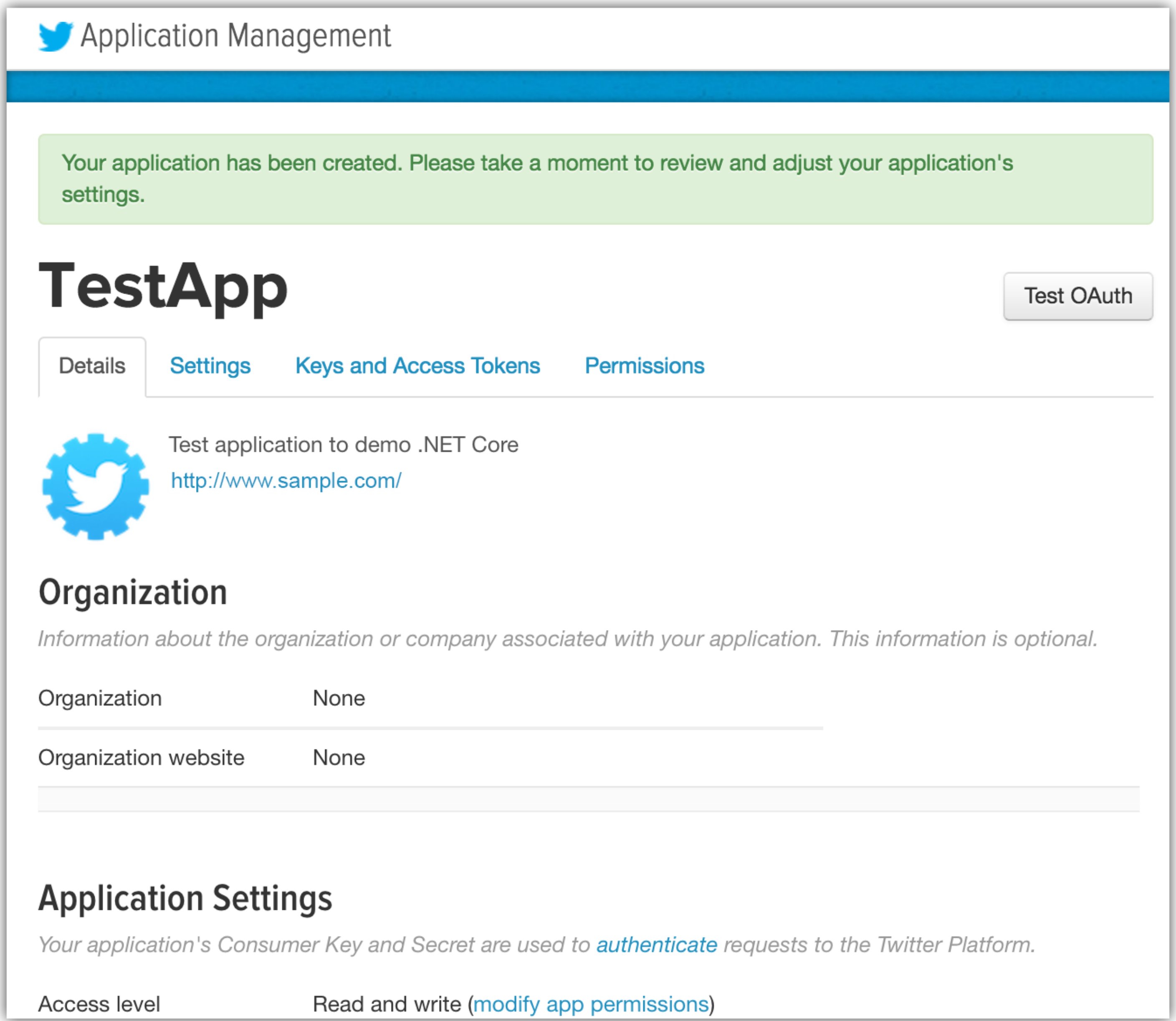
 Note

When deploying the site you'll need to register a new public url.

 Note

You don't need to configure **signin-twitter** as a route in your app. The Twitter middleware automatically intercepts requests at this route and handles them to implement the OAuth flow.

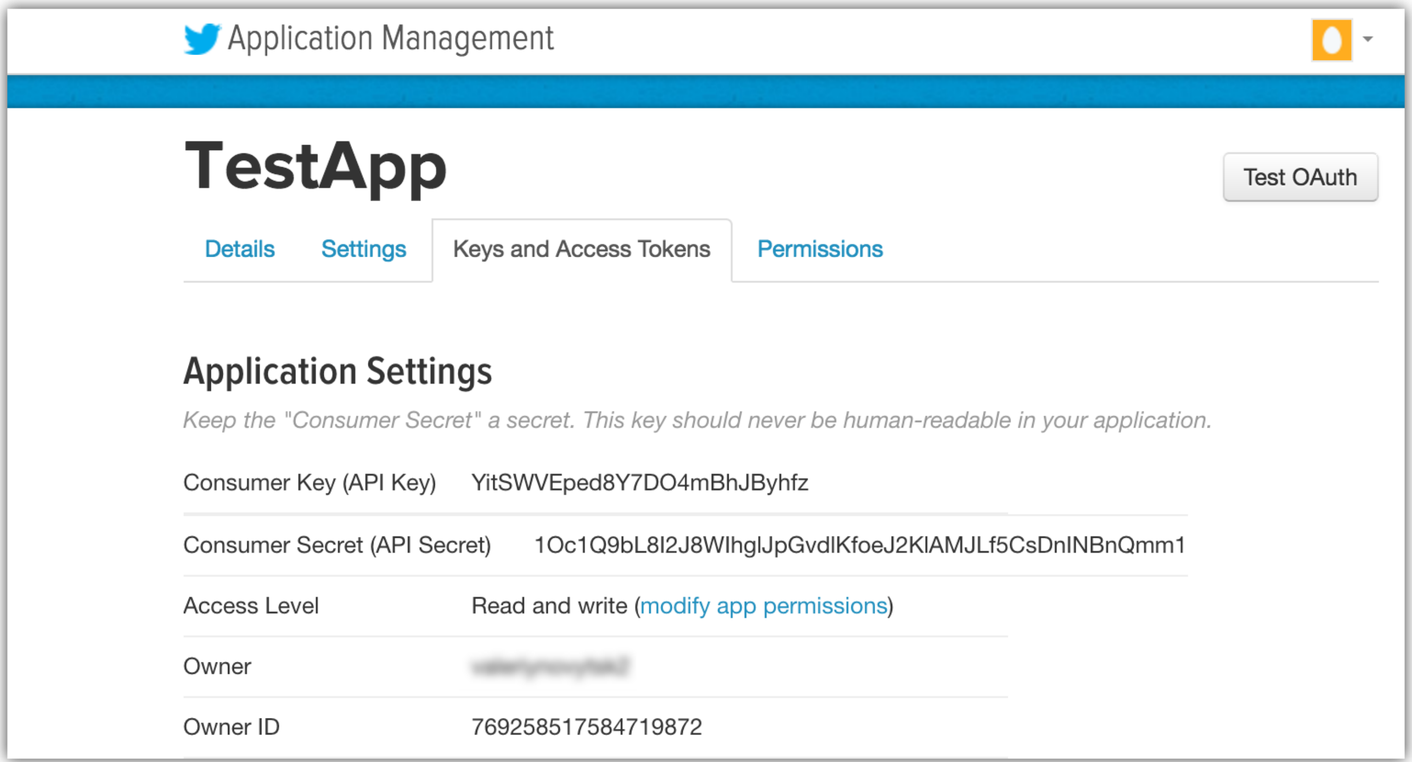
Tap **Create your Twitter application**. New application details are displayed:



Storing Twitter ConsumerKey and ConsumerSecret

Link sensitive settings like Twitter `ConsumerKey` and `ConsumerSecret` to your application configuration by using the Secret Manager tool instead of storing them in your configuration file directly, as described on the social login overview page.

Switch to the **Keys and Access Tokens** tab. Note the `Consumer Key` and `Consumer Secret`:



Execute the following commands in your project working directory to store the Twitter secrets:

```
dotnet user-secrets set Authentication:Twitter:ConsumerKey <consumer-key>
dotnet user-secrets set Authentication:Twitter:ConsumerSecret <consumer-secret>
```

The following code reads the configuration values stored by the Secret Manager:

```
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);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=532709
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

Enable Twitter middleware

Note

Use NuGet to install the Microsoft.AspNetCore.Authentication.Twitter package if it hasn't already been installed. Alternatively, execute the following commands in your project directory:

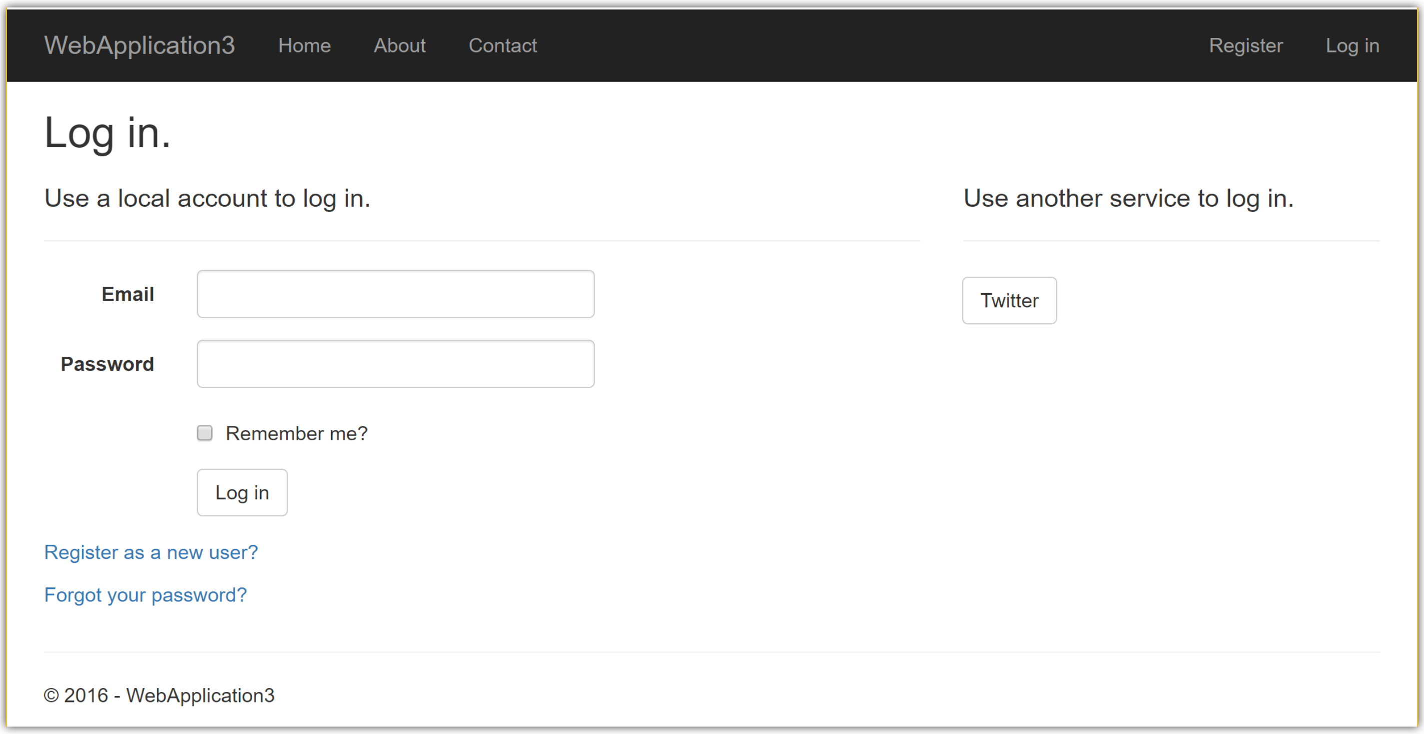
`dotnet add package Microsoft.AspNetCore.Authentication.Twitter`

Add the Twitter middleware in the `Configure` method in `Startup.cs`:

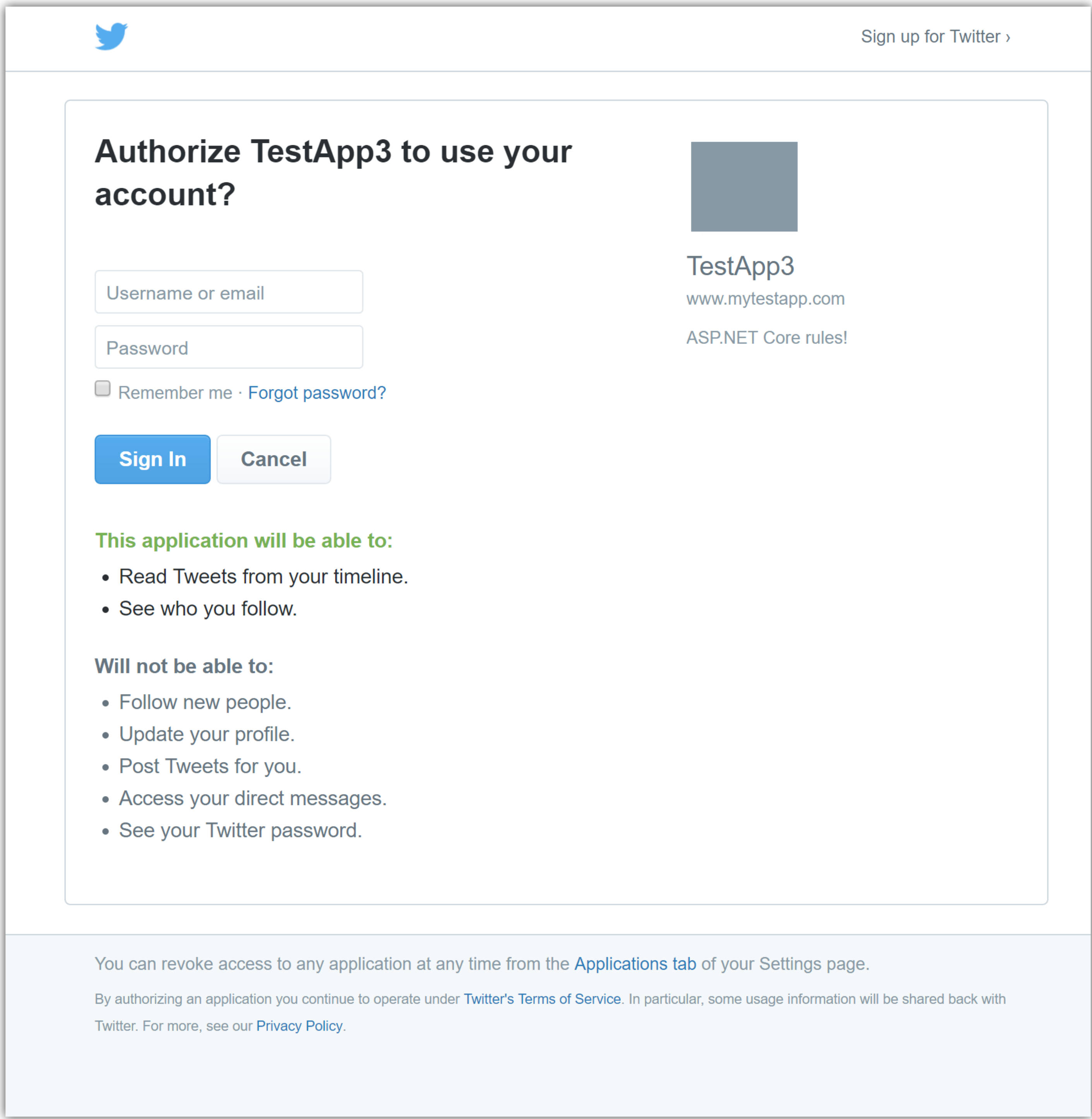
```
app.UseTwitterAuthentication(new TwitterOptions()
{
    ConsumerKey = Configuration["Authentication:Twitter:ConsumerKey"],
    ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"]
});
```

Sign in with Twitter

Run your application and click **Log in**. An option to sign in with Twitter appears:

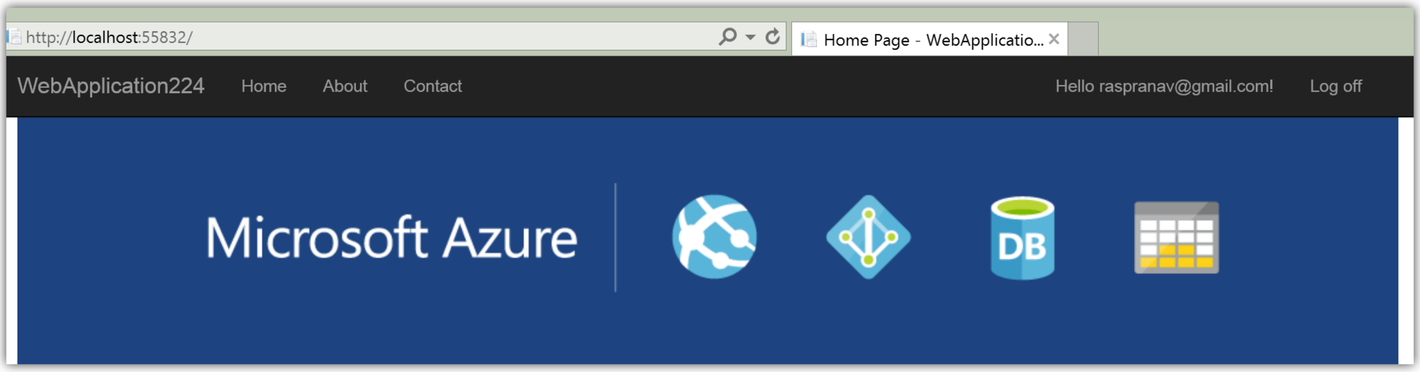


Clicking on **Twitter** redirects to Twitter for authentication:



After entering your Twitter credentials, you are redirected back to the web site where you can set your email.

You are now logged in using your Twitter credentials:



Next steps

This article showed how you can authenticate with Twitter. You can follow a similar approach to authenticate with other providers listed on the previous page.

Once you publish your web site to Azure web app, you should reset the `ConsumerSecret` in the Twitter developer portal.

Set the `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

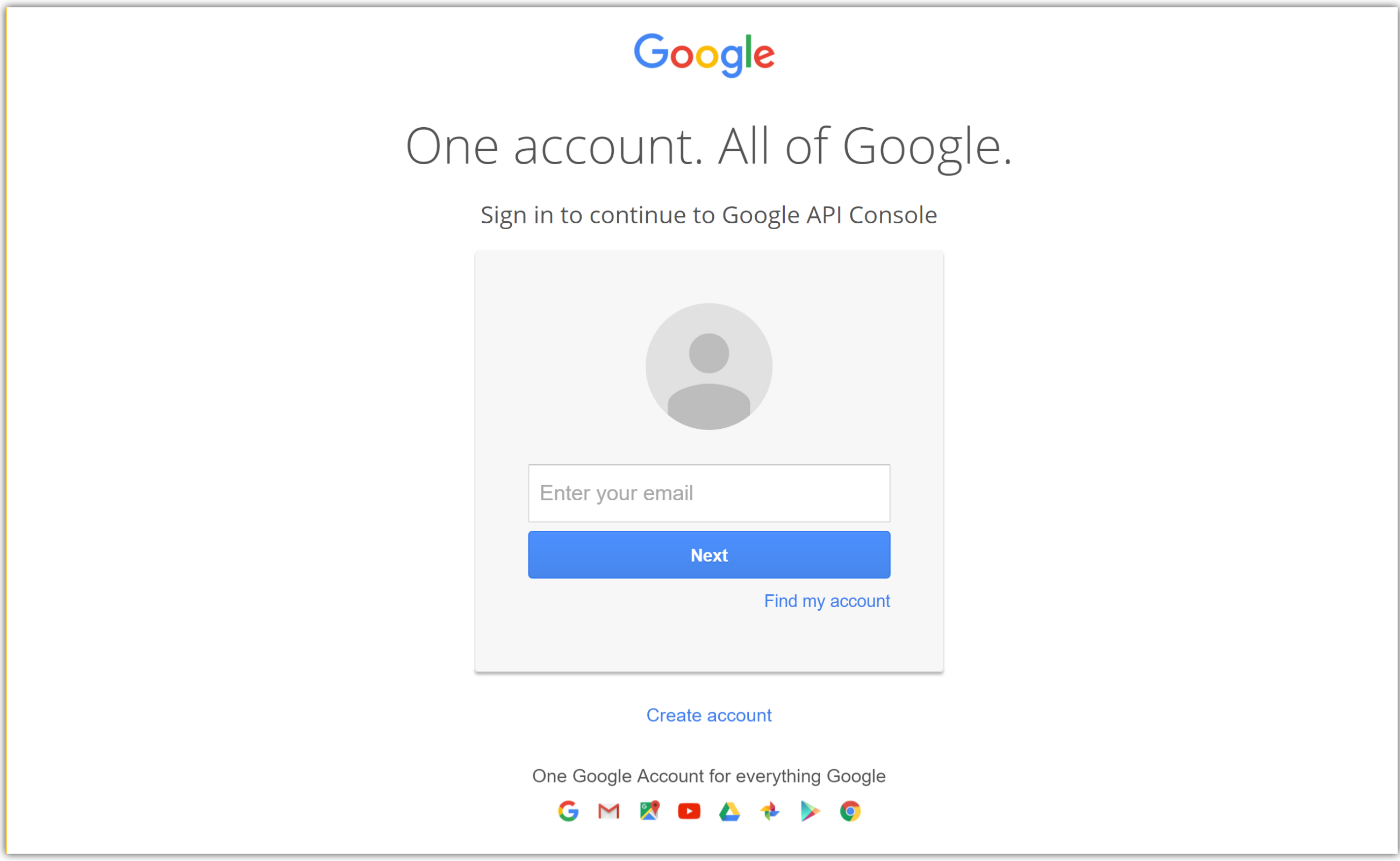
Configuring Google authentication

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

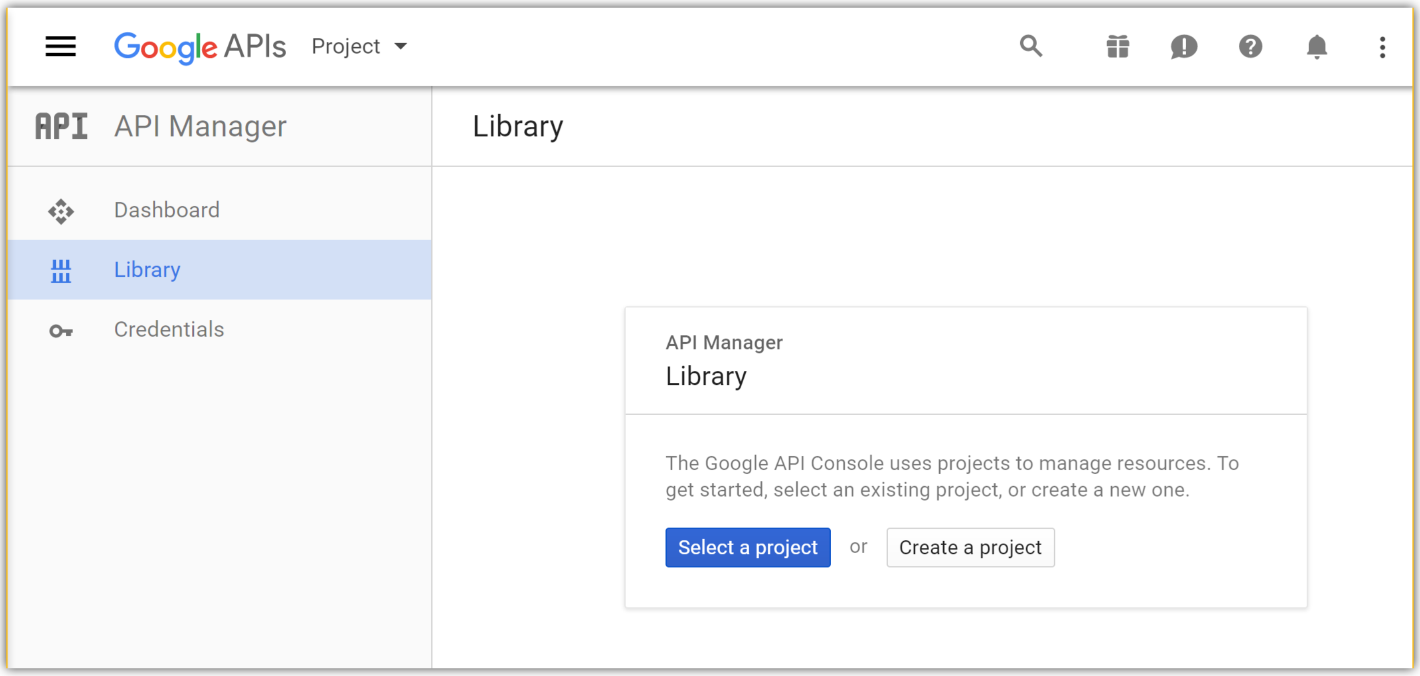
This tutorial shows you how to enable your users to sign in with their Google+ account using a sample ASP.NET Core project created on the previous page. We start by following the official steps to create a new app in Google API Console.

Creating the app in Google API Console

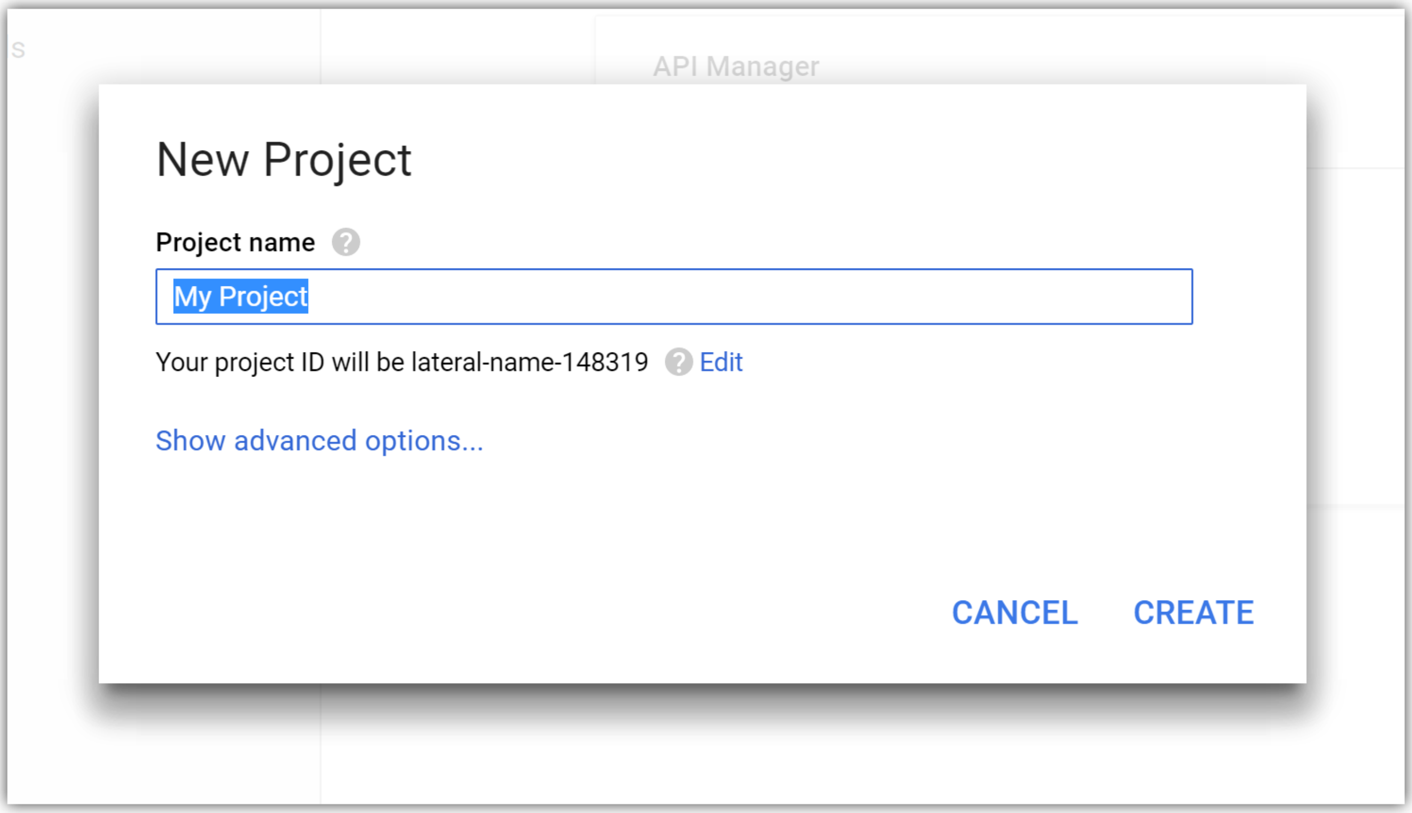
Navigate to https://console.developers.google.com/projectselector/apis/library and sign in. If you don't already have a Google account, use the **Create account** link to create one:



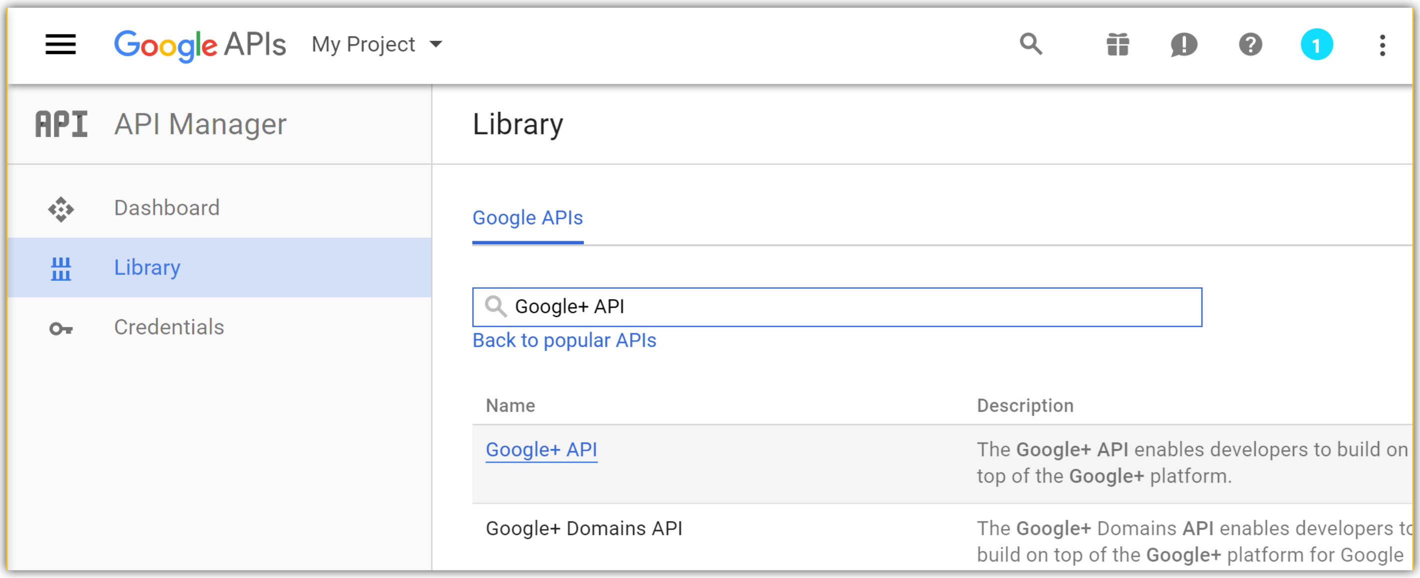
You are redirected to API Manager Library page:



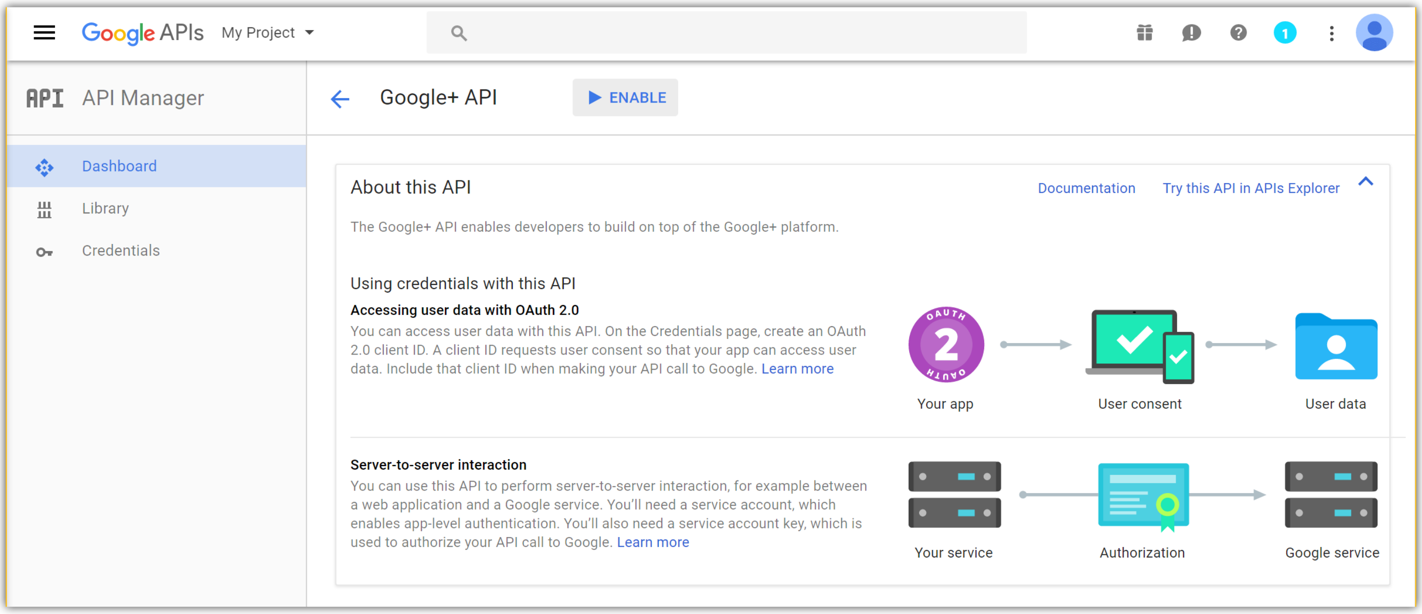
Tap **Create a project** and enter your application name:



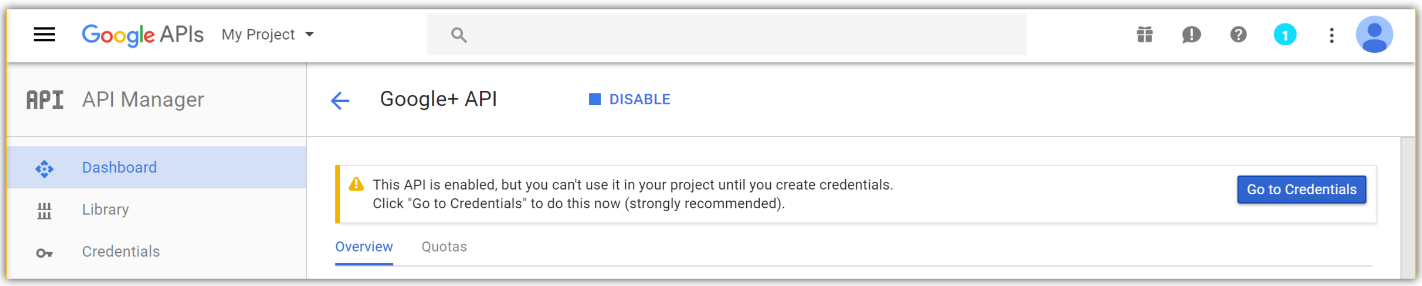
After accepting the dialog, you are redirected back to the Library page allowing you to choose features for your new app. Find **Google+ API** in the list and click on its link to add the API feature:



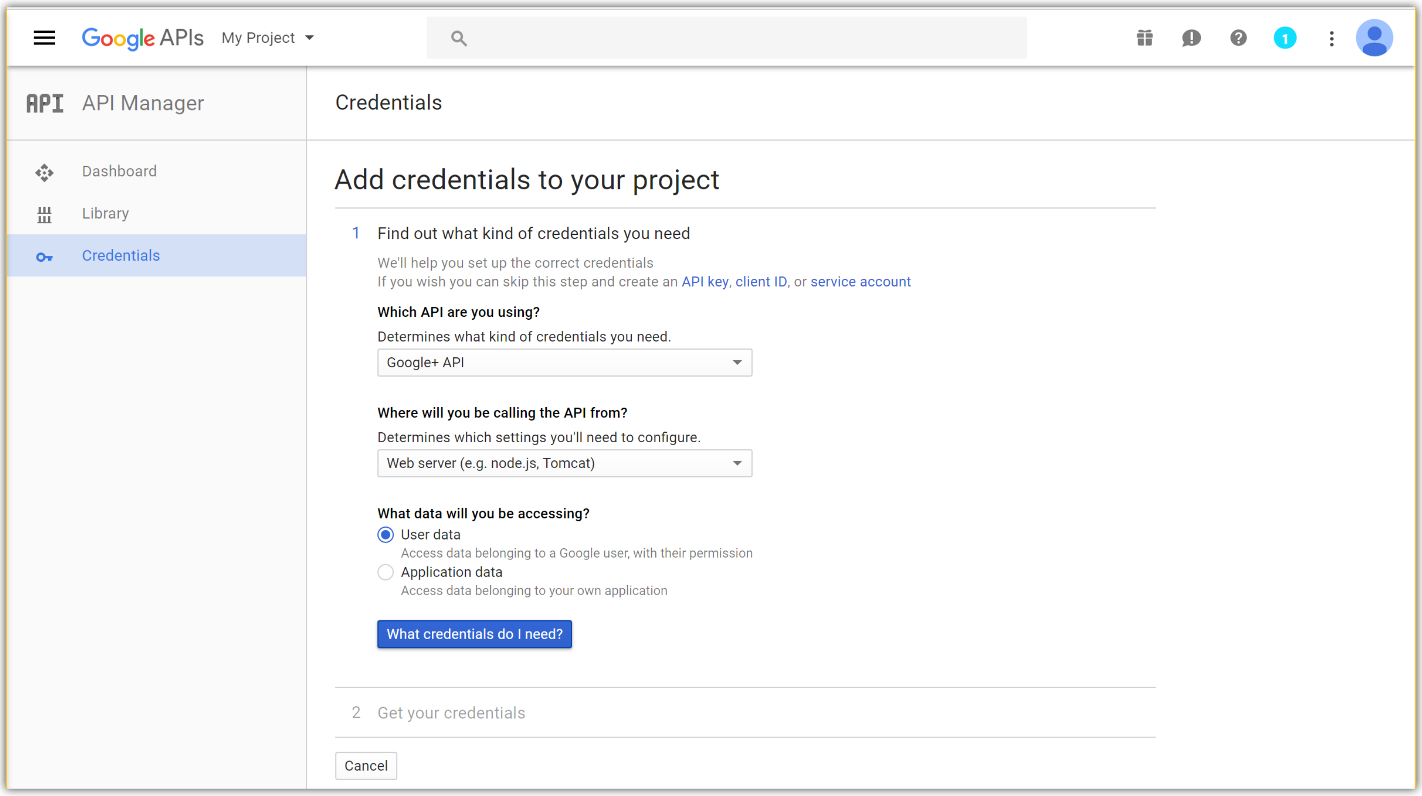
The page for the newly added API is displayed. Tap **Enable** to add Google+ sign in feature to your app:



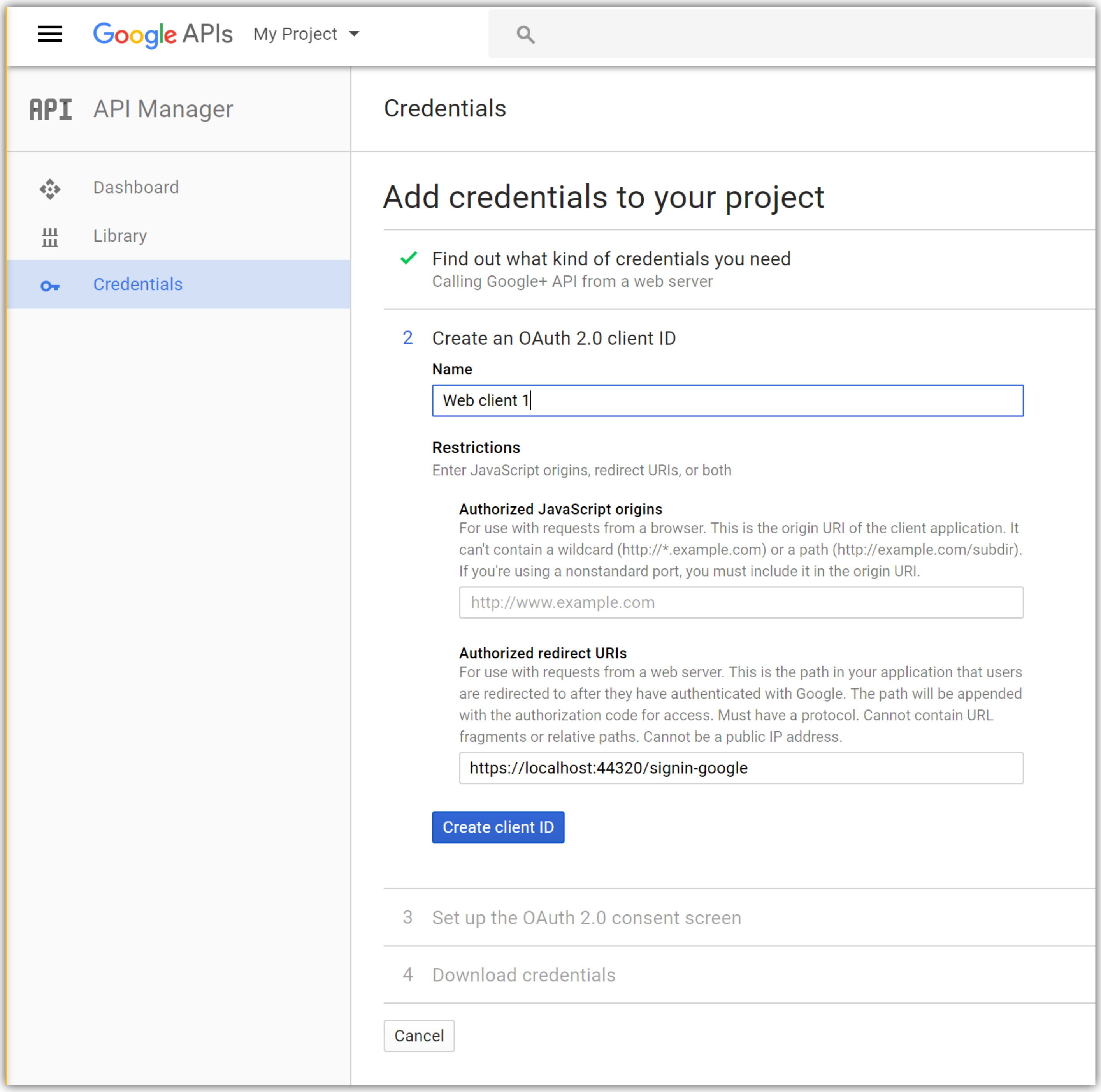
After enabling the API, tap **Go to Credentials** to configure the secrets:



Choose:
Google+ API
Web server (e.g. node.js, Tomcat), and
User data:



Tap **What credentials do I need?** which takes you to the second step of app configuration:



Because we are creating a Google+ project with just one feature (sign in), we can enter the same **Name** for the OAuth 2.0 client ID as the one we used for the project.

Enter your current site URL with *signin-google* appended into the **Authorized redirect URIs** field. For example, `https://localhost:44320/signin-google`.

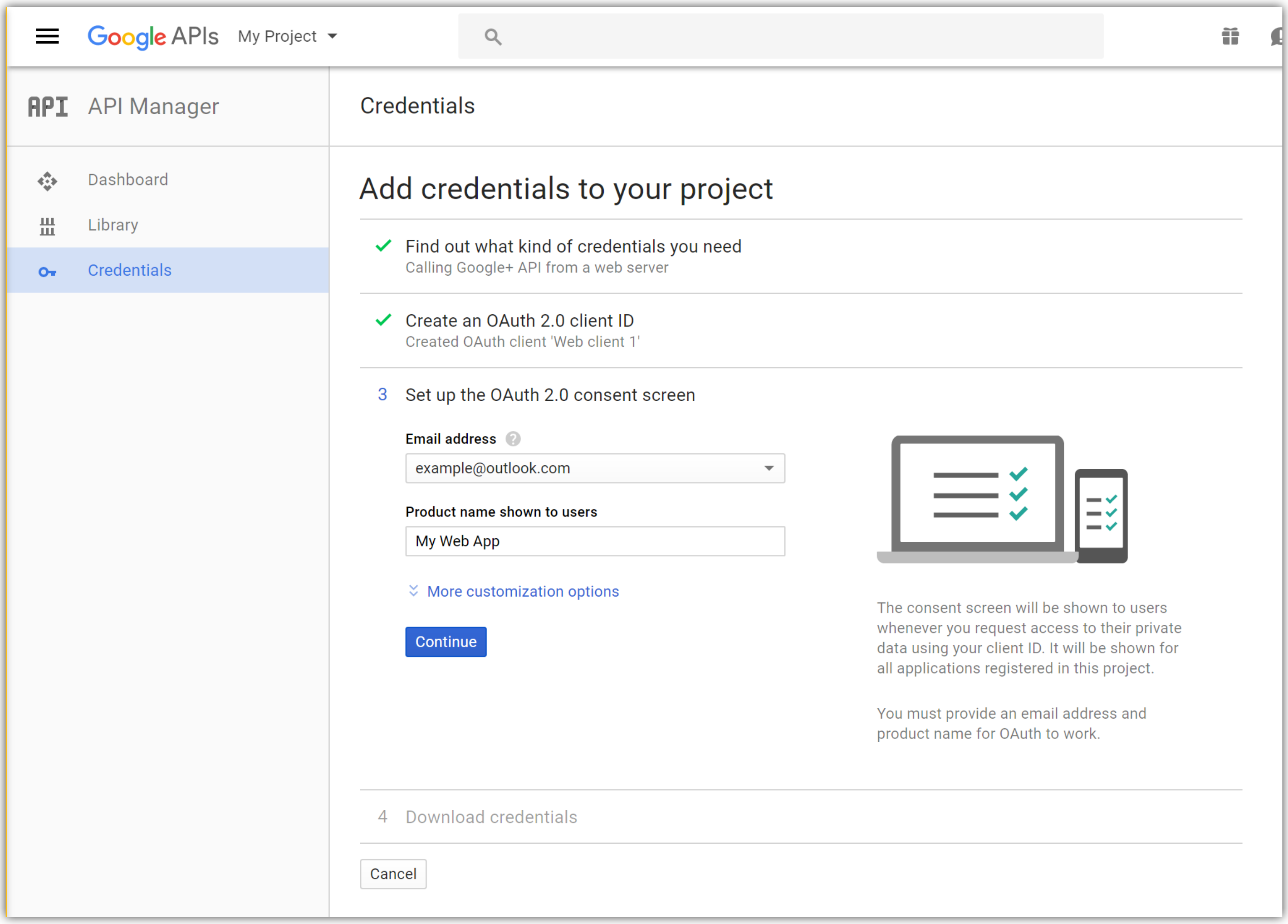
▯ Note

When deploying the site you'll need to register a new public url.

▯ Note

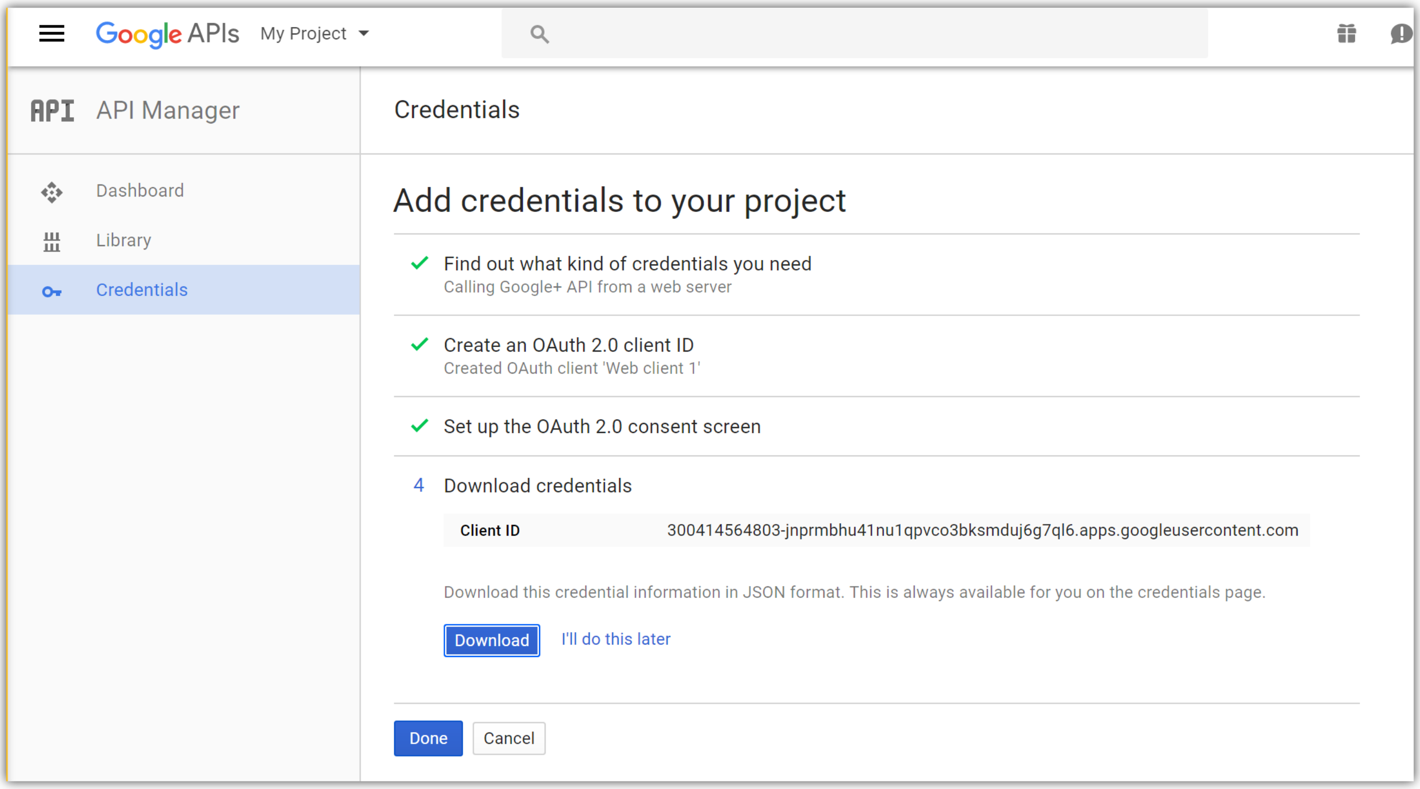
You don't need to configure **signin-google** as a route in your app. The Google middleware automatically intercepts requests at this route and handles them to implement the OAuth flow.

Tap **Create client ID**, which takes you to the third step:



Enter your public facing **Email address** and the **Product name** shown for your app when Google+ prompts the user to sign in.

Tap **Continue** to proceed to the last step:



Tap **Download** to save a JSON file with application secrets, and **Done** to complete creation of the new app.

Storing Google ClientID and ClientSecret

Link sensitive settings like Google `ClientID` and `ClientSecret` to your application configuration by using the Secret Manager tool instead of storing them in your configuration file directly, as described in the social authentication overview page.

Open the JSON file downloaded in the last step. Note the `client_id` and `client_secret` values present in the JSON structure.

Execute the following commands in your project working directory to store the Google secrets:

```
dotnet user-secrets set Authentication:Google:ClientID <client_id>
dotnet user-secrets set Authentication:Google:ClientSecret <client-secret>
```

The following code reads the configuration values stored by the Secret Manager:

```
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);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=532709
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

Enable Google middleware

Note

Use NuGet to install the Microsoft.AspNetCore.Authentication.Google package if it hasn't already been installed. Alternatively, execute the following commands in your project directory:

`dotnet add package Microsoft.AspNetCore.Authentication.Google`

Add the Google middleware in the `Configure` method in `Startup.cs`:

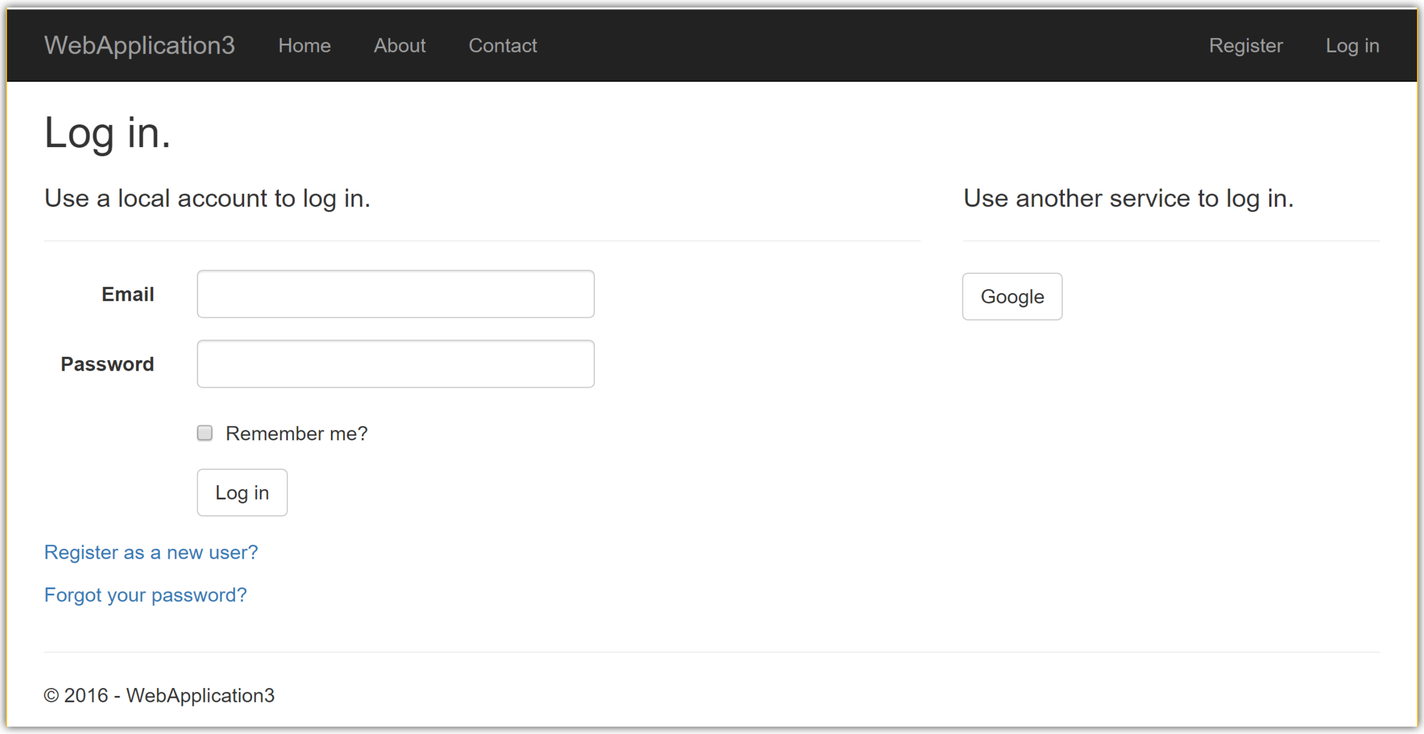
```
app.UseGoogleAuthentication(new GoogleOptions()
{
    ClientId = Configuration["Authentication:Google:ClientId"],
    ClientSecret = Configuration["Authentication:Google:ClientSecret"]
});
```

Note

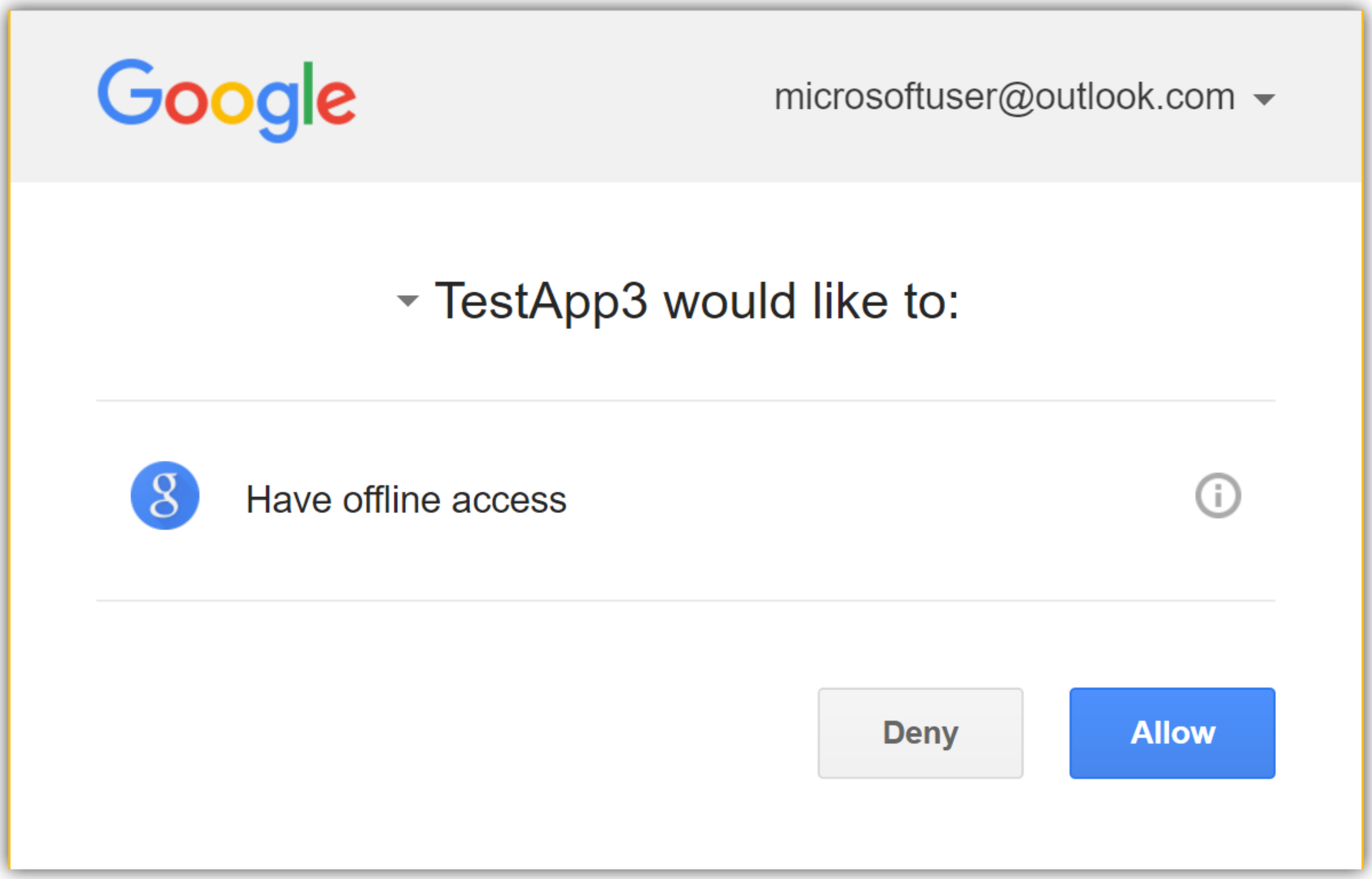
Call `UseIdentity` before you call `UseGoogleAuthentication`. See the social authentication overview page.

Sign in with Google

Run your application and click **Log in**. An option to sign in with Google appears:

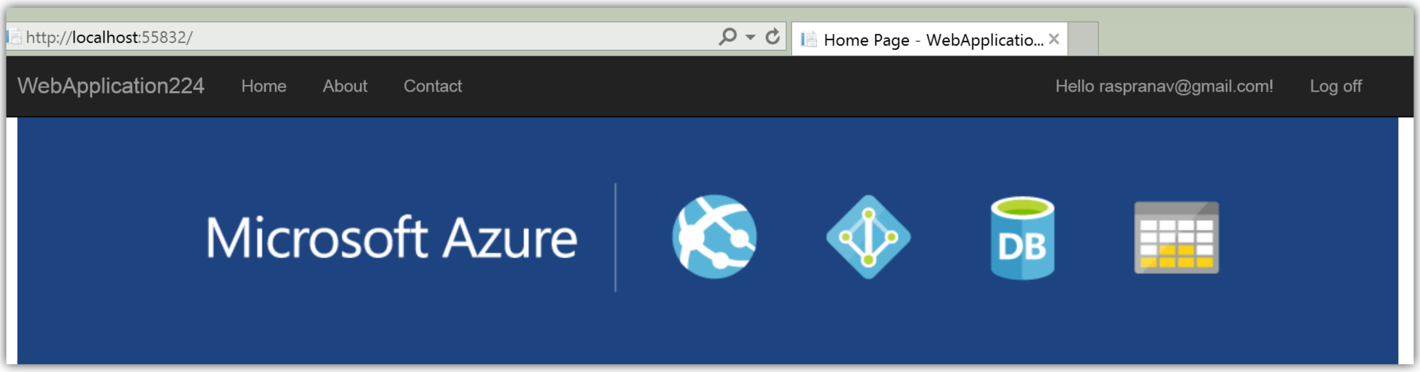


When you click on Google, you are redirected to Google for authentication:



After entering your Google credentials, then you are redirected back to the web site where you can set your email.

You are now logged in using your Google credentials:



Note

If instead you receive a `403 (Forbidden)` error page from your own app when running in development mode (or break into the debugger with the same error), ensure that **Google+ API** has been enabled in the **API Manager Library** by following the steps listed earlier on this page. If the sign in doesn't work and you aren't getting any errors, switch to development mode to make the issue easier to debug.

Next steps

This article showed how you can authenticate with Google. You can follow a similar approach to authenticate with other providers listed on the previous page.

Once you publish your web site to Azure web app, you should reset the `ClientSecret` in the Google API Console.

Set the `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

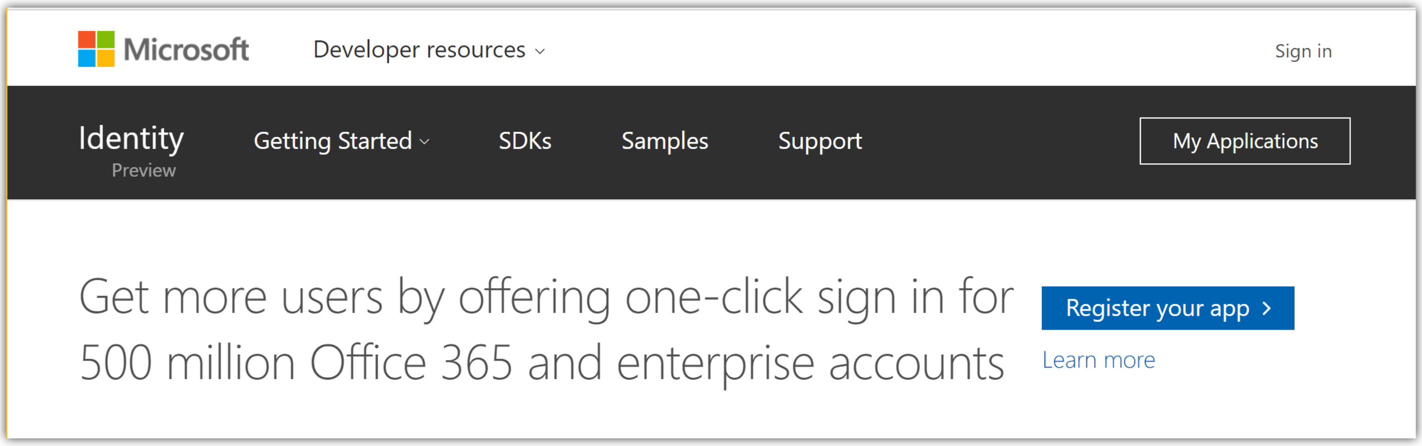
Configuring Microsoft Account authentication

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

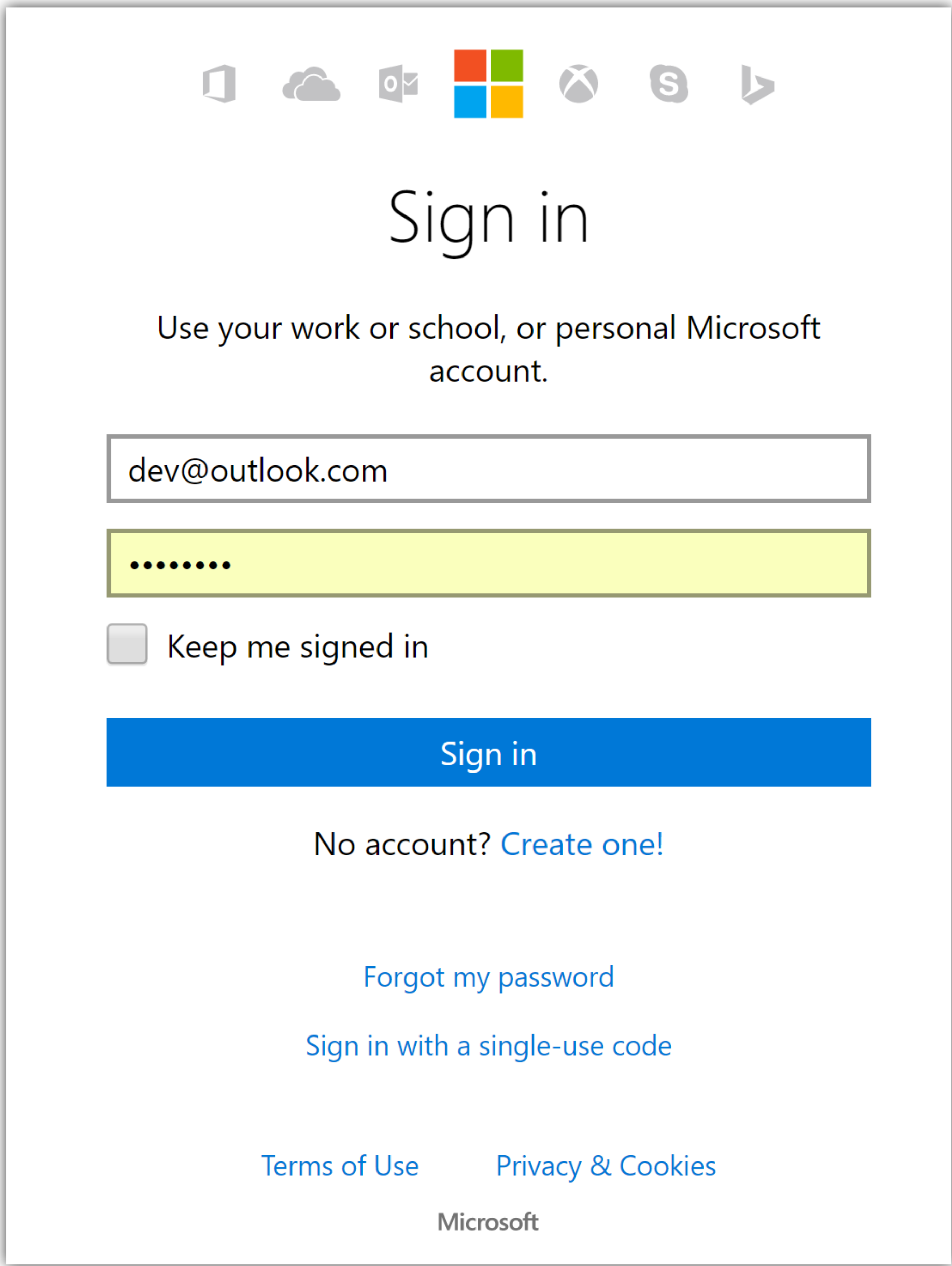
This tutorial shows you how to enable your users to sign in with their Microsoft account using a sample ASP.NET Core project created on the previous page.

Creating the app in Microsoft Developer Portal

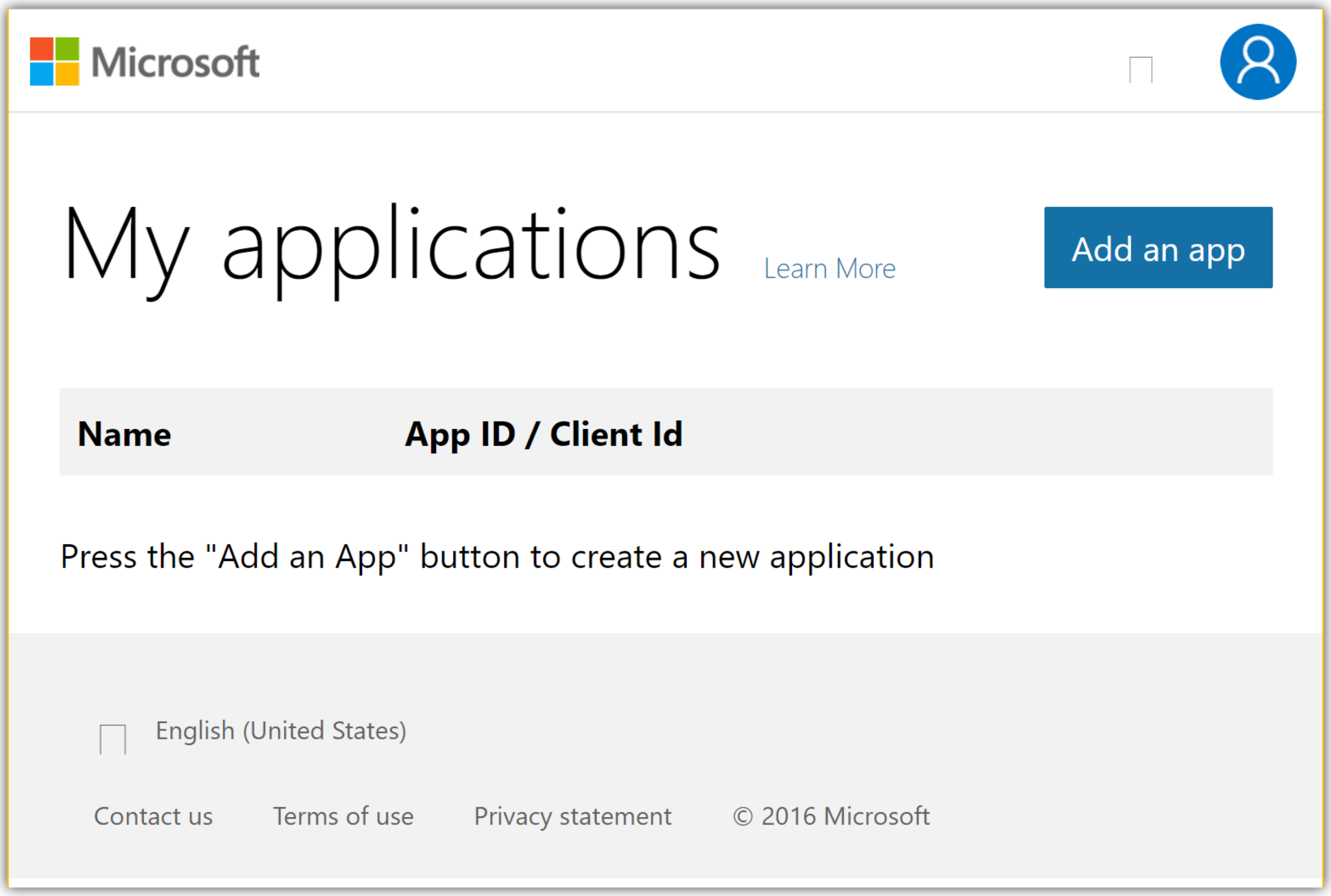
Navigate to https://apps.dev.microsoft.com:



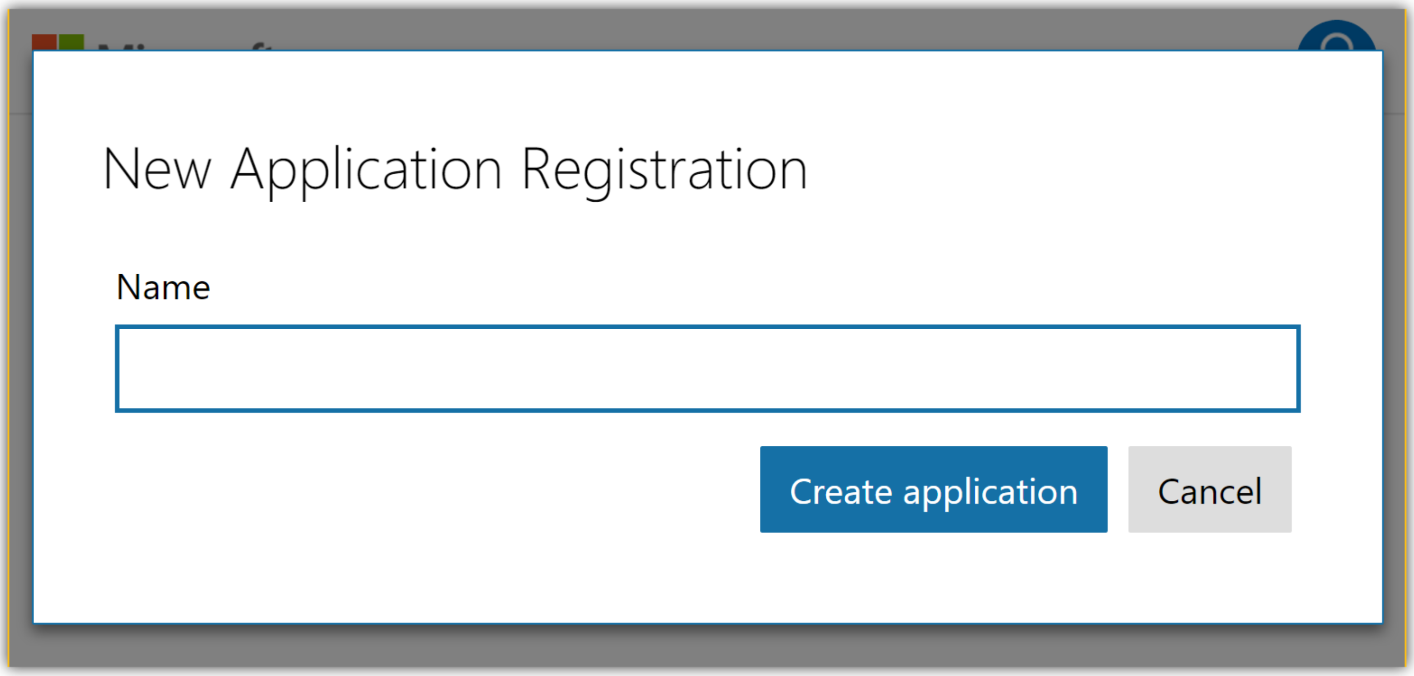
Tap **sign in**:



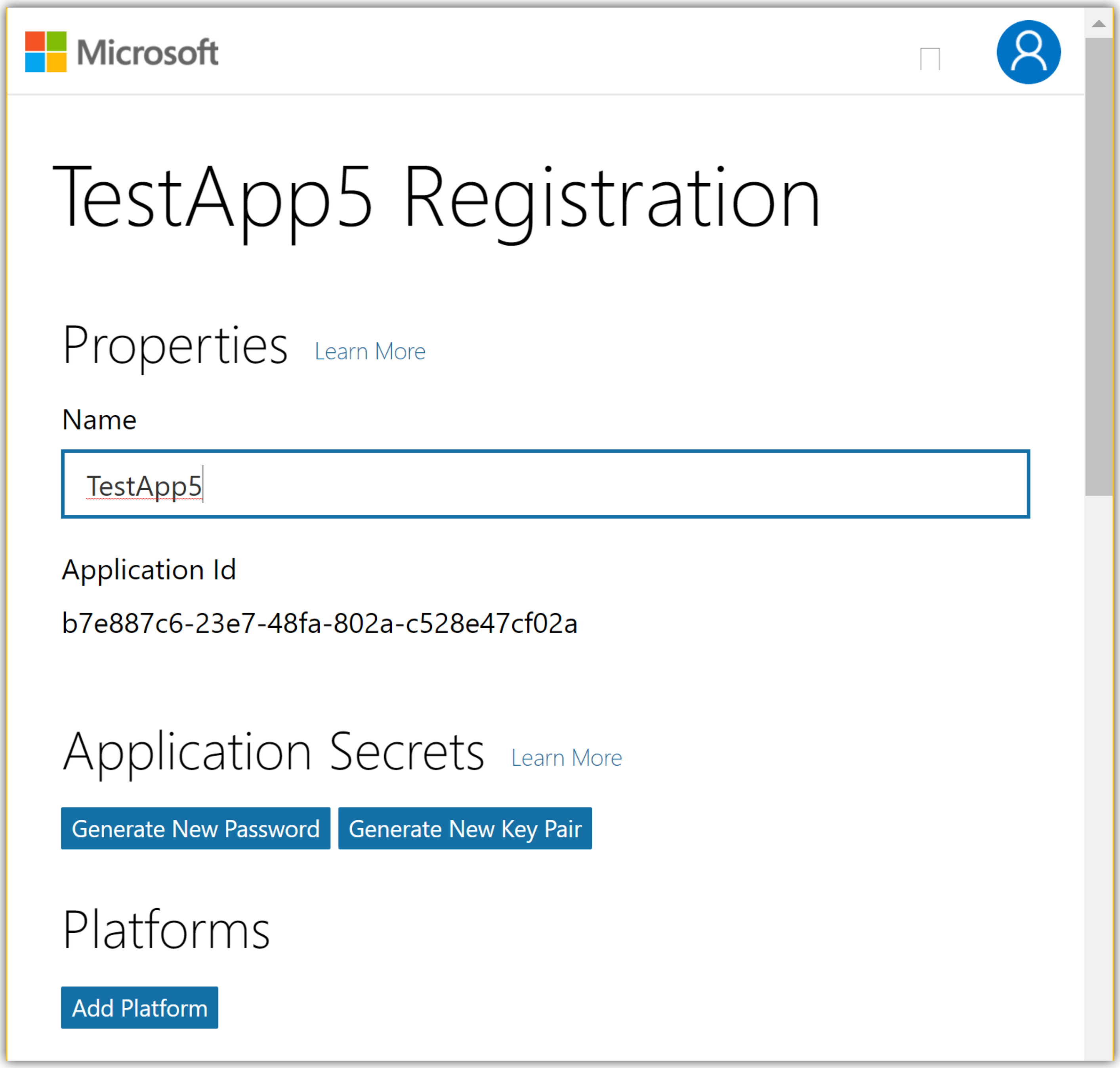
If you don't already have a Microsoft account, tap **Create one!**. After signing in you are redirected to **My applications** page:



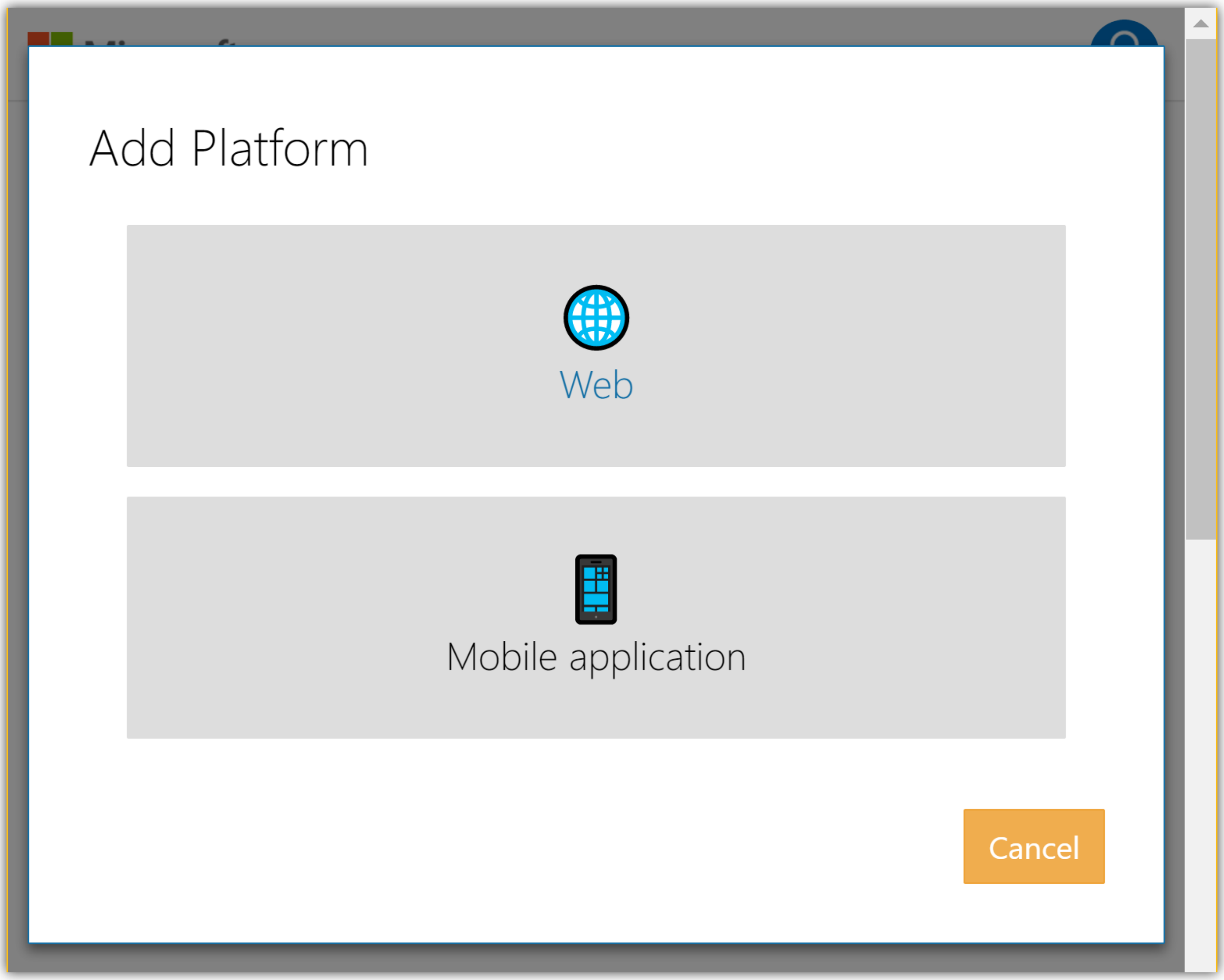
Tap **Add an app** in the upper right corner and enter your **application name**:



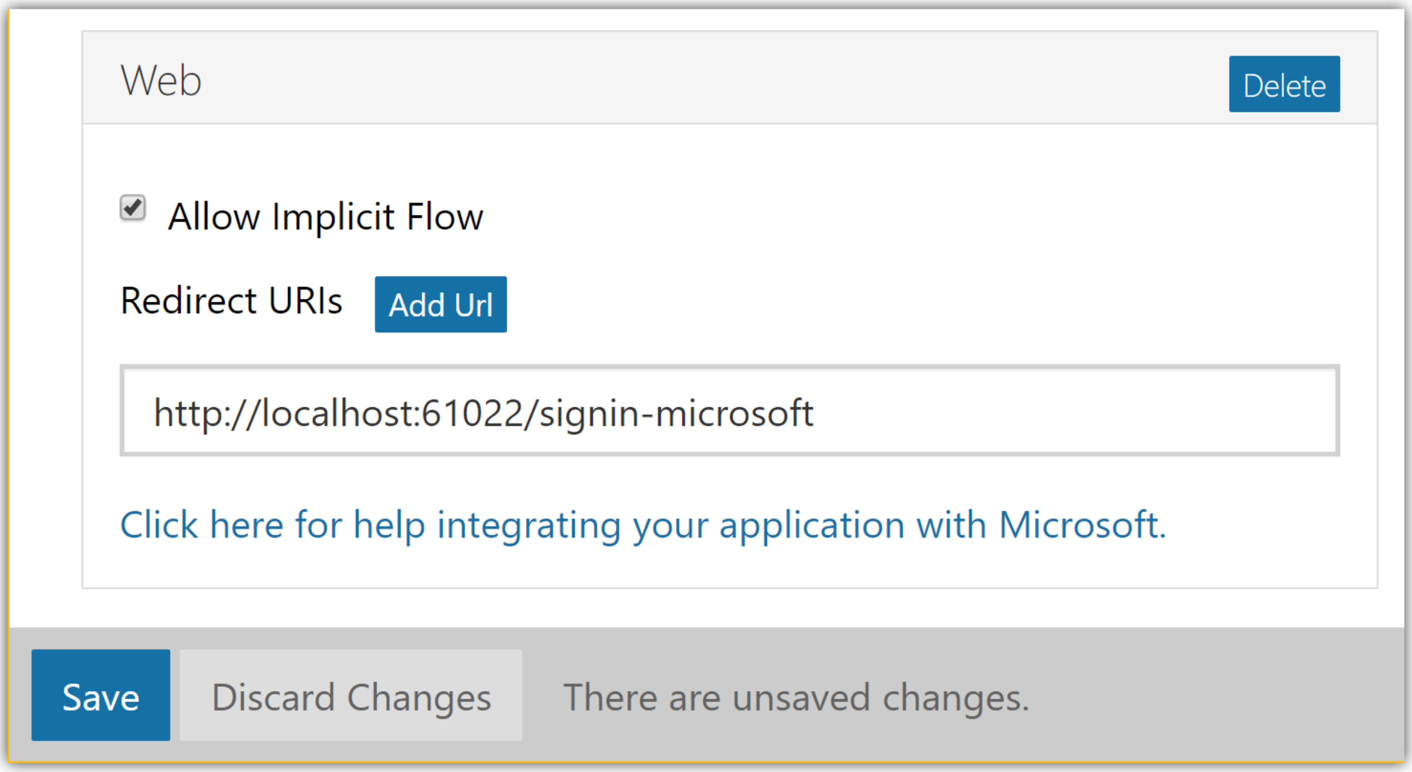
The **Registration** page is displayed:



Tap **Add Platform** in the **Platforms** section and select the **Web** platform:



In the new **Web** platform section, enter your current site URL with *signin-microsoft* appended into the **Redirect URIs** field. For example, `https://localhost:44320/signin-microsoft`:



▯ Note

When deploying the site you'll need to register a new public url.

▯ Note

You don't need to configure **signin-microsoft** as a route in your app. The Microsoft Account middleware automatically intercepts requests at this route and handles them to implement the OAuth flow.

Don't forget to tap **Add Url** to ensure the Url was added.

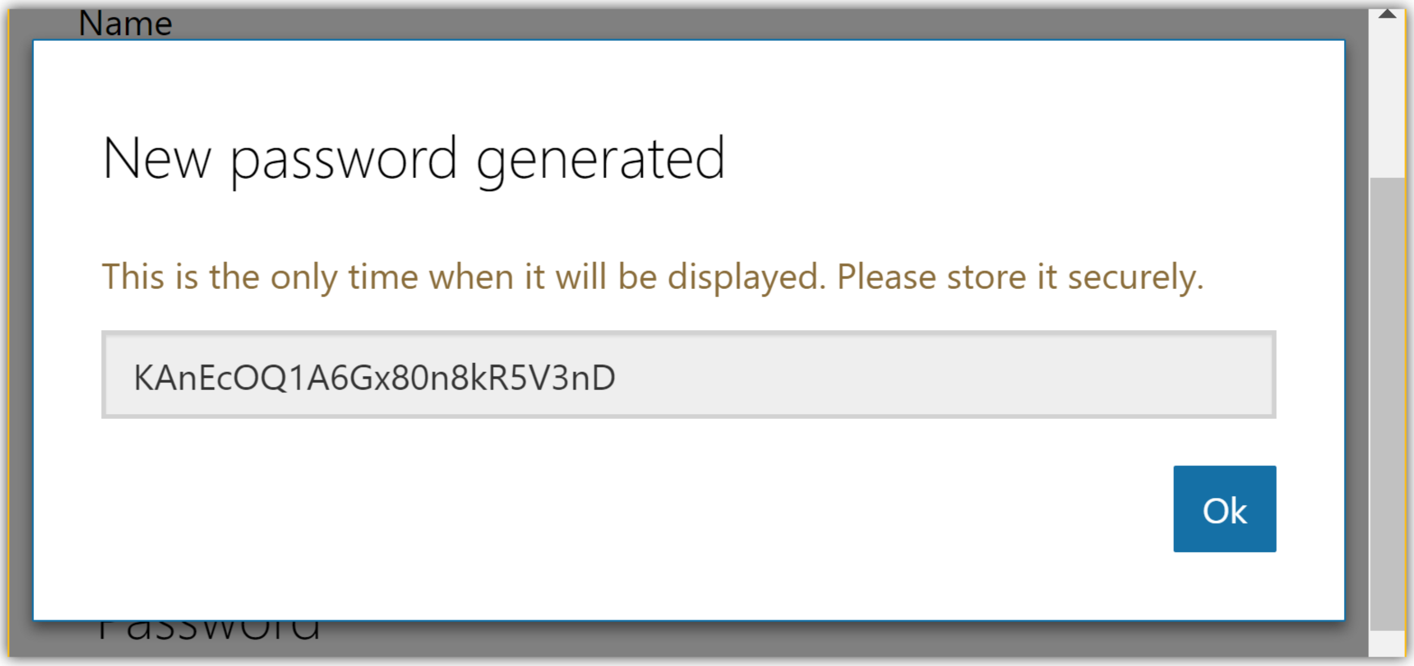
Tap **Save** to save changes.

Storing Microsoft ApplicationId and Secret

Link sensitive settings like Microsoft `ApplicationId` and `Secret` to your application configuration by using the Secret Manager tool instead of storing them in your configuration file directly, as described in the social login overview page.

Note the `Application Id` displayed on the **Registration** page.

Tap **Generate New Password** in the **Application Secrets** section. This displays a box where you can copy the application secret:



Execute the following commands in your project working directory to store the Microsoft secrets:

```
dotnet user-secrets set Authentication:Microsoft:ClientId <client-id>
dotnet user-secrets set Authentication:Microsoft:ClientSecret <client-secret>
```

The following code reads the configuration values stored by the Secret Manager:

```
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);

    if (env.IsDevelopment())
    {
        // For more details on using the user secret store see http://go.microsoft.com/fwlink/?LinkID=532709
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

Enable Microsoft Account middleware

Note

Use NuGet to install the Microsoft.AspNetCore.Authentication.MicrosoftAccount package if it hasn't already been installed. Alternatively, execute the following commands in your project directory:

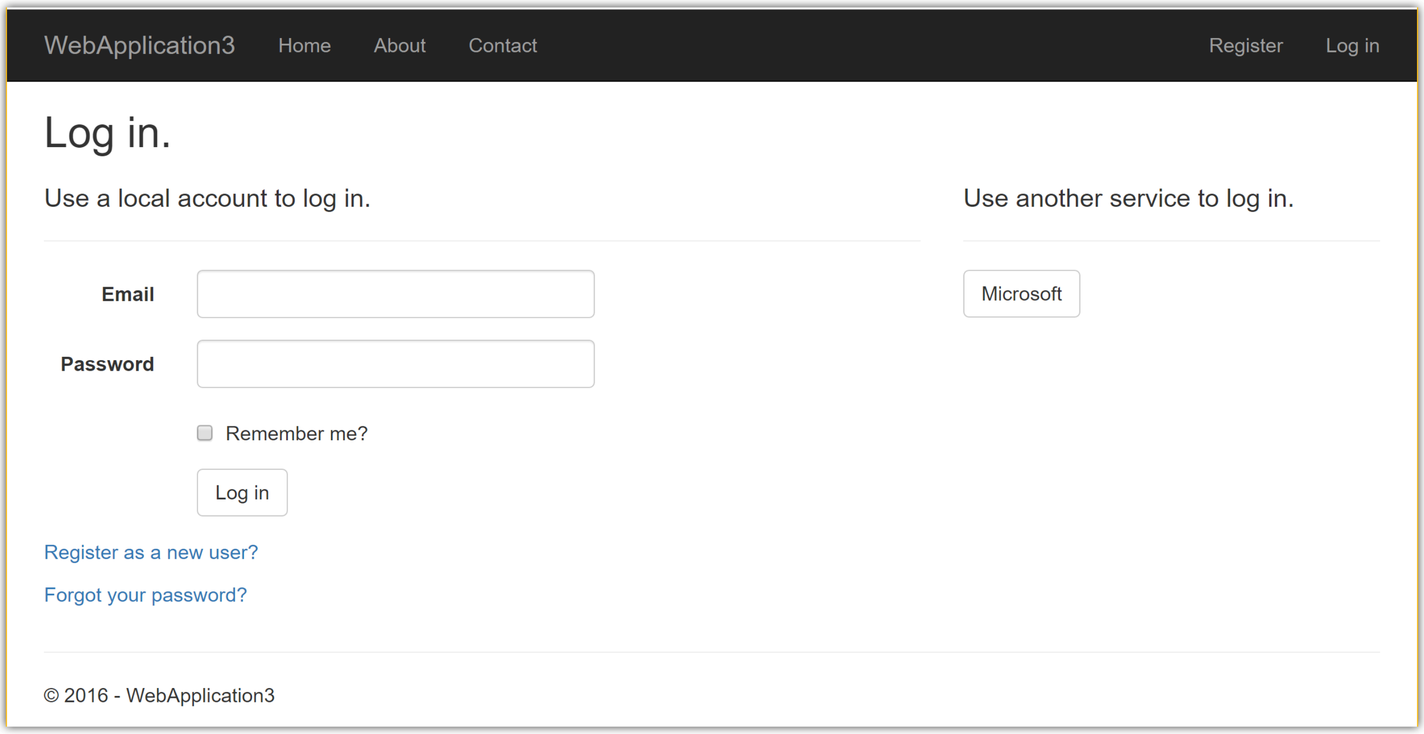
`dotnet add package Microsoft.AspNetCore.Authentication.MicrosoftAccount`

Add the Microsoft Account middleware in the `Configure` method in `Startup.cs`:

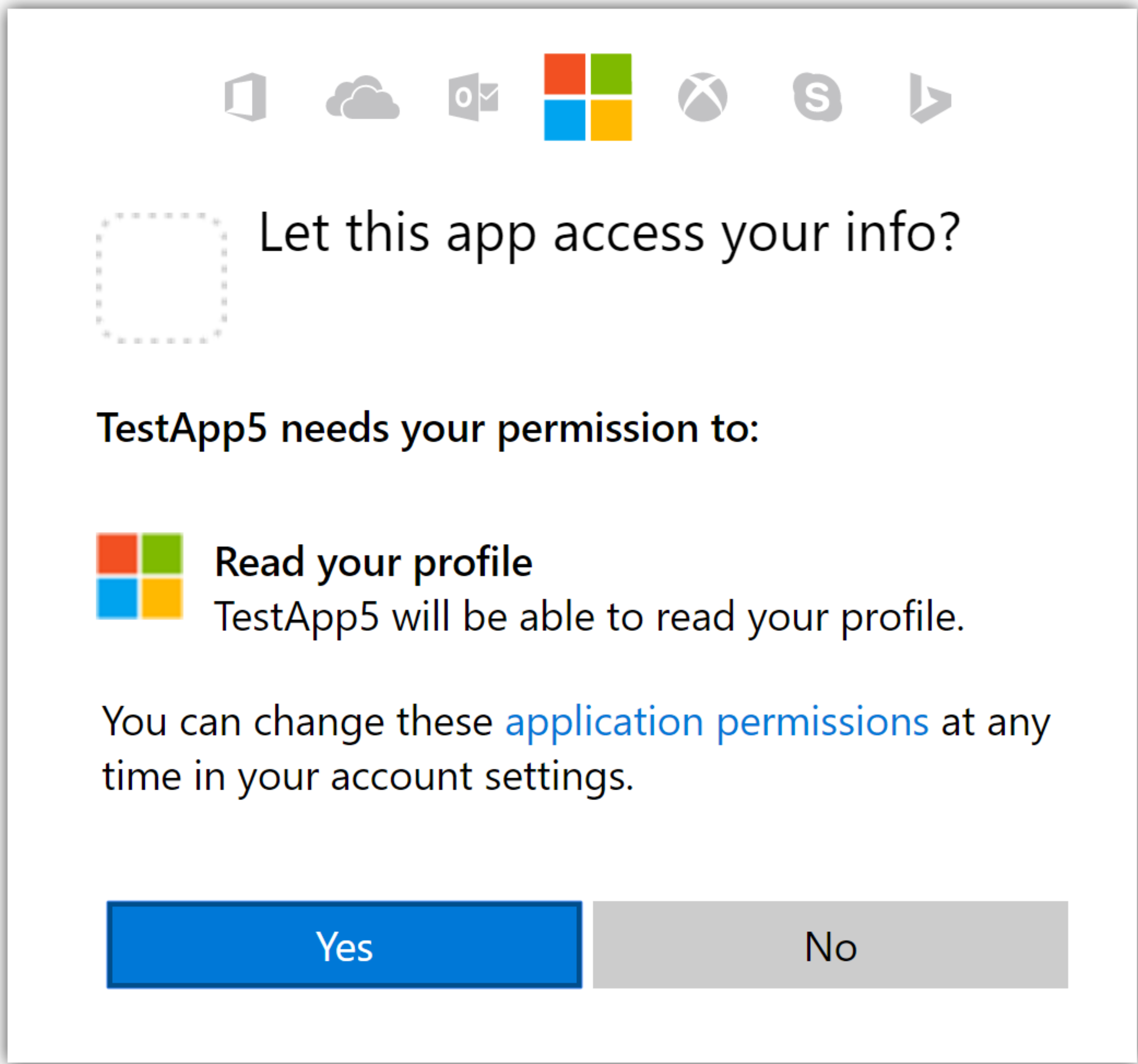
```
app.UseMicrosoftAccountAuthentication(new MicrosoftAccountOptions()
{
    ClientId = Configuration["Authentication:Microsoft:ClientId"],
    ClientSecret = Configuration["Authentication:Microsoft:ClientSecret"]
});
```

Sign in with Microsoft Account

Run your application and click **Log in**. An option to sign in with Microsoft appears:

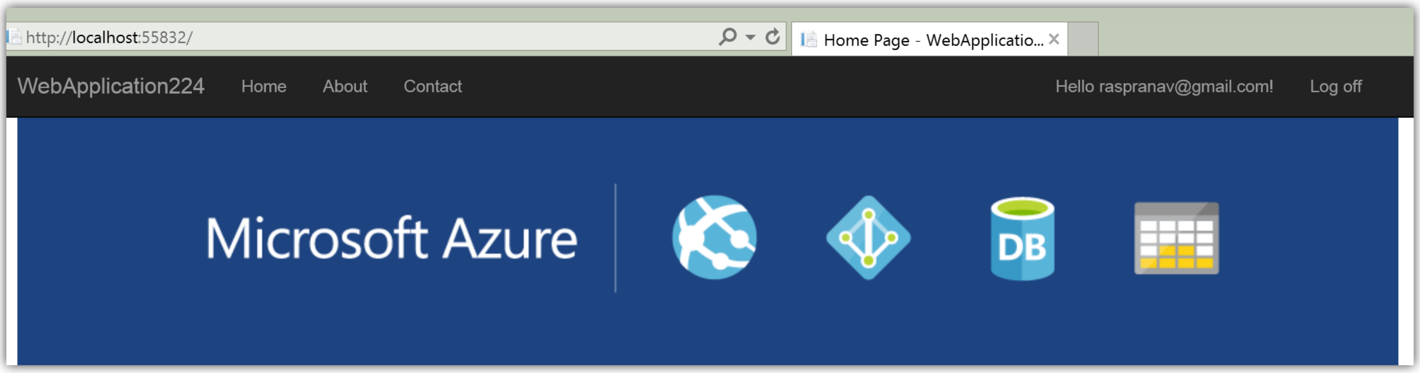


When you click on Microsoft, you are redirected to Microsoft for authentication:



After entering your Microsoft Account credentials, you are redirected back to the web site where you can set your email.

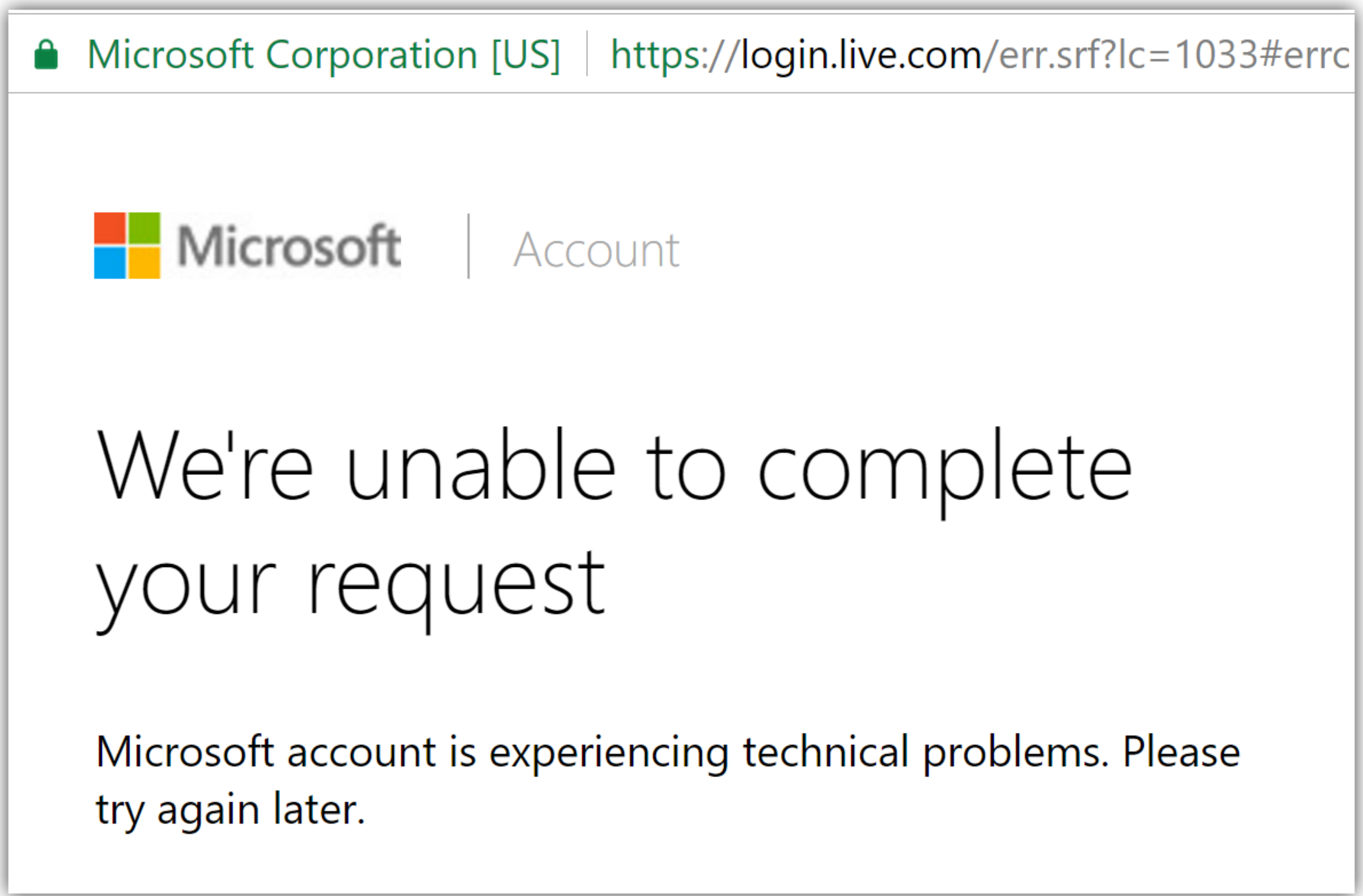
You are now logged in using your Microsoft credentials:



Note

If the Microsoft Account provider redirects you to a sign in error page, note the error title and description directly following the `#`

(hashtag) in the Uri. The most common cause is your application Uri not matching any of the **Redirect URIs** specified for the **Web** platform. In this case, ensure protocol, host, and port are all correct. Your application should be using `https` protocol and the redirect uri should end with **signin-microsoft** as that's the route Microsoft Account middleware requests the login provider to redirect to.



Next steps

This article showed how you can authenticate with Microsoft. You can follow a similar approach to authenticate with other providers listed on the previous page.

Once you publish your web site to Azure web app, you should reset the `Secret` in the Microsoft developer portal.

Set the `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Short survey of other authentication providers

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

Here are set up instructions for some other common OAuth providers. Third-party NuGet packages such as the ones maintained by aspnet-contrib can be used to complement authentication providers implemented by the ASP.NET Core team.

Set up **LinkedIn** sign in: https://developer.linkedin.com/my-apps. See official steps.

Set up **Instagram** sign in: https://www.instagram.com/developer/clients/manage. See official steps.

Set up **Reddit** sign in: https://www.reddit.com/prefs/apps. See official steps.

Set up **Github** sign in: https://github.com/settings/applications/new. See official steps.

Set up **Yahoo** sign in: https://developer.yahoo.com/apps/create/. See official steps.

Set up **Tumblr** sign in: https://www.tumblr.com/oauth/apps. See official steps.

Set up **Pinterest** sign in: https://developers.pinterest.com/apps. See official steps.

Set up **Pocket** sign in: http://getpocket.com/developer/apps/new. See official steps.

Set up **Flickr** sign in: https://www.flickr.com/services/apps/create. See official steps.

Set up **Dribble** sign in: https://dribbble.com/signup. See official steps.

Set up **Vimeo** sign in: https://developer.vimeo.com/apps. See official steps.

Set up **SoundCloud** sign in: http://soundcloud.com/you/apps/new. See official steps.

Set up **VK** sign in: https://vk.com/apps?act=manage. See official steps.

Account confirmation and password recovery

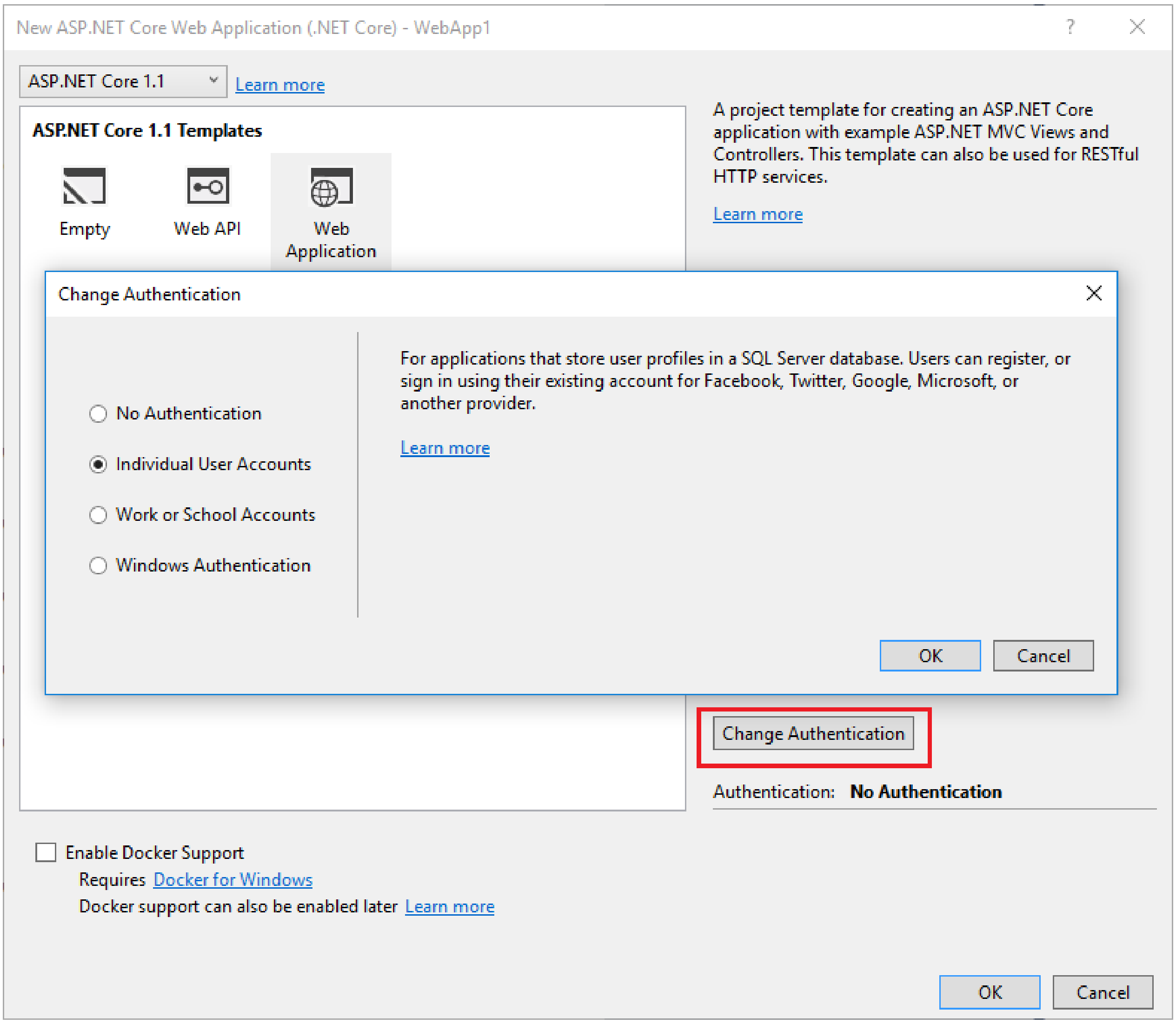
By Rick Anderson

This tutorial shows you how to build an ASP.NET Core app with email confirmation and password reset.

Create a New ASP.NET Core Project

The tutorial requires Visual Studio 2017 or higher.

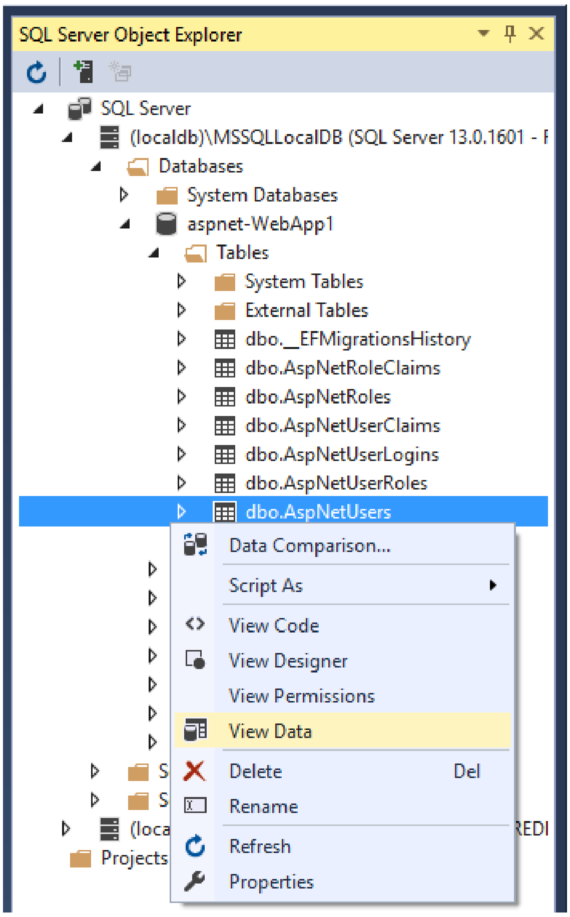
In Visual Studio, create a New Web Application Project.
Select **Change Authentication** and set to **Individual User Accounts**.



Run the app, select the **Register** link, and register a user. Follow the instructions to run Entity Framework migrations. At this point, the only validation on the email is with the [EmailAddress] attribute. After you submit the registration, you are logged into the app. Later in the tutorial we'll change this so new users cannot log in until their email has been validated.

View the Identity database

From the **View** menu, select **SQL Server Object Explorer** (SSOX).
Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right click on **dbo.AspNetUsers** > **View Data**:



Note the `EmailConfirmed` field is `False`.

You might want to use this email again in the next step when the app sends a confirmation email. Right-click on the row and select **Delete**. Deleting the email alias now will make it easier in the following steps.

Require SSL and setup IIS Express for SSL

See Enforcing SSL.

Require email confirmation

It's a best practice to confirm the email of a new user registration to verify they are not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, you would want to prevent "yli@example.com" from registering as "nolivetto@contoso.com." Without email confirmation, "nolivetto@contoso.com" could get unwanted email from your app. Suppose the user accidentally registered as "ylo@example.com" and hadn't noticed the misspelling of "yli," they wouldn't be able to use password recovery because the app doesn't have their correct email. Email confirmation provides only limited protection from bots and doesn't provide protection from determined spammers who have many working email aliases they can use to register.

You generally want to prevent new users from posting any data to your web site before they have a confirmed email. In the sections below, we will enable email confirmation and modify the code to prevent newly registered users from logging in until their email has been confirmed.

Update `ConfigureServices` to require a confirmed email:

```
// Requires using Microsoft.AspNetCore.Mvc;
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });

    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
        {
            config.SignIn.RequireConfirmedEmail = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();

    services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

Configure email provider

In this tutorial we'll be using SendGrid to send email. You'll need a SendGrid account and key to send email. We'll use the Options pattern to access the user account and key settings. For more information, see configuration.

Create a class to fetch the secure email key. For this sample, the `AuthMessageSenderOptions` class is created in the *Services/AuthMessageSenderOptions.cs* file.

```
public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}
```

Set the `SendGridUser` and `SendGridKey` with the secret-manager tool. For example:

```
C:\WebAppl\src\WebApp1>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

On Windows, Secret Manager stores your keys/value pairs in a *secrets.json* file in the %APPDATA%/Microsoft/UserSecrets/ directory.

The contents of the *secrets.json* file are not encrypted. The *secrets.json* file is shown below (the `SendGridKey` value has been removed.)

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "<key removed>"
}
```

Configure startup to use `AuthMessageSenderOptions`

Add `AuthMessageSenderOptions` to the service container at the end of the `ConfigureServices` method in the *Startup.cs* file:

```
// Requires using Microsoft.AspNetCore.Mvc;
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });

    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>(config =>
        {
            config.SignIn.RequireConfirmedEmail = true;
        })
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();

    services.Configure<AuthMessageSenderOptions>(Configuration);
}
```

Configure the AuthMessageSender class

This tutorial shows how to add email notification through SendGrid, but you can send email using SMTP and other mechanisms.

Install the `SendGrid` NuGet package. From the Package Manager Console, enter the following the following command:

```
Install-Package SendGrid
```

See Get Started with SendGrid for Free to register for a free SendGrid account.

Add code in *Services/MessageServices.cs* similar to the following to configure SendGrid:

```
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebApp1.Services
{
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions 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 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);
        }

        public Task SendSmsAsync(string number, string message)
        {
            // Plug in your SMS service here to send a text message.
            return Task.FromResult(0);
        }
    }
}
```

Enable account confirmation and password recovery

The template already has the code for account confirmation and password recovery. Follow these steps to enable it:

Find the `[HttpPost] Register` method in the *AccountController.cs* file.
Uncomment the code to enable account confirmation.

```
//
// POST: /Account/Register
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // Send an email with this link
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Action(nameof(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>");

            // Comment out following line to prevent a new user automatically logged on.
            // await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation(3, "User created a new account with password.");
            return RedirectToLocal(returnUrl);
        }
        AddErrors(result);
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Note: We're also preventing a newly registered user from being automatically logged on by commenting out the following line:

```
//await _signInManager.SignInAsync(user, isPersistent: false);
```

Enable password recovery by uncommenting the code in the `ForgotPassword` action in the *Controllers/AccountController.cs* file.

```
//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await _userManager.FindByEmailAsync(model.Email);
        if (user == null || !(await _userManager.IsEmailConfirmedAsync(user)))
        {
            // Don't reveal that the user does not exist or is not confirmed.
            return View("ForgotPasswordConfirmation");
        }

        // Send an email with this link
        var code = await _userManager.GeneratePasswordResetTokenAsync(user);
        var callbackUrl = Url.Action(nameof(ResetPassword), "Account",
            new { userId = user.Id, code = code }, protocol: HttpContext.Request.Scheme);
        await _emailSender.SendEmailAsync(model.Email, "Reset Password",
           $"Please reset your password by clicking here: <a href='{callbackUrl}'>link</a>");
        return View("ForgotPasswordConfirmation");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

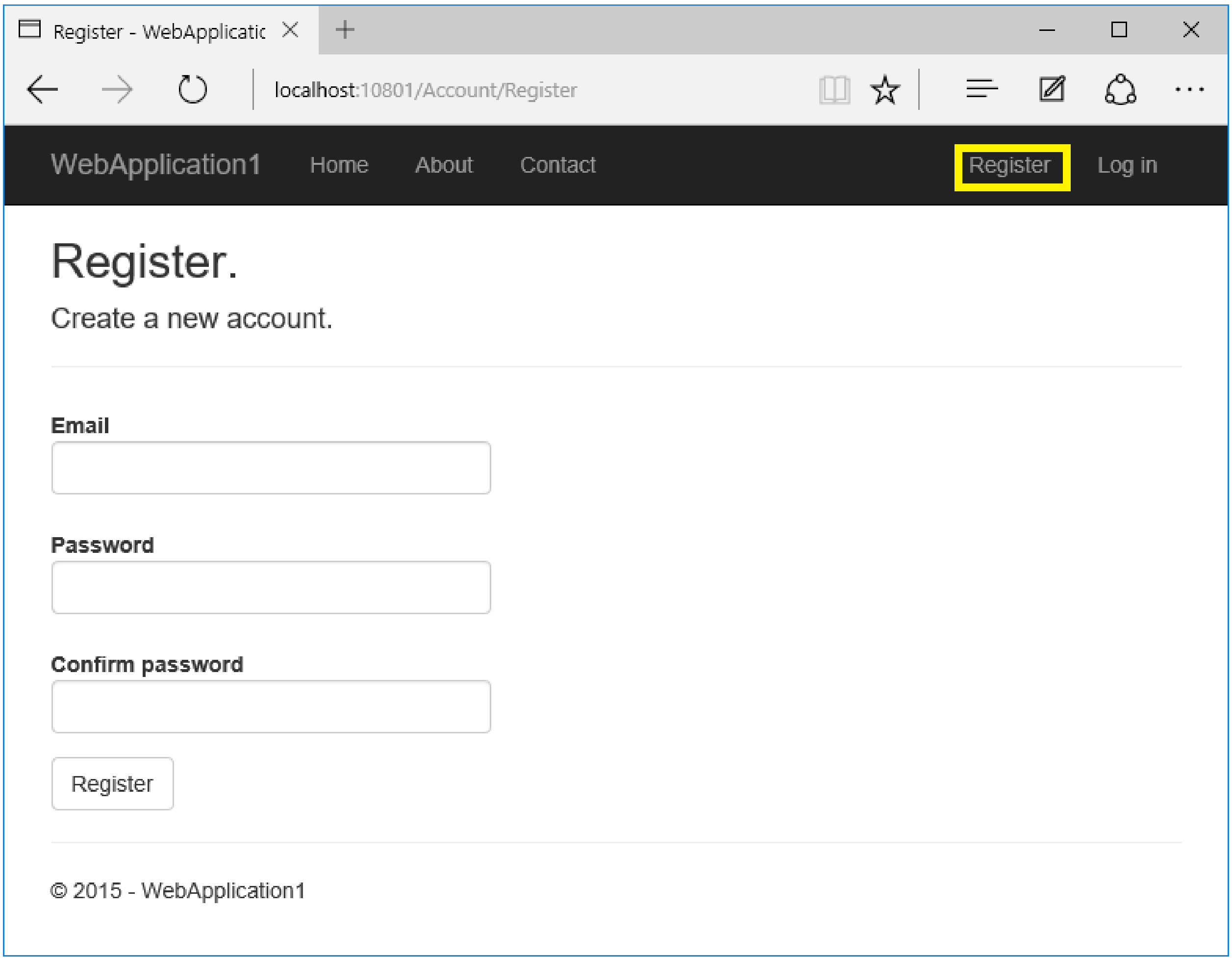
Uncomment the markup in *Views/Account/ForgotPassword.cshtml*:

```
@model ForgotPasswordViewModel
@{
    ViewData["Title"] = "Forgot your password?";
}

<h2>@ViewData["Title"]</h2>
<p>
    For more information on how to enable reset password please see this
    <a href="https://go.microsoft.com/fwlink/?LinkID=532713">article</a>.
</p>

<form asp-controller="Account" asp-action="ForgotPassword" method="post" class="form-horizontal">
    <h4>Enter your email.</h4>
    <hr />
    <div asp-validation-summary="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">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Submit</button>
        </div>
    </div>
</form>

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

Register, confirm email, and reset password

In this section, run the web app and show the account confirmation and password recovery flow.

Run the app and register a new user



Check your email for the account confirmation link. If you don't get the email notification:

Check the SendGrid web site to verify your sent mail messages.
Check your spam folder.
Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
In SSOX, navigate to **dbo.AspNetUsers** and delete the email entry and try again.

Click the link to confirm your email.

Log in with your email and password.
Log off.

Test password reset

If you're logged in, select **Logout**.
Select the **Log in** link and select the **Forgot your password?** link.
Enter the email you used to register the account.
An email with a link to reset your password will be sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can login with your email and new password.

Prevent login at registration

With the current templates, once a user completes the registration form, they are logged in (authenticated). You generally want to confirm their email before logging them in. In the section below, we will modify the code to require new users have a confirmed email before they are logged in. Update the `[HttpPost] Login` action in the *AccountController.cs* file with the following highlighted changes.

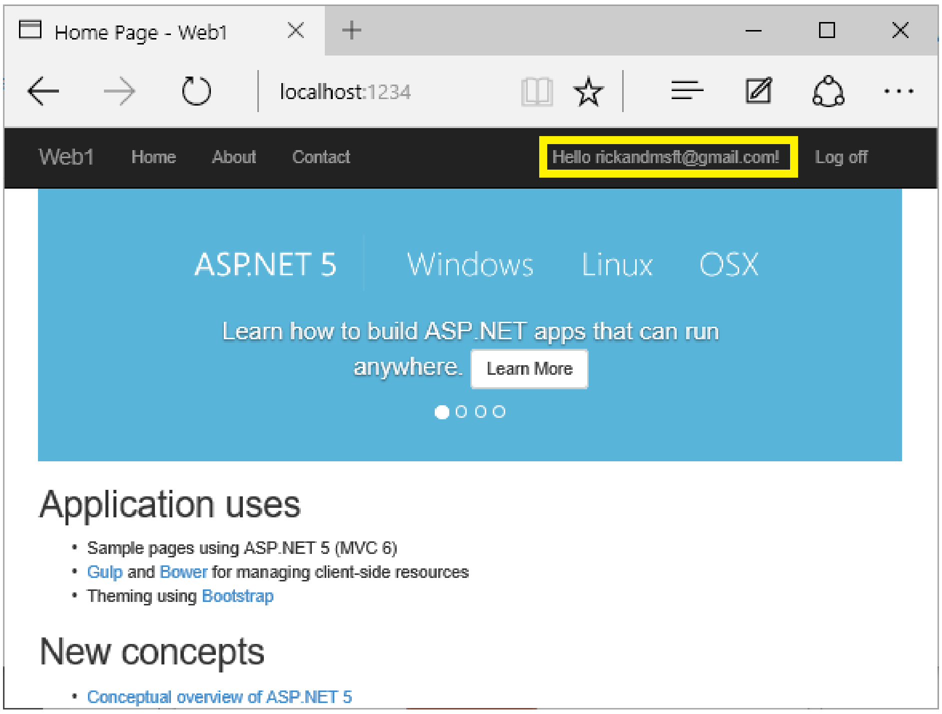
```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    if (ModelState.IsValid)
    {
        // Require the user to have a confirmed email before they can log on.
        var user = await _userManager.FindByEmailAsync(model.Email);
        if (user != null)
        {
            if (!await _userManager.IsEmailConfirmedAsync(user))
            {
                ModelState.AddModelError(string.Empty,
                              "You must have a confirmed email to log in.");
                return View(model);
            }
        }
        // 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);
}
```

Note: A security best practice is to not use production secrets in test and development. If you publish the app to Azure, you can set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is setup to read keys from environment variables.

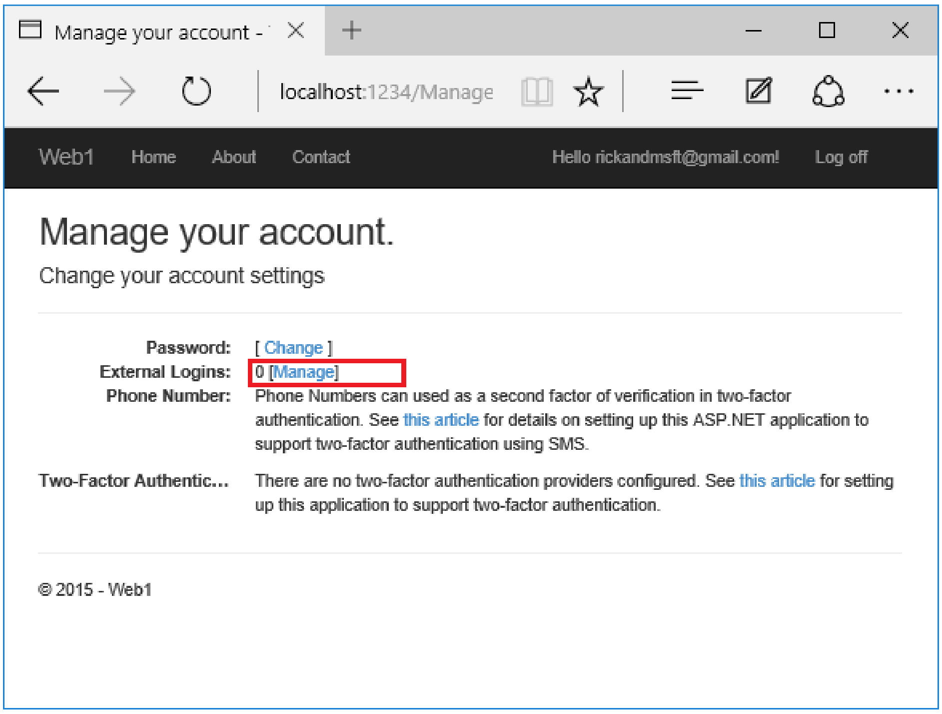
Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See Enabling authentication using Facebook, Google and other external providers.

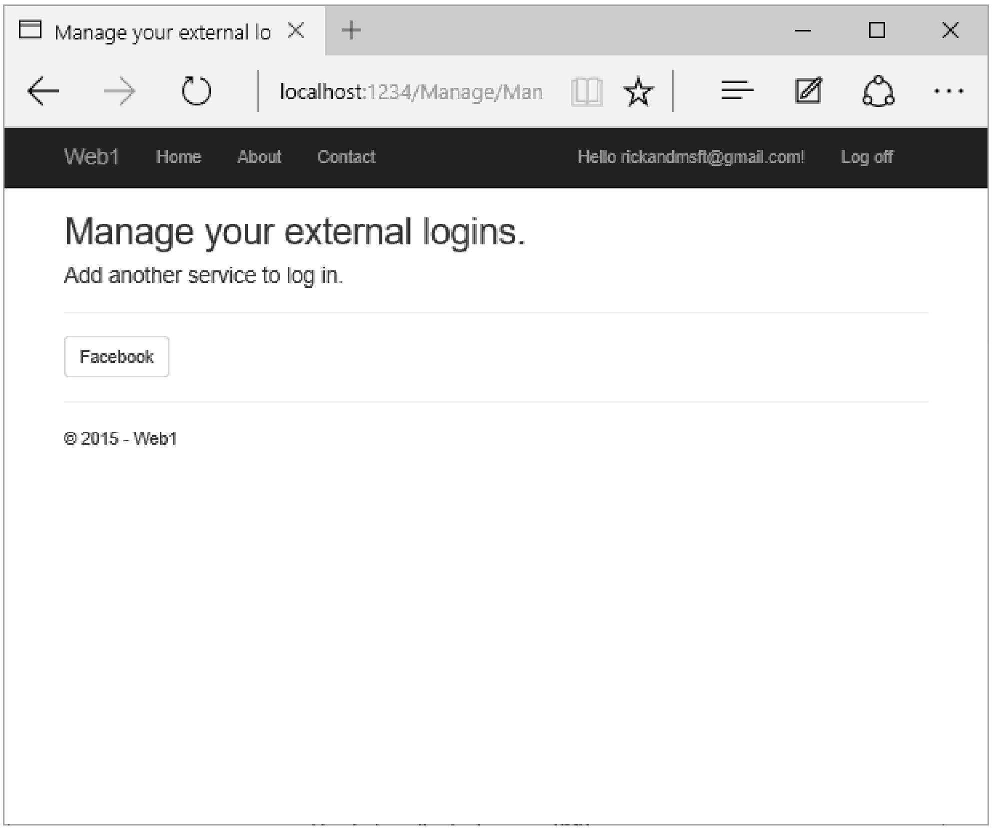
You can combine local and social accounts by clicking on your email link. In the following sequence "RickAndMSFT@gmail.com" is first created as a local login, but you can create the account as a social login first, then add a local login.



Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



Click the link to another login service and accept the app requests. In the image below, Facebook is the external authentication provider:



The two accounts have been combined. You will be able to log on with either account. You might want your users to add local accounts in case their social log in authentication service is down, or more likely they have lost access to their social account.

Two-factor authentication with SMS

By Rick Anderson and Swiss-Devs

This tutorial will show you how to set up two-factor authentication (2FA) using SMS. Instructions are given for twilio and ASPSMS, but you can use any other SMS provider. We recommend you complete Account Confirmation and Password Recovery before starting this tutorial.

View the completed sample. How to download.

Create a new ASP.NET Core project

Create a new ASP.NET Core web app named `Web2FA` with individual user accounts. Follow the instructions in Enforcing SSL in an ASP.NET Core app to set up and require SSL.

Create an SMS account

Create an SMS account, for example, from twilio or ASPSMS. Record the authentication credentials (for twilio: accountSid and authToken, for ASPSMS: Userkey and Password).

Figuring out SMS Provider credentials

Twilio:
From the Dashboard tab of your Twilio account, copy the **Account SID** and **Auth token**.

ASPSMS:
From your account settings, navigate to **Userkey** and copy it together with your **Password**.

We will later store these values in with the secret-manager tool within the keys `SMSAccountIdentification` and `SMSAccountPassword`.

Specifying SenderID / Originator

Twilio:
From the Numbers tab, copy your Twilio **phone number**.

ASPSMS:
Within the Unlock Originators Menu, unlock one or more Originators or choose an alphanumeric Originator (Not supported by all networks).

We will later store this value with the secret-manager tool within the key `SMSAccountFrom`.

Provide credentials for the SMS service

We'll use the Options pattern to access the user account and key settings.

Create a class to fetch the secure SMS key. For this sample, the `SMSoptions` class is created in the *Services/SMSoptions.cs* file.

```
namespace Web2FA.Services
{
    public class SMSoptions
    {
        public string SMSAccountIdentification { get; set; }
        public string SMSAccountPassword { get; set; }
        public string SMSAccountFrom { get; set; }
    }
}
```

Set the `SMSAccountIdentification`, `SMSAccountPassword` and `SMSAccountFrom` with the secret-manager tool. For example:

```
C:/Web2FA/src/WebApp1>dotnet user-secrets set SMSAccountIdentification 12345
info: Successfully saved SMSAccountIdentification = 12345 to the secret store.
```

Add the NuGet package for the SMS provider. From the Package Manager Console (PMC) run:

Twilio:

`Install-Package Twilio`

ASPSMS:

`Install-Package ASPSMS`

Add code in the *Services/MessageServices.cs* file to enable SMS. Use either the Twilio or the ASPSMS section:

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;
            var

            TwilioClient.Init(accountSid, authToken);

            var msg = MessageResource.Create(
              to: new PhoneNumber(number),
              from: new PhoneNumber(Options.SMSAccountFrom),
              body: message);
            return Task.FromResult(0);
        }
    }
}
```

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);
        }
    }
}
```

Configure startup to use `SMSoptions`

Add `SMSoptions` to the service container in the `ConfigureServices` method in the *Startup.cs*:

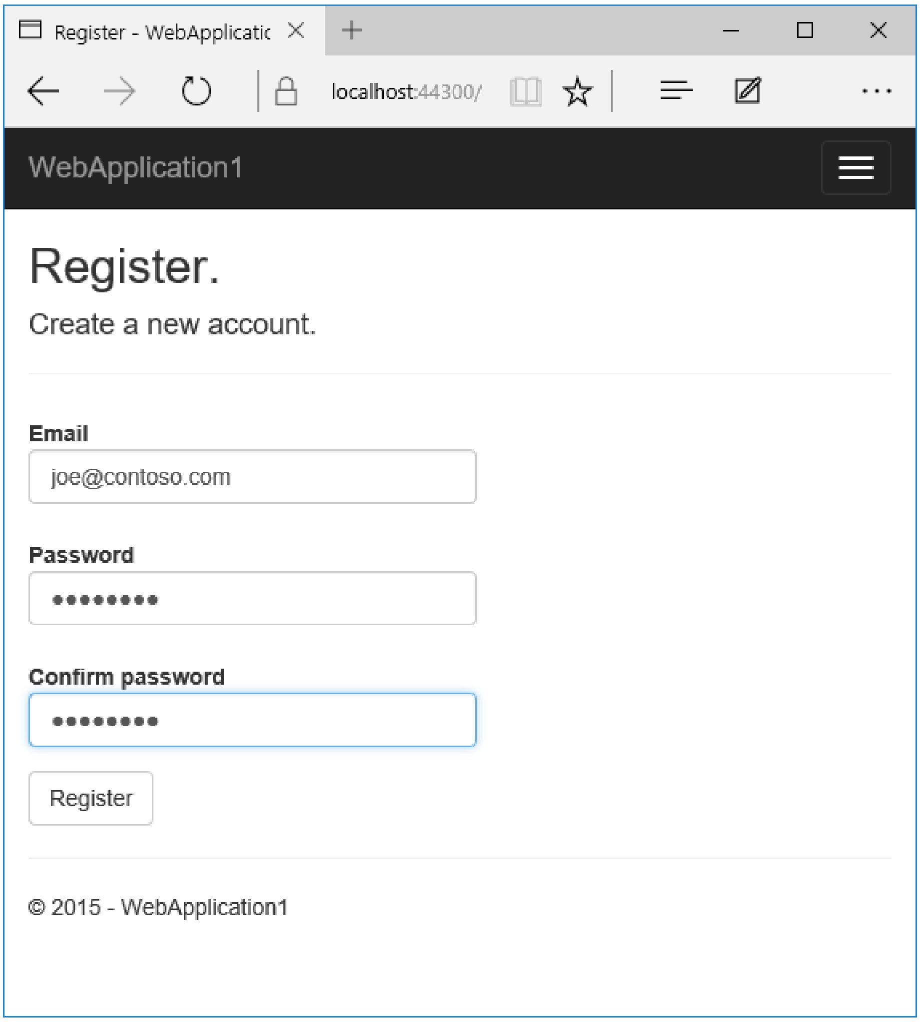
```
    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

Enable two-factor authentication

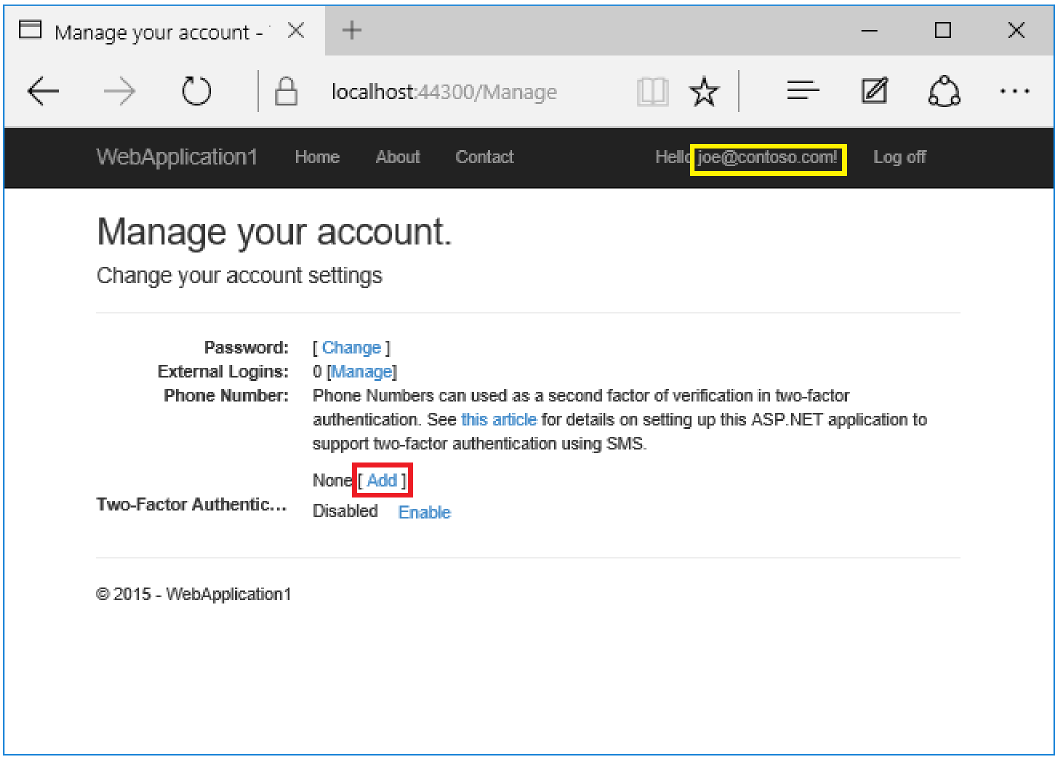
Open the *Views/Manage/Index.cshtml* Razor view file and remove the comment characters (so no markup is commnted out).

Log in with two-factor authentication

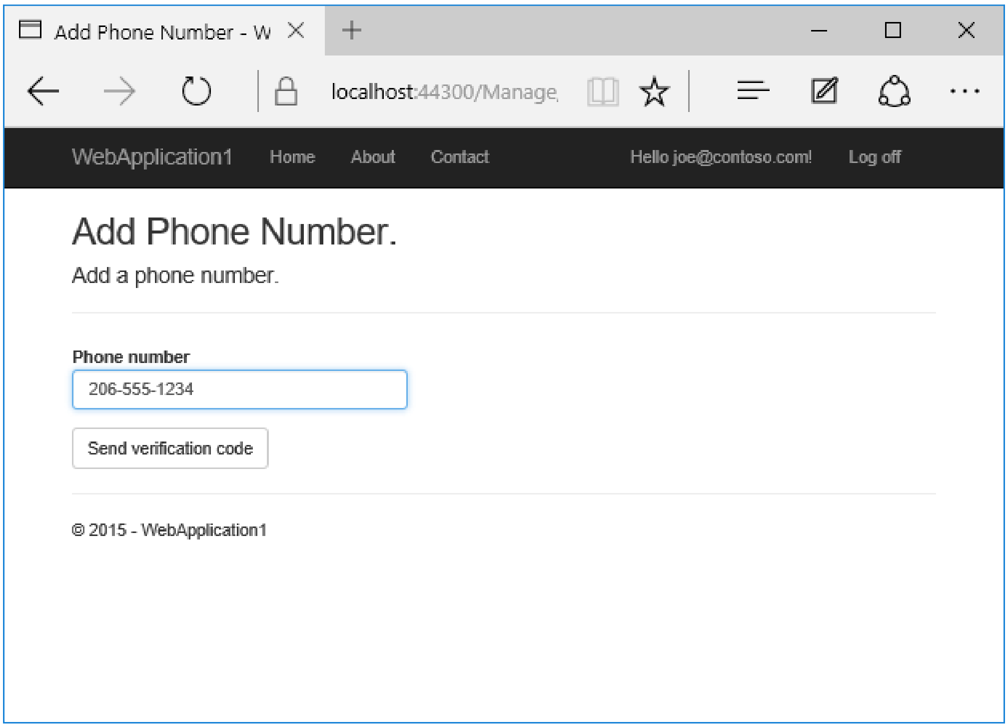
Run the app and register a new user



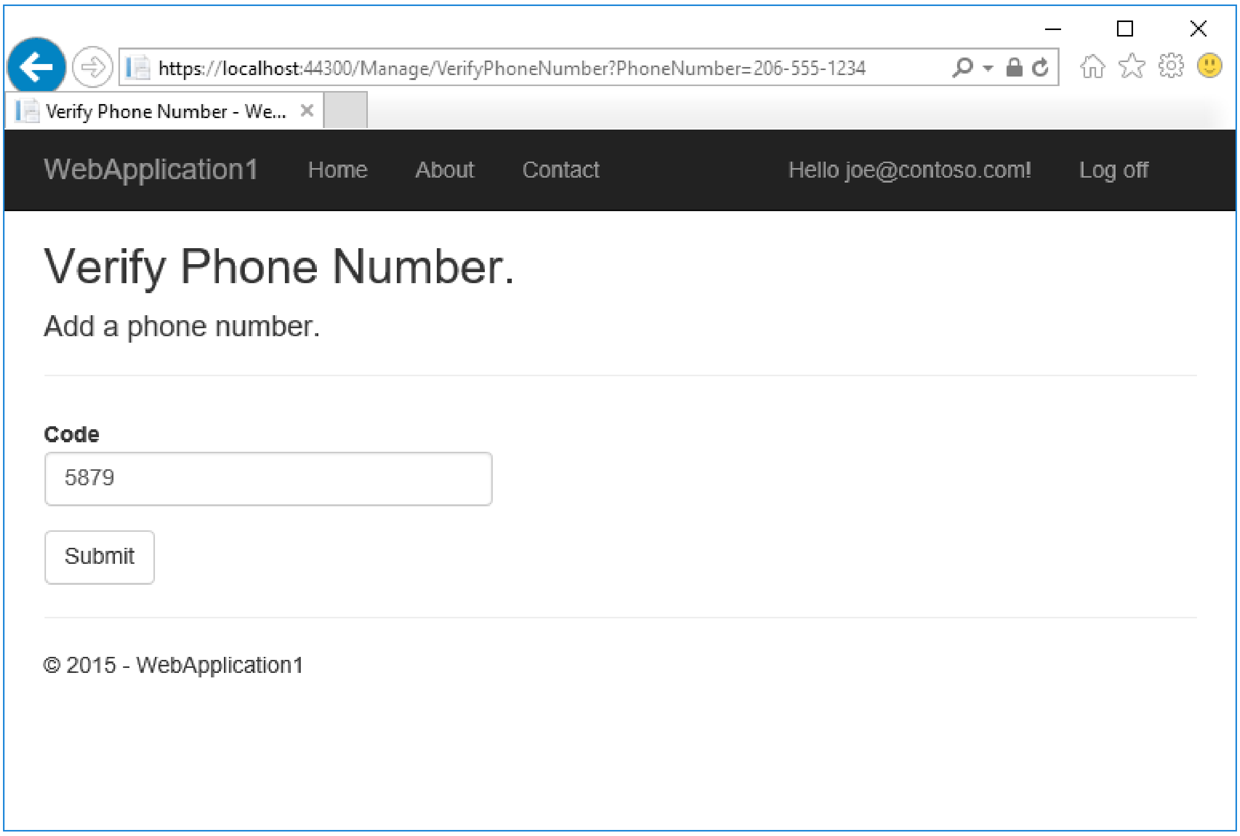
Tap on your user name, which activates the `Index` action method in Manage controller. Then tap the phone number **Add** link.



Add a phone number that will receive the verification code, and tap **Send verification code**.

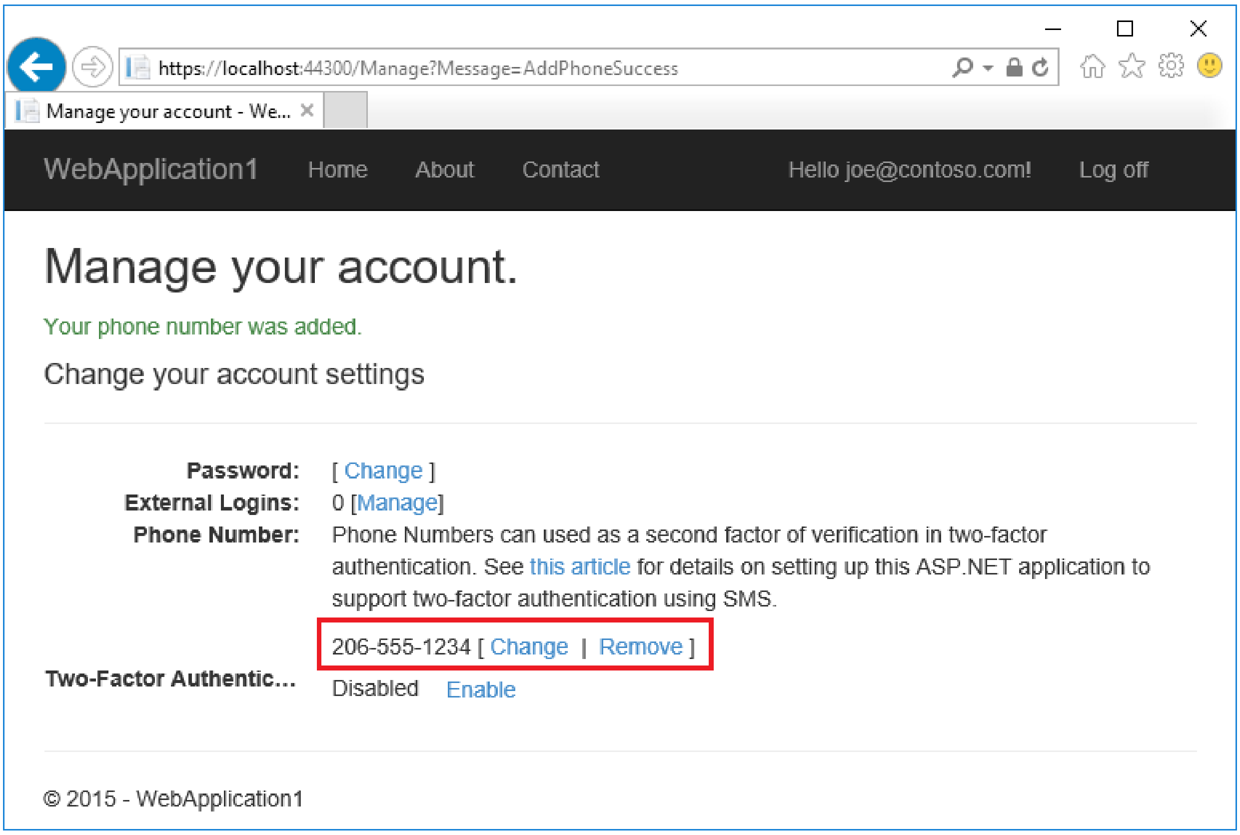


You will get a text message with the verification code. Enter it and tap **Submit**

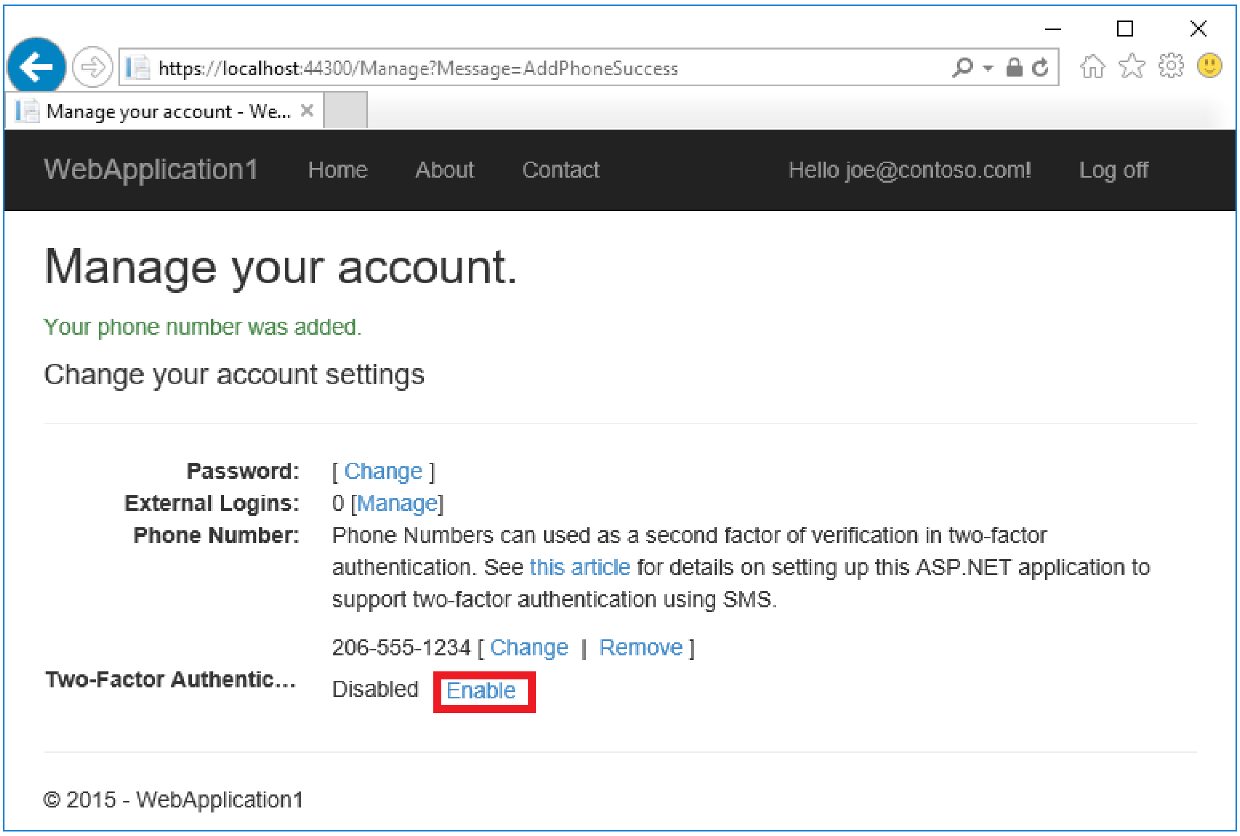


If you don't get a text message, see twilio log page.

The Manage view shows your phone number was added successfully.



Tap **Enable** to enable two-factor authentication.

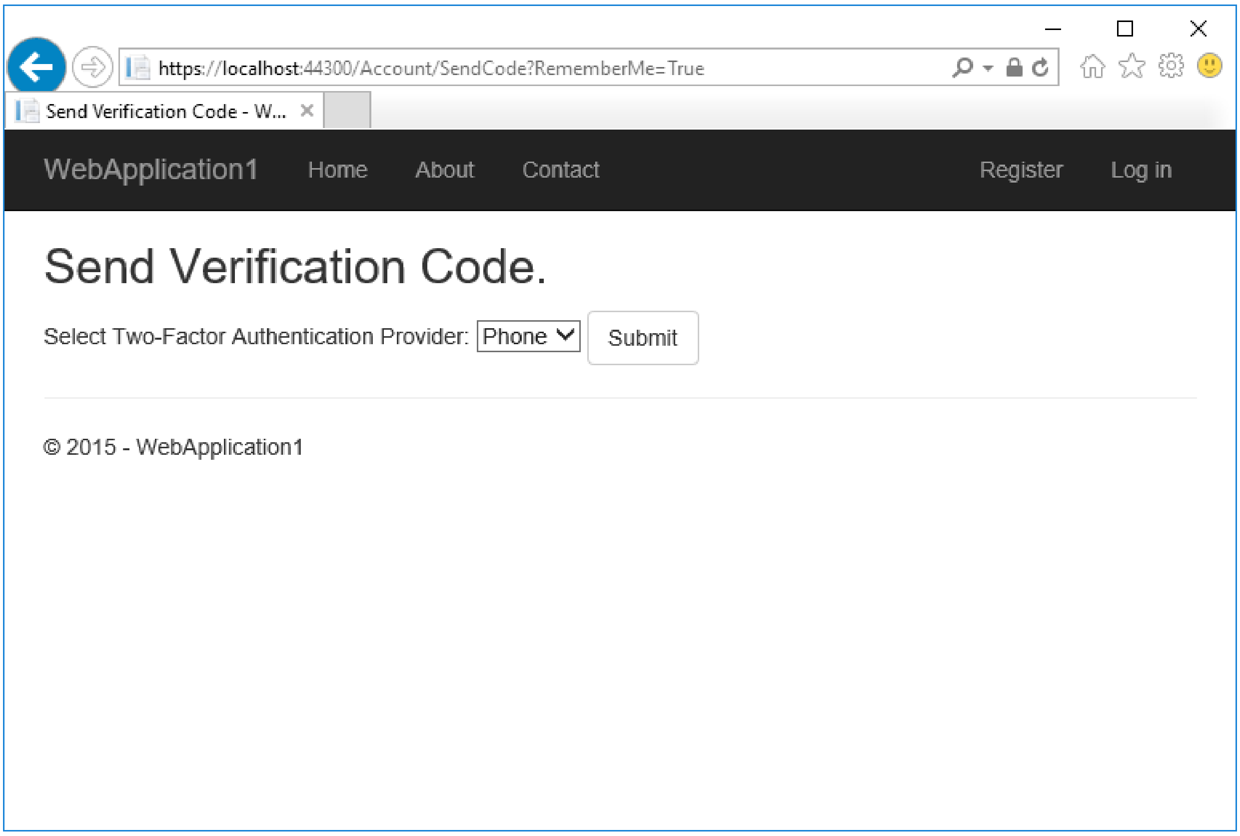


Test two-factor authentication

Log off.

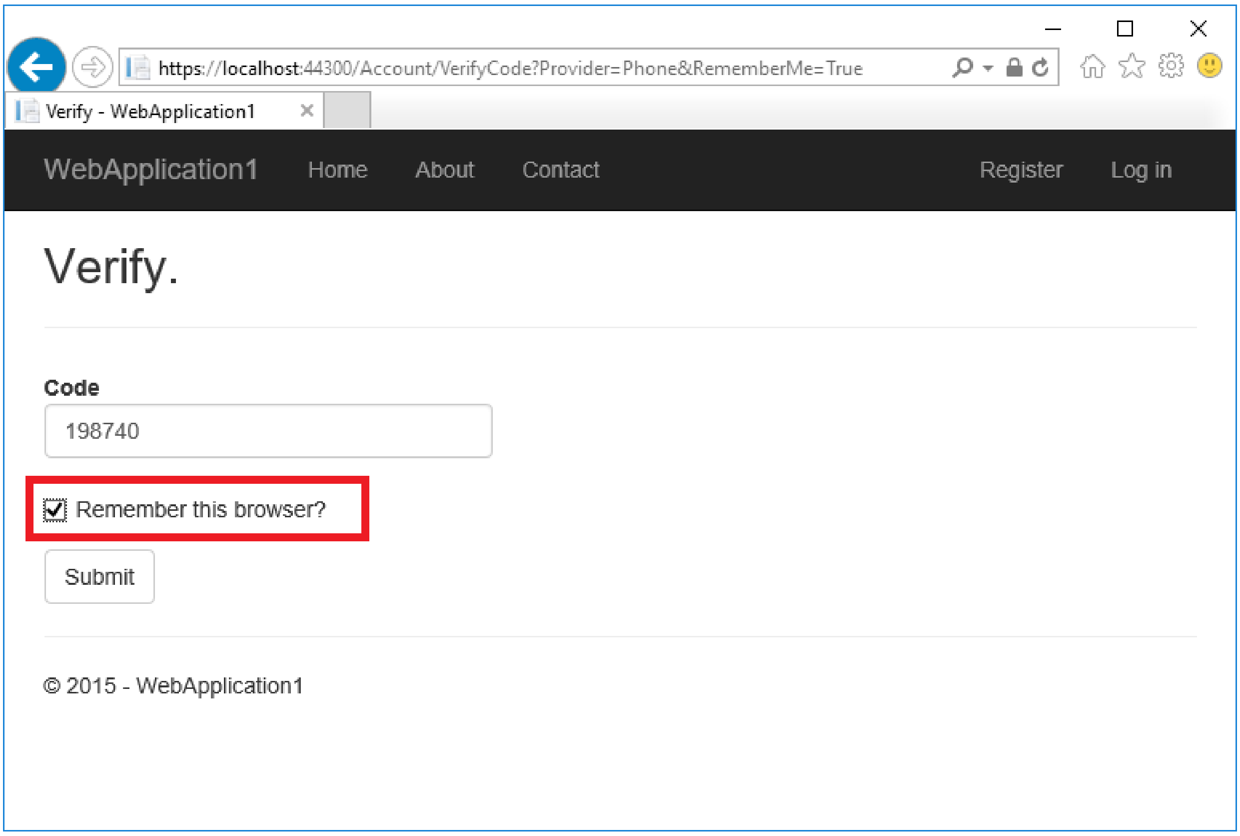
Log in.

The user account has enabled two-factor authentication, so you have to provide the second factor of authentication . In this tutorial you have enabled phone verification. The built in templates also allow you to set up email as the second factor. You can set up additional second factors for authentication such as QR codes. Tap **Submit**.



Enter the code you get in the SMS message.

Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log on when using the same device and browser. Enabling 2FA and clicking on **Remember this browser** will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your device. You can do this on any private device you regularly use. By setting **Remember this browser**, you get the added security of 2FA from devices you don't regularly use, and you get the convenience on not having to go through 2FA on your own devices.



Account lockout for protecting against brute force attacks

We recommend you use account lockout with 2FA. Once a user logs in (through a local account or social account), each failed attempt at 2FA is stored, and if the maximum attempts (default is 5) is reached, the user is locked out for five minutes (you can set the lock out time with `DefaultAccountLockoutTimeSpan`). The following configures Account to be locked out for 10 minutes after 10 failed attempts.

```
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();

    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
        options.Lockout.MaxFailedAccessAttempts = 10;
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

Using Cookie Middleware without ASP.NET Core Identity

ASP.NET Core provides cookie middleware which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal and assigns it to the `User` property on `HttpContext`. If you want to provide your own login screens and user databases you can use the cookie middleware as a standalone feature.

Adding and configuring

The first step is adding the cookie middleware to your application. First use nuget to add the `Microsoft.AspNetCore.Authentication.Cookies` package. Then add the following lines to the `Configure` method in your *Startup.cs* file before the `app.UseMvc()` statement;

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
   {
       AuthenticationScheme = "MyCookieMiddlewareInstance",
       LoginPath = new PathString("/Account/Unauthorized/"),
       AccessDeniedPath = new PathString("/Account/Forbidden/"),
       AutomaticAuthenticate = true,
       AutomaticChallenge = true
   });
```

The code snippet above configures a few options;

AuthenticationScheme - this is a value by which the middleware is known. This is useful when there are multiple instances of middleware and you want to limit authorization to one instance.

LoginPath - this is the relative path requests will be redirected to when a user attempts to access a resource but has not been authenticated.

AccessDeniedPath - this is the relative path requests will be redirected to when a user attempts to access a resource but does not pass any authorization policies for that resource.

AutomaticAuthenticate - this flag indicates that the middleware should run on every request and attempt to validate and reconstruct any serialized principal it created.

AutomaticChallenge - this flag indicates that the middleware should redirect the browser to the `LoginPath` or `AccessDeniedPath` when authorization fails.

Other options include the ability to set the issuer for any claims the middleware creates, the name of the cookie the middleware drops, the domain for the cookie and various security properties on the cookie. By default the cookie middleware will use appropriate security options for any cookies it creates, setting HTTPONLY to avoid the cookie being accessible in client side JavaScript and limiting the cookie to HTTPS if a request has come over HTTPS.

Creating an identity cookie

To create a cookie holding your user information you must construct a ClaimsPrincipal holding the information you wish to be serialized in the cookie. Once you have a suitable *ClaimsPrincipal* inside your controller method call

```
await HttpContext.Authentication.SignInAsync("MyCookieMiddlewareInstance", principal);
```

This will create an encrypted cookie and add it to the current response. The `AuthenticationScheme` specified during configuration must also be used when calling `SignInAsync`.

Under the covers the encryption used is ASP.NET's Data Protection system. If you are hosting on multiple machines, load balancing or using a web farm then you will need to configure data protection to use the same key ring and application identifier.

Signing out

To sign out the current user, and delete their cookie call the following inside your controller

```
await HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
```

Reacting to back-end changes

⚠ Warning

Once a principal cookie has been created it becomes the single source of identity - even if you disable a user in your back-end systems the cookie middleware has no knowledge of this and a user will continue to stay logged in as long as their cookie is valid.

The cookie authentication middleware provides a series of Events in its option class. The `ValidateAsync()` event can be used to intercept and override validation of the cookie identity.

Consider a back-end user database that may have a LastChanged column. In order to invalidate a cookie when the database changes you should first, when creating the cookie, add a LastChanged claim containing the current value. Then, when the database changes the LastChanged value should also be updated.

To implement an override for the `ValidateAsync()` event you must write a method with the following signature;

```
Task ValidateAsync(CookieValidatePrincipalContext context);
```

ASP.NET Core Identity implements this check as part of its `SecurityStampValidator`. A simple example would look something like as follows;

```
public static class LastChangedValidator
   {
       public static async Task ValidateAsync(CookieValidatePrincipalContext context)
       {
           // Pull database from registered DI services.
           var userRepository = context.HttpContext.RequestServices.GetRequiredService<IUserRepository>();
           var userPrincipal = context.Principal;

           // Look for the last changed claim.
           string lastChanged;
           lastChanged = (from c in userPrincipal.Claims
                          where c.Type == "LastUpdated"
                          select c.Value).FirstOrDefault();

           if (string.IsNullOrEmpty(lastChanged) ||
               !userRepository.ValidateLastChanged(userPrincipal, lastChanged))
           {
               context.RejectPrincipal();
               await context.HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
           }
       }
   }
```

This would then be wired up during cookie middleware configuration

```
app.UseCookieAuthentication(options =>
   {
       options.Events = new CookieAuthenticationEvents
       {
           // Set other options
           OnValidatePrincipal = LastChangedValidator.ValidateAsync
       };
   });
```

If you want to non-destructively update the user principal, for example, their name might have been updated, a decision which doesn't affect security in any way you can call `context.ReplacePrincipal()` and set the `context.ShouldRenew` flag to `true`.

Controlling cookie options

The `CookieAuthenticationOptions` class comes with various configuration options to enable you to fine tune the cookies created.

ClaimsIssuer - the issuer to be used for the Issuer property on any claims created by the middleware.

CookieDomain - the domain name the cookie will be served to. By default this is the host name the request was sent to. The browser will only serve the cookie to a matching host name. You may wish to adjust this to have cookies available to any host in your domain. For example setting the cookie domain to `.contoso.com` will make it available to `contoso.com`, `www.contoso.com`, `staging.www.contoso.com` etc.

CookieHttpOnly - a flag indicating if the cookie should only be accessible to servers. This defaults to `true`. Changing this value may open your application to cookie theft should your application have a Cross Site Scripting bug.

CookiePath - this can be used to isolate applications running on the same host name. If you have an app running in `/app1` and want to limit the cookies issued to just be sent to that application then you should set the `CookiePath` property to `/app1`. The cookie will now only be available to requests to `/app1` or anything underneath it.

CookieSecure - a flag indicating if the cookie created should be limited to HTTPS, HTTP or HTTPS, or the same protocol as the request. This defaults to `SameAsRequest`.

ExpireTimeSpan - the `TimeSpan` after which the cookie will expire. This is added to the current date and time to create the expiry date for the cookie.

SlidingExpiration - a flag indicating if the cookie expiration date will be reset when more than half of the `ExpireTimeSpan` interval has passed. The new expiry date will be moved forward to be the current date plus the `ExpireTimespan`. An absolute expiry time can be set by using the `AuthenticationProperties` class when calling `SignInAsync`. An absolute expiry can improve the security of your application by limiting the amount of time for which the authentication cookie is valid.

Persistent cookies and absolute expiry times

You may want to make the cookie expire be remembered over browser sessions. You may also want an absolute expiry to the identity and the cookie transporting it. You can do these things by using the `AuthenticationProperties` parameter on the `HttpContext.Authentication.SignInAsync` method called when signing in an identity and creating the cookie. The `AuthenticationProperties` class is in the `Microsoft.AspNetCore.Http.Authentication` namespace.

For example;

```
await HttpContext.Authentication.SignInAsync(
       "MyCookieMiddlewareInstance",
       principal,
       new AuthenticationProperties
       {
           IsPersistent = true
       });
```

This code snippet will create an identity and corresponding cookie which will survive through browser closures. Any sliding expiration settings previously configured via cookie options will still be honored, if the cookie expires whilst the browser is closed the browser will clear it once it is restarted.

```
await HttpContext.Authentication.SignInAsync(
       "MyCookieMiddlewareInstance",
       principal,
       new AuthenticationProperties
       {
           ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
       });
```

This code snippet will create an identity and corresponding cookie which will last for 20 minutes. This ignores any sliding expiration settings previously configured via cookie options.

The ExpiresUtc and IsPersistent properties are mutually exclusive.

Introduction

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is orthogonal and independent from authentication, which is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

Authorization Types

ASP.NET Core authorization provides a simple declarative role and a rich policy based model. Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements. Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes are found in the `Microsoft.AspNetCore.Authorization` namespace.

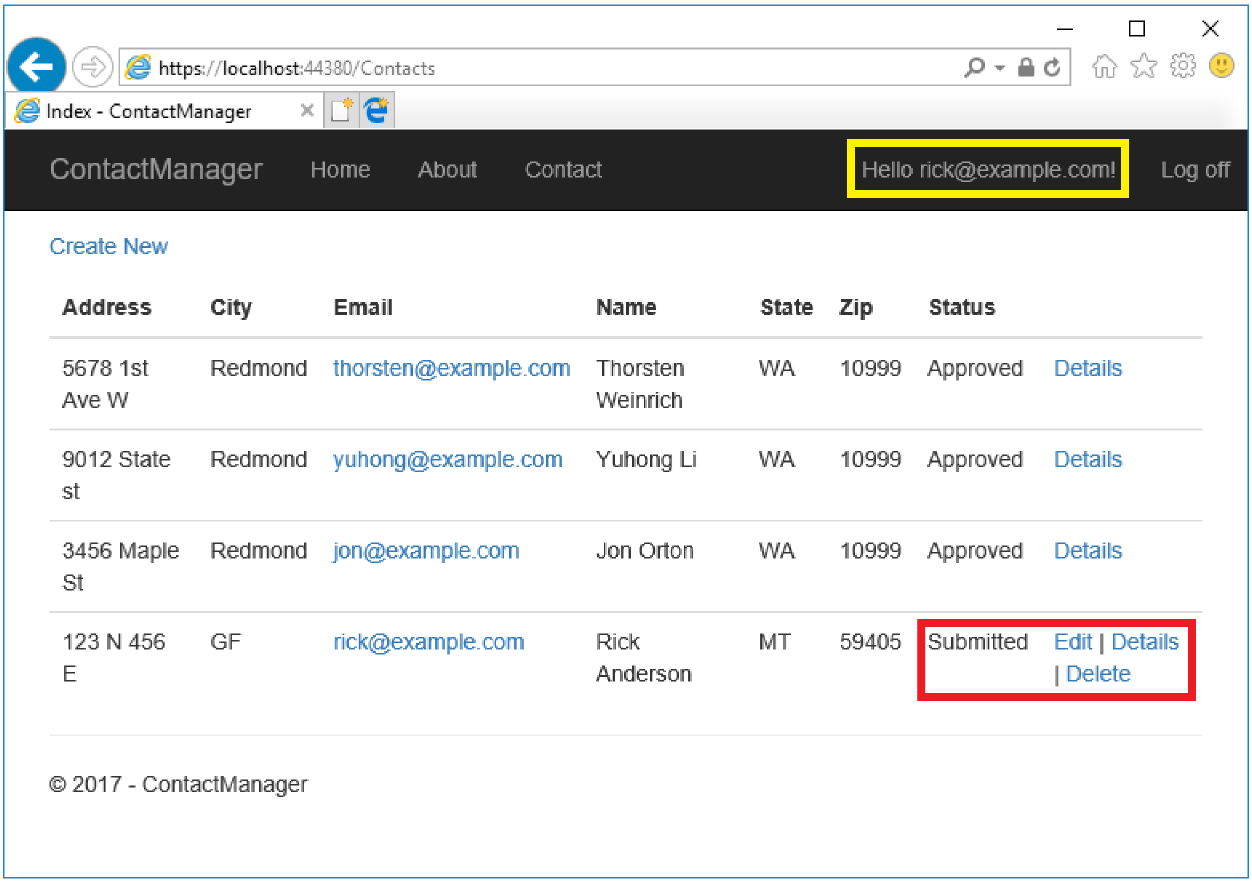
Create an ASP.NET Core app with user data protected by authorization

By Rick Anderson and Joe Audette

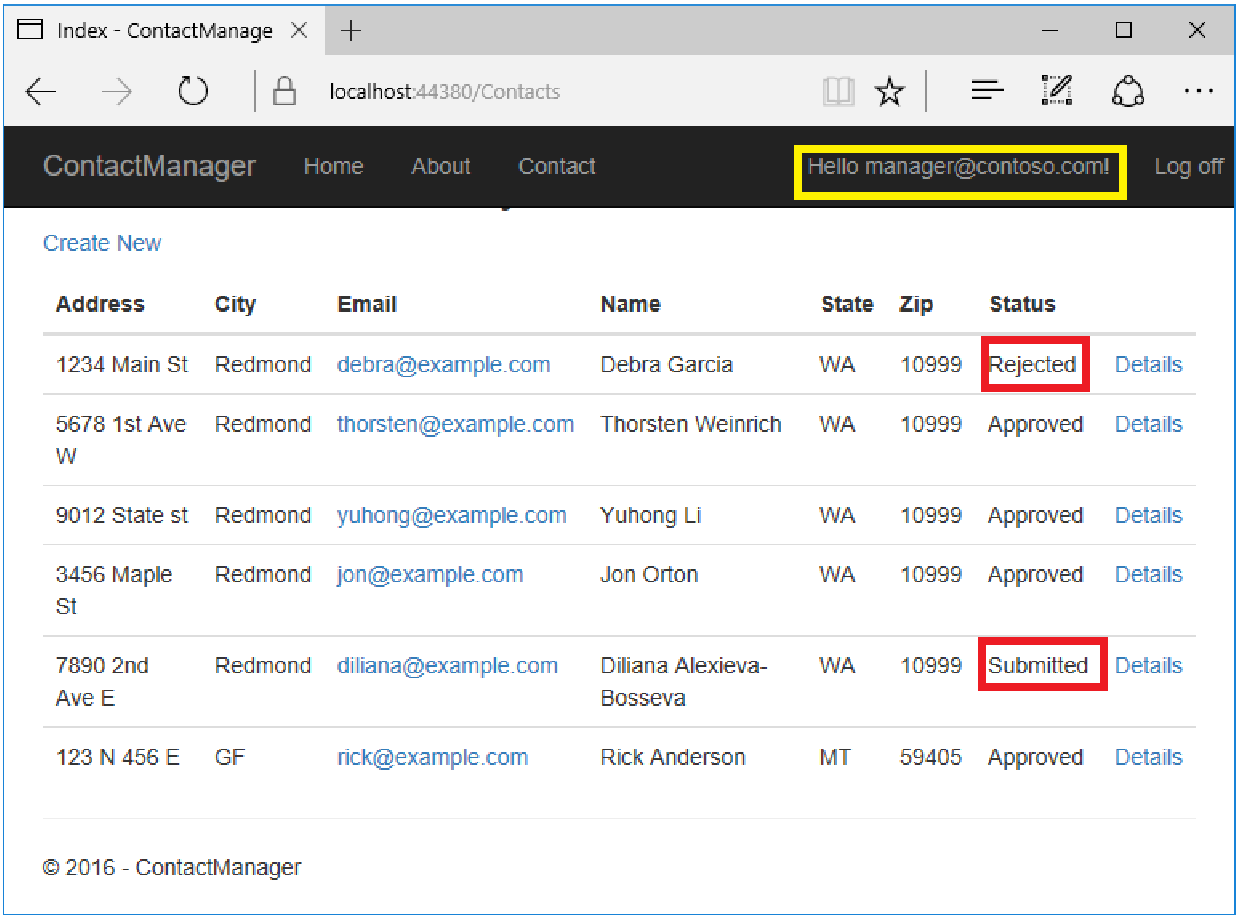
This tutorial shows how to create a web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

Registered users can view all the approved contact data.
Registered users can edit/delete their own data.
Managers can approve or reject contact data. Only approved contacts are visible to users.
Administrators can approve/reject and edit/delete any data.

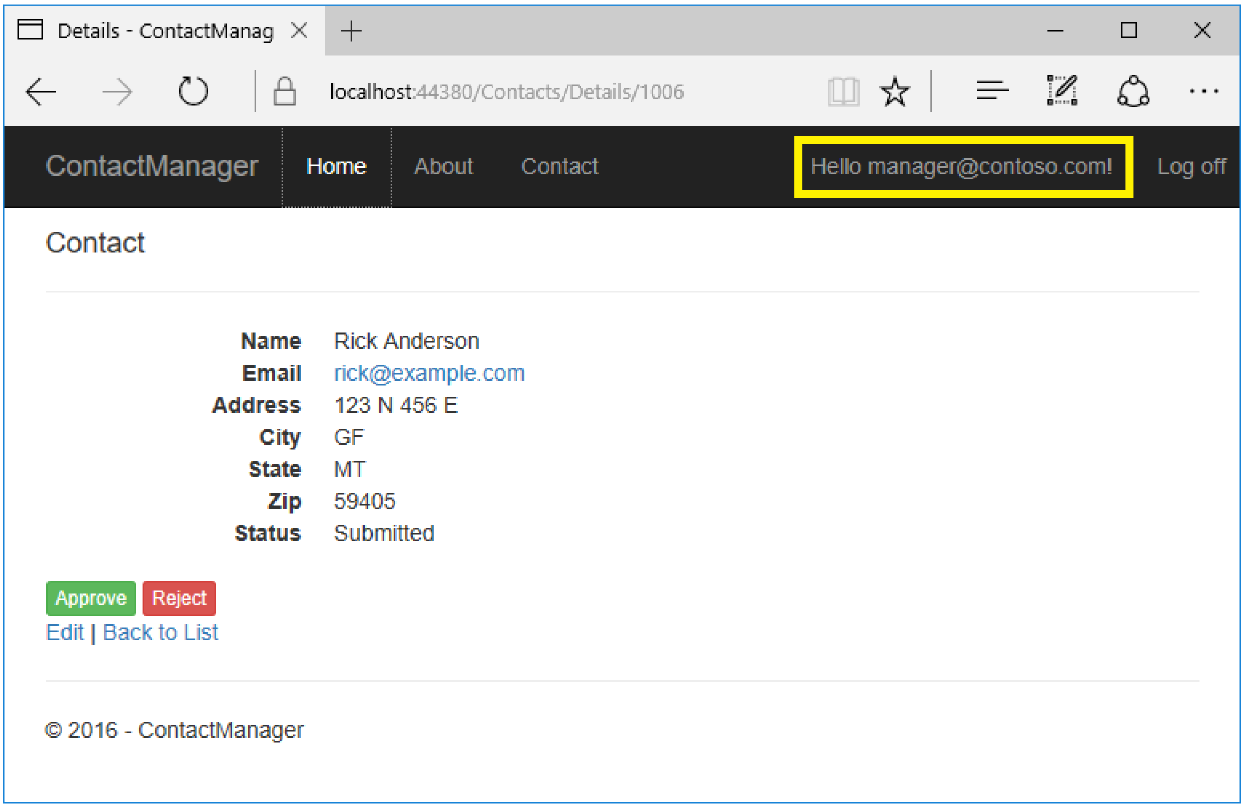
In the following image, user Rick (`rick@example.com`) is signed in. User Rick can only view approved contacts and edit/delete his contacts. Only the last record, created by Rick, displays edit and delete links



In the following image, `manager@contoso.com` is signed in and in the managers role.

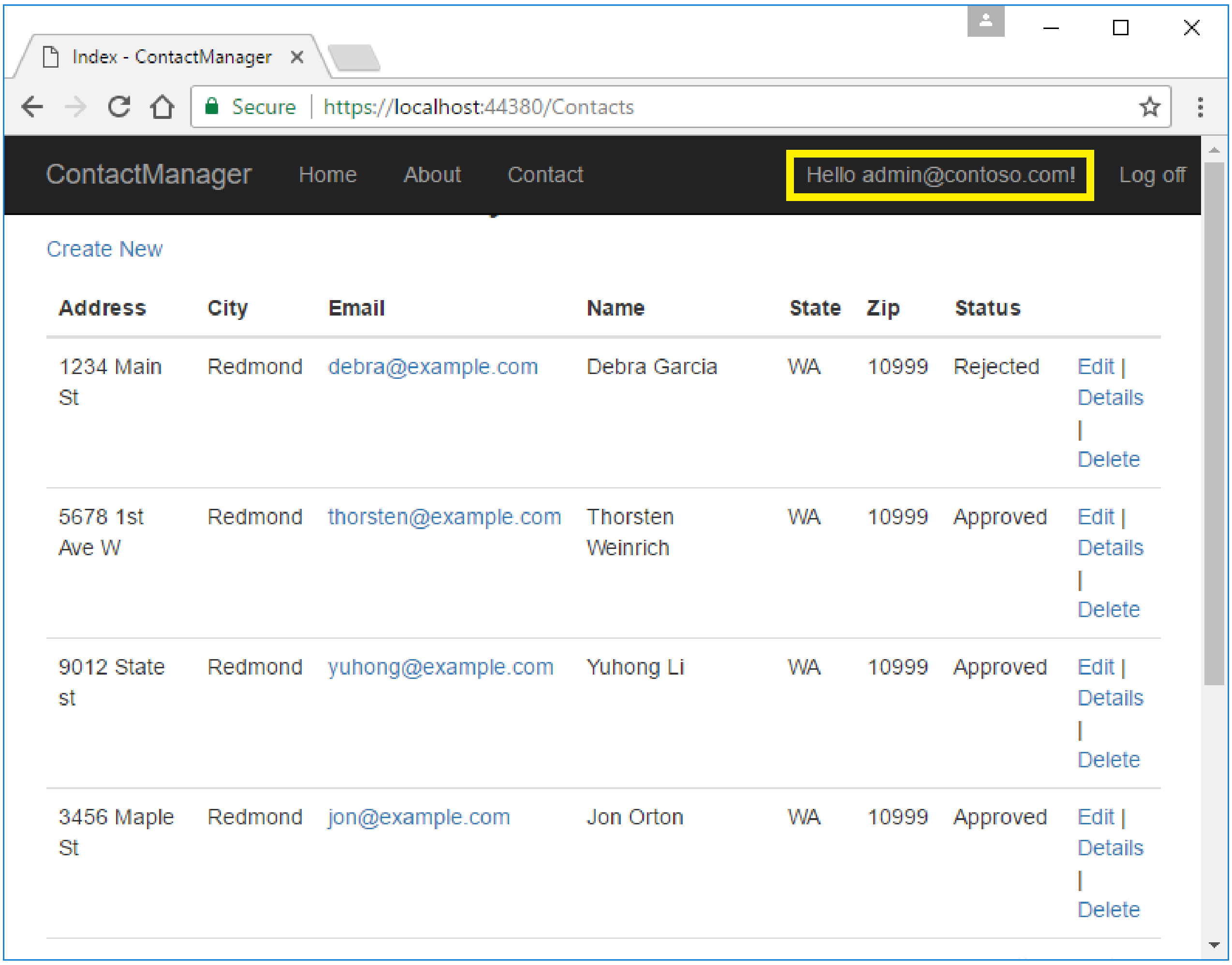


The following image shows the managers details view of a contact.



Only managers and administrators have the approve and reject buttons.

In the following image, `admin@contoso.com` is signed in and in the administrator's role.



The administrator has all privileges. She can read/edit/delete any contact and change the status of contacts.

The app was created by scaffolding the following `Contact` model:

```
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; }
    public string Email { get; set; }
}
```

A `ContactIsOwnerAuthorizationHandler` authorization handler ensures that a user can only edit their data. A `ContactManagerAuthorizationHandler` authorization handler allows managers to approve or reject contacts. A `ContactAdministratorsAuthorizationHandler` authorization handler allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This is not a beginning tutorial. You should be familiar with:

ASP.NET Core MVC
Entity Framework Core

The starter and completed app

Download the completed app. Test the completed app so you become familiar with its security features.

The starter app

It's helpful to compare your code with the completed sample.

Download the starter app.

See Create the starter app if you'd like to create it from scratch.

Update the database:

```
dotnet ef database update
```

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact.

This tutorial has all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Modify the app to secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET Identity user ID to ensure users can edit their data, but not other users data. Add `OwnerID` to the `Contact` model:

```
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` is the user's ID from the `AspNetUser` table in the Identity database. The `Status` field determines if a contact is viewable by general users.

Scaffold a new migration and update the database:

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Require SSL and authenticated users

In the `ConfigureServices` method of the *Startup.cs* file, add the RequireHttpsAttribute authorization filter:

```
var skipSSL = Configuration.GetValue<bool>("LocalTest:skipSSL");
// requires using Microsoft.AspNetCore.Mvc;
services.Configure<MvcOptions>(options =>
{
// Set LocalTest:skipSSL to true to skip SSL requrement in
// debug mode. This is useful when not using Visual Studio.
if (_hostingEnv.IsDevelopment() && !skipSSL)
    {
        options.Filters.Add(new RequireHttpsAttribute());
    }
});
```

If you're using Visual Studio, see Set up IIS Express for SSL/HTTPS. To redirect HTTP requests to HTTPS, see URL Rewriting Middleware. If you are using Visual Studio Code or testing on local platform that doesn't include a test certificate for SSL:

Set `"LocalTest:skipSSL": true` in the *appsettings.json* file.

Require authenticated users

Set the default authentication policy to require users to be authenticated. You can opt out of authentication at the controller or action method with the `[AllowAnonymous]` attribute. With this approach, any new controllers added will automatically require authentication, which is safer than relying on new controllers to include the `[Authorize]` attribute. Add the following to the `ConfigureServices` method of the *Startup.cs* file:

```
// 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));
});
```

Add `[AllowAnonymous]` to the home controller so anonymous users can get information about the site before they register.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace ContactManager.Controllers
{
    [AllowAnonymous]
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
```

Configure the test account

The `SeedData` class creates two accounts, administrator and manager. Use the Secret Manager tool to set a password for these accounts. Do this from the project directory (the directory containing *Program.cs*).

```
dotnet user-secrets set SeedUserPW <PW>
```

Update `Configure` to use the test password:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseIdentity();

    app.UseMvcWithDefaultRoute();

    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>
    var testUserPw = Configuration["SeedUserPW"];

    if (String.IsNullOrEmpty(testUserPw))
    {
        throw new System.Exception("Use secrets manager to set SeedUserPW \n" +
                                   "dotnet user-secrets set SeedUserPW <pw>");
    }

    try
    {
        SeedData.Initialize(app.ApplicationServices, testUserPw).Wait();
    }
    catch
    {
        throw new System.Exception("You need to update the DB "
            + "\nPM > Update-Database " + "\n or \n" +
              "> dotnet ef database update"
              + "\nIf that doesn't work, comment out SeedData and "
              + "register a new user");
    }
```

Add the administrator user ID and `Status = ContactStatus.Approved` to the contacts. Only one contact is shown, add the user ID to all contacts:

```
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
        },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The `ContactIsOwnerAuthorizationHandler` will verify the user acting on the resource owns the resource.

```
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 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 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.FromResult(0);
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.FromResult(0);
        }
    }
}
```

The `ContactIsOwnerAuthorizationHandler` calls `context.Succeed` if the current authenticated user is the contact owner. Authorization handlers generally return `context.Succeed` when the requirements are met. They return `Task.FromResult(0)` when requirements are not met. `Task.FromResult(0)` is neither success or failure, it allows other authorization handler to run. If you need to explicitly fail, return `context.Fail()`.

We allow contact owners to edit/delete their own data, so we don't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` will verify the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

```
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>
    {
        UserManager<ApplicationUser> _userManager;

        public ContactManagerAuthorizationHandler(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 not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.FromResult(0);
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
                return Task.FromResult(0);
            }

            return Task.FromResult(0);
        }
    }
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` will verify the user acting on the resource is a administrator. Administrator can do all operations.

```
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.FromResult(0);
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.FromResult(0);
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for dependency injection using AddScoped. The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core Identity, which is built on Entity Framework Core. Register the handlers with the service collection so they will be available to the `ContactsController` through dependency injection. Add the following code to the end of `ConfigureServices`:

```
// Authorization handlers.
services.AddScoped<IAuthorizationHandler,
                      ContactIsOwnerAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
                      ContactAdministratorsAuthorizationHandler>();

services.AddSingleton<IAuthorizationHandler,
                      ContactManagerAuthorizationHandler>();
```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They are singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

The complete `ConfigureServices`:

```
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>();

    var skipSSL = Configuration.GetValue<bool>("LocalTest:skipSSL");
    // requires using Microsoft.AspNetCore.Mvc;
    services.Configure<MvcOptions>(options =>
    {
    // Set LocalTest:skipSSL to true to skip SSL requrement in
    // debug mode. This is useful when not using Visual Studio.
    if (_hostingEnv.IsDevelopment() && !skipSSL)
        {
            options.Filters.Add(new RequireHttpsAttribute());
        }
    });

    // 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>();
}
```

Update the code to support authorization

In this section, you update the controller and views and add an operations requirements class.

Update the Contacts controller

Update the `ContactsController` constructor:

Add the `IAuthorizationService` service to access to the authorization handlers.
Add the `Identity` `UserManager` service:

```
public class ContactsController : Controller
{
    private readonly ApplicationDbContext _context;
    private readonly IAuthorizationService _authorizationService;
    private readonly UserManager<ApplicationUser> _userManager;

    public ContactsController(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<ApplicationUser> userManager)
    {
        _context = context;
        _userManager = userManager;
        _authorizationService = authorizationService;
    }
```

Add a contact operations requirements class

Add the `ContactOperationsRequirements` class to the *Authorization* folder. This class contain the requirements our app supports:

```
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";
    }
}
```

Update Create

Update the `HTTP POST Create` method to:

Add the user ID to the `Contact` model.
Call the authorization handler to verify the user owns the contact.

```
// POST: Contacts/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(ContactEditViewModel editModel)
{
    if (!ModelState.IsValid)
    {
        return View(editModel);
    }

    var contact = ViewModel_to_model(new Contact(), editModel);

    contact.OwnerID = _userManager.GetUserId(User);

    var isAuthorized = await _authorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Create);
    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    _context.Add(contact);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

Update Edit

Update both `Edit` methods to use the authorization handler to verify the user owns the contact. Because we are performing resource authorization we cannot use the `[Authorize]` attribute. We don't have access to the resource when attributes are evaluated. Resource based authorization must be imperative. Checks must be performed once we have access to the resource, either by loading it in our controller, or by loading it within the handler itself. Frequently you will access the resource by passing in the resource key.

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

    var contact = await _context.Contact.SingleOrDefaultAsync(
                                                m => m.ContactId == id);
    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Update);
    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    var editModel = Model_to_viewModel(contact);

    return View(editModel);
}

// POST: Contacts/Edit/5
[HttpPost]
```

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, ContactEditViewModel editModel)
{
    if (!ModelState.IsValid)
    {
        return View(editModel);
    }

    // Fetch Contact from DB to get OwnerID.
    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);
    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
                                                   ContactOperations.Update);
    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    contact = ViewModel_to_model(contact, editModel);

    if (contact.Status == ContactStatus.Approved)
    {
        // If the contact is updated after approval,
        // and the user cannot approve set the status back to submitted
        var canApprove = await _authorizationService.AuthorizeAsync(User, contact,
                                ContactOperations.Approve);

        if (!canApprove) contact.Status = ContactStatus.Submitted;
    }

    _context.Update(contact);
    await _context.SaveChangesAsync();

    return RedirectToAction("Index");
}
```

Update the Delete method

Update both `Delete` methods to use the authorization handler to verify the user owns the contact.

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

    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);
    if (contact == null)
    {
        return NotFound();
    }

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
                                ContactOperations.Delete);
    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    return View(contact);
}

// POST: Contacts/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var contact = await _context.Contact.SingleOrDefaultAsync(m => m.ContactId == id);

    var isAuthorized = await _authorizationService.AuthorizeAsync(User, contact,
                                ContactOperations.Delete);
    if (!isAuthorized)
    {
        return new ChallengeResult();
    }

    _context.Contact.Remove(contact);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

Inject the authorization service into the views

Currently the UI shows edit and delete links for data the user cannot modify. We'll fix that by applying the authorization handler to the views.

Inject the authorization service in the *Views/_ViewImports.cshtml* file so it will be available to all views:

```
@using ContactManager
@using ContactManager.Models
@using ContactManager.Models.AccountViewModels
@using ContactManager.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

Update the *Views/Contacts/Index.cshtml* Razor view to only display the edit and delete links for users who can edit/delete the contact.

Add `@using ContactManager.Authorization;`

Update the `Edit` and `Delete` links so they are only rendered for users with permission to edit and delete the contact.

```
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.Zip)
    </td>
    <td>
        @Html.DisplayFor(modelItem => item.Status)
    </td>
    <td>
        @if (await AuthorizationService.AuthorizeAsync(User,
                                         item, ContactOperations.Update))
        {
            <a asp-action="Edit" asp-route-id="@item.ContactId">Edit</a><text> | </text>
        }
        <a asp-action="Details" asp-route-id="@item.ContactId">Details</a>
        @if (await AuthorizationService.AuthorizeAsync(User,
                                           item, ContactOperations.Delete))
        {
            <text> | </text>
            <a asp-action="Delete" asp-route-id="@item.ContactId">Delete</a>
        }
    </td>
</tr>
```

Warning: Hiding links from users that do not have permission to edit or delete data does not secure the app. Hiding links makes the app more user friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The controller must repeat the access checks to be secure.

Update the Details view

Update the details view so managers can approve or reject contacts:

```
        <dd>
            @Html.DisplayFor(model => model.Status)
        </dd>
    </dl>
</div>
@if (Model.Status != ContactStatus.Approved)
{
    @if (await AuthorizationService.AuthorizeAsync(User, Model, ContactOperations.Approve))
    {
        <form asp-action="SetStatus" asp-controller="Contacts" style="display:inline;">
            <input type="hidden" name="id" value="@Model.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}
@if (Model.Status != ContactStatus.Rejected)
{
    @if (await AuthorizationService.AuthorizeAsync(User, Model, ContactOperations.Reject))
    {
        <form asp-action="SetStatus" asp-controller="Contacts" style="display:inline;">
            <input type="hidden" name="id" value="@Model.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-danger">Reject</button>
        </form>
    }
}
<div>
    <a asp-action="Edit" asp-route-id="@Model.ContactId">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Test the completed app

If you are using Visual Studio Code or testing on local platform that doesn't include a test certificate for SSL:

Set `"LocalTest:skipSSL": true` in the *appsettings.json* file.

If you have run the app and have contacts, delete all the records in the `Contact` table and restart the app to seed the database. If you are using Visual Studio, you need to exit and restart IIS Express to seed the database.

Register a user to browse the contacts.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate versions). In one browser, register a new user, for example, `test@example.com`. Sign in to each browser with a different user. Verify the following:

Registered users can view all the approved contact data.
Registered users can edit/delete their own data.
Managers can approve or reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
Administrators can approve/reject and edit/delete any data.

USER	OPTIONS
test@example.com	Can edit/delete own data
manager@contoso.com	Can approve/reject and edit/delete own data
admin@contoso.com	Can edit/delete and approve/reject all data

Create a contact in the administrators browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user cannot perform these operations.

Create the starter app

Follow these instructions to create the starter app.

Create an **ASP.NET Core Web Application** using Visual Studio 2017 named "ContactManager"

Create the app with **Individual User Accounts**.
Name it "ContactManager" so your namespace will match the namespace use in the sample.

Add the following `Contact` model:

```
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; }
    public string Email { get; set; }
}
```

Scaffold the `Contact` model using Entity Framework Core and the `ApplicationDbContext` data context. Accept all the scaffolding defaults. Using `ApplicationDbContext` for the data context class puts the contact table in the Identity database. See Adding a model for more information.

Update the **ContactManager** anchor in the *Views/Shared/_Layout.cshtml* file from `asp-controller="Home"` to `asp-controller="Contacts"` so tapping the **ContactManager** link will invoke the Contacts controller. The original markup:

```
    <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">ContactManager</a>
```

The updated markup:

```
    <a asp-area="" asp-controller="Contacts" asp-action="Index" class="navbar-brand">ContactManager</a>
```

Scaffold the initial migration and update the database

```
    dotnet ef migrations add initial
    dotnet ef database update
```

Test the app by creating, editing and deleting a contact

Seed the database

Add the `SeedData` class to the *Data* folder. If you've downloaded the sample, you can copy the *SeedData.cs* file to the *Data* folder of the starter project.

```

using ContactManager.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;
using System.Threading.Tasks;
```

```
namespace ContactManager.Data
{
    public static class SeedData
    {
        public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
            {
                var uid = await CreateTestUser(serviceProvider, testUserPw);
                SeedDB(context, uid);
            }
        }

        private static async Task<string> CreateTestUser(IServiceProvider serviceProvider, string testUserPw)
        {
            if (String.IsNullOrEmpty(testUserPw))
                return "";

            const string SeedUserName = "test@example.com";

            var userManager = serviceProvider.GetService<UserManager<ApplicationUser>>();

            var user = await userManager.FindByNameAsync(SeedUserName);
            if (user == null)
            {
                user = new ApplicationUser { UserName = SeedUserName };
                await userManager.CreateAsync(user, testUserPw);
            }

            return user.Id;
        }

        public static void SeedDB(ApplicationDbContext context, string uid)
        {
            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"
                },
             new Contact
             {
                 Name = "Thorsten Weinrich",
                 Address = "5678 1st Ave W",
                 City = "Redmond",
                 State = "WA",
                 Zip = "10999",
                 Email = "thorsten@example.com"
             },
             new Contact
             {
                 Name = "Yuhong Li",
                 Address = "9012 State st",
                 City = "Redmond",
                 State = "WA",
```

```
                  Zip = "10999",
                  Email = "yuhong@example.com"
              },
              new Contact
              {
                  Name = "Jon Orton",
                  Address = "3456 Maple St",
                  City = "Redmond",
                  State = "WA",
                  Zip = "10999",
                  Email = "jon@example.com"
              },
              new Contact
              {
                  Name = "Diliana Alexieva-Bosseva",
                  Address = "7890 2nd Ave E",
                  City = "Redmond",
                  State = "WA",
                  Zip = "10999",
                  Email = "diliana@example.com"
              }
              );
             context.SaveChanges();
         }
     }
}
```

Add the highlighted code to the end of the `Configure` method in the *Startup.cs* file:

```
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?}");
    });

    try
    {
        SeedData.Initialize(app.ApplicationServices, "").Wait();
    }
    catch
    {
        throw new System.Exception("You need to update the DB "
            + "\nPM > Update-Database " + "\n or \n" +
              "> dotnet ef database update"
              + "\nIf that doesn't work, comment out SeedData and register a new user");
    }
}
```

Test that the app seeded the database. The seed method does not run if there are any rows in the contact DB.

Create a class used in the tutorial

Create a folder named *Authorization*.
Copy the *Authorization\ContactOperations.cs* file from the completed project download, or copy the following code:

```
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";
    }
}
```

Additional resources

ASP.NET Core Authorization Lab. This lab goes into more detail on the security features introduced in this tutorial.

Authorization in ASP.NET Core : Simple, role, claims-based and custom

Custom Policy-Based Authorization

Simple Authorization

Authorization in MVC is controlled through the `AuthorizeAttribute` attribute and its various parameters. At its simplest applying the `AuthorizeAttribute` attribute to a controller or action limits access to the controller or action to any authenticated user.

For example, the following code limits access to the `AccountController` to any authenticated user.

```
[Authorize]
   public class AccountController : Controller
   {
       public ActionResult Login()
       {
       }

       public ActionResult Logout()
       {
       }
   }
```

If you want to apply authorization to an action rather than the controller simply apply the `AuthorizeAttribute` attribute to the action itself;

```
public class AccountController : Controller
   {
       public ActionResult Login()
       {
       }

       [Authorize]
       public ActionResult Logout()
       {
       }
   }
```

Now only authenticated users can access the logout function.

You can also use the `AllowAnonymousAttribute` attribute to allow access by non-authenticated users to individual actions; for example

```
[Authorize]
   public class AccountController : Controller
   {
       [AllowAnonymous]
       public ActionResult Login()
       {
       }

       public ActionResult Logout()
       {
       }
   }
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

Warning

`[AllowAnonymous]` bypasses all authorization statements. If you apply combine `[AllowAnonymous]` and any `[Authorize]` attribute then the Authorize attributes will always be ignored. For example if you apply `[AllowAnonymous]` at the controller level any `[Authorize]` attributes on the same controller, or on any action within it will be ignored.

Role based Authorization

When an identity is created it may belong to one or more roles, for example Tracy may belong to the Administrator and User roles whilst Scott may only belong to the user role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole property on the ClaimsPrincipal class.

Adding role checks

Role based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

For example the following code would limit access to any actions on the `AdministrationController` to users who are a member of the `Administrator` group.

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

You can specify multiple roles as a comma separated list;

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

This controller would be only accessible by users who are members of the `HRManager` role or the `Finance` role.

If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the `PowerUser` and `ControlPanelUser` role.

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

You can further limit access by applying additional role authorization attributes at the action level;

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the previous code snippet members of the `Administrator` role or the `PowerUser` role can access the controller and the `SetTime` action, but only members of the `Administrator` role can access the `ShutDown` action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions.

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

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

Policy based role checks

Role requirements can also be expressed using the new Policy syntax, where a developer registers a policy at startup as part of the Authorization service configuration. This normally occurs in `ConfigureServices()` in your *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole", policy => policy.RequireRole("Administrator"));
    });
}
```

Policies are applied using the `Policy` property on the `AuthorizeAttribute` attribute;

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

If you want to specify multiple allowed roles in a requirement then you can specify them as parameters to the `RequireRole` method;

```
options.AddPolicy("ElevatedRights", policy =>
                  policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

This example authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Claims-Based Authorization

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is name value pair that represents what the subject is, not what the subject can do. For example you may have a Drivers License, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and does not check the value.

First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in `ConfigureServices()` in your *Startup.cs* file.

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

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

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

You then apply the policy using the `Policy` property on the `AuthorizeAttribute` attribute to specify the policy name;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `AuthorizeAttribute` attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

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

If you have a controller that is protected by the `AuthorizeAttribute` attribute, but want to allow anonymous access to particular actions you apply the `AllowAnonymousAttribute` attribute.

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

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

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
                          policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

Multiple Policy Evaluation

If you apply multiple policies to a controller or action then all policies must pass before access is granted. For example:

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

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

In the above example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However in order to call the `UpdateSalary` action the identity must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it then checking the age is 21 or older then you need to write custom policy handlers.

Custom Policy-Based Authorization

Underneath the covers the role authorization and claims authorization make use of a requirement, a handler for the requirement and a pre-configured policy. These building blocks allow you to express authorization evaluations in code, allowing for a richer, reusable, and easily testable authorization structure.

An authorization policy is made up of one or more requirements and registered at application startup as part of the Authorization service configuration, in `ConfigureServices` in the *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Over21",
                          policy => policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

Here you can see an "Over21" policy is created with a single requirement, that of a minimum age, which is passed as a parameter to the requirement.

Policies are applied using the `Authorize` attribute by specifying the policy name, for example;

```
[Authorize(Policy="Over21")]
public class AlcoholPurchaseRequirementsController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our Minimum Age policy the requirement we have is a single parameter, the minimum age. A requirement must implement `IAuthorizationRequirement`. This is an empty, marker interface. A parameterized minimum age requirement might be implemented as follows;

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int age)
    {
        MinimumAge = age;
    }

    protected int MinimumAge { get; set; }
}
```

A requirement doesn't need to have data or properties.

Authorization Handlers

An authorization handler is responsible for the evaluation of any properties of a requirement. The authorization handler must evaluate them against a provided `AuthorizationHandlerContext` to decide if authorization is allowed. A requirement can have multiple handlers. Handlers must inherit `AuthorizationHandler<T>` where T is the requirement it handles.

The minimum age handler might look like this:

```
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"))
        {
            // .NET 4.x -> 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);
        }
        return Task.CompletedTask;
    }
}
```

In the code above we first look to see if the current user principal has a date of birth claim which has been issued by an Issuer we know and trust. If the claim is missing we can't authorize so we return. If we have a claim, we figure out how old the user is, and if they meet the minimum age passed in by the requirement then authorization has been successful. Once authorization is successful we call `context.Succeed()` passing in the requirement that has been successful as a parameter.

Handlers must be registered in the services collection during configuration, for example;

```

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Over21",
                          policy => policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

Each handler is added to the services collection by using `services.AddSingleton<IAuthorizationHandler, YourHandlerClass>();` passing in your handler class.

What should a handler return?

You can see in our handler example that the `Handle()` method has no return value, so how do we indicate success or failure?

A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.

A handler does not need to handle failures generally, as other handlers for the same requirement may succeed.

To guarantee failure even if other handlers for a requirement succeed, call `context.Fail`.

Regardless of what you call inside your handler all handlers for a requirement will be called when a policy requires the requirement. This allows requirements to have side effects, such as logging, which will always take place even if `context.Fail()` has been called in another handler.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis you implement multiple handlers for a single requirement. For example, Microsoft has doors which only open with key cards. If you leave your key card at home the receptionist prints a temporary sticker and opens the door for you. In this scenario you'd have a single requirement, *EnterBuilding*, but multiple handlers, each one examining a single requirement.

```
public class EnterBuildingRequirement : IAuthorizationRequirement
{
}

public class BadgeEntryHandler : AuthorizationHandler<EnterBuildingRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
EnterBuildingRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == ClaimTypes.BadgeId &&
                                       c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

public class HasTemporaryStickerHandler : AuthorizationHandler<EnterBuildingRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
EnterBuildingRequirement 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);
        }
        return Task.CompletedTask;
    }
}
```

Now, assuming both handlers are registered when a policy evaluates the `EnterBuildingRequirement` if either handler succeeds the policy evaluation will succeed.

Using a func to fufill a policy

There may be occasions where fufilling a policy is simple to express in code. It is possible to simply supply a `Func<AuthorizationHandlerContext, bool>` when configuring your policy with the `RequireAssertion` policy builder.

For example the previous `BadgeEntryHandler` could be rewritten as follows;

```
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"));
                          }));
    }
 }
```

Accessing MVC Request Context In Handlers

The `Handle` method you must implement in an authorization handler has two parameters, an `AuthorizationContext` and the `Requirement` you are handling. Frameworks such as MVC or Jabbr are free to add any object to the `Resource` property on the `AuthorizationContext` to pass through extra information.

For example MVC passes an instance of `Microsoft.AspNetCore.Mvc.Filters.AuthorizationFilterContext` in the resource property which is used to access HttpContext, RouteData and everything else MVC provides.

The use of the `Resource` property is framework specific. Using information in the `Resource` property will limit your authorization policies to particular frameworks. You should cast the `Resource` property using the `as` keyword, and then check the cast has succeed to ensure your code doesn't crash with `InvalidCastExceptions` when run on other frameworks;

```
var mvcContext = context.Resource as Microsoft.AspNetCore.Mvc.Filters.AuthorizationFilterContext;

if (mvcContext != null)
{
    // Examine MVC specific things like routing data.
}
```

Dependency Injection in requirement handlers

Authorization handlers must be registered in the service collection during configuration (using dependency injection).

Suppose you had a repository of rules you wanted to evaluate inside an authorization handler and that repository was registered in the service collection. Authorization will resolve and inject that into your constructor.

For example, if you wanted to use ASP.NET's logging infrastructure you would to inject `ILoggerFactory` into your handler. Such a handler might look like:

```
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;
       }
   }
```

You would register the handler with `services.AddSingleton()`:

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

An instance of the handler will be created when your application starts, and DI will inject the registered `ILoggerFactory` into your constructor.

 Note

Handlers that use Entity Framework shouldn't be registered as singletons.

Resource Based Authorization

Often authorization depends upon the resource being accessed. For example a document may have an author property. Only the document author would be allowed to update it, so the resource must be loaded from the document repository before an authorization evaluation can be made. This cannot be done with an Authorize attribute, as attribute evaluation takes place before data binding and before your own code to load a resource runs inside an action. Instead of declarative authorization, the attribute method, we must use imperative authorization, where a developer calls an authorize function within their own code.

Authorizing within your code

Authorization is implemented as a service, `IAuthorizationService`, registered in the service collection and available via dependency injection for Controllers to access.

```
public class DocumentController : Controller
   {
       IAuthorizationService _authorizationService;

       public DocumentController(IAuthorizationService authorizationService)
       {
           _authorizationService = authorizationService;
       }
   }
```

`IAuthorizationService` has two methods, one where you pass the resource and the policy name and the other where you pass the resource and a list of requirements to evaluate.

```
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                             object resource,
                             IEnumerable<IAuthorizationRequirement> requirements);
   Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                             object resource,
                             string policyName);
```

To call the service load your resource within your action then call the `AuthorizeAsync` overload you require. For example

```
public async Task<IActionResult> Edit(Guid documentId)
   {
       Document document = documentRepository.Find(documentId);

       if (document == null)
       {
           return new HttpNotFoundResult();
       }

       if (await _authorizationService.AuthorizeAsync(User, document, "EditPolicy"))
       {
           return View(document);
       }
       else
       {
           return new ChallengeResult();
       }
   }
```

Writing a resource based handler

Writing a handler for resource based authorization is not that much different to writing a plain requirements handler. You create a requirement, and then implement a handler for the requirement, specifying the requirement as before and also the resource type. For example, a handler which might accept a Document resource would look as follows;

```
public class DocumentAuthorizationHandler : AuthorizationHandler<MyRequirement, Document>
   {
       public override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   MyRequirement requirement,
                                                   Document resource)
       {
           // Validate the requirement against the resource and identity.

           return Task.CompletedTask;
       }
   }
```

Don't forget you also need to register your handler in the `ConfigureServices` method;

```
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
```

Operational Requirements

If you are making decisions based on operations such as read, write, update and delete, you can use the `OperationAuthorizationRequirement` class in the `Microsoft.AspNetCore.Authorization.Infrastructure` namespace. This prebuilt requirement class enables you to write a single handler which has a parameterized operation name, rather than create individual classes for each operation. To use it provide some operation names:

```
public static class Operations
   {
       public static OperationAuthorizationRequirement Create =
           new OperationAuthorizationRequirement { Name = "Create" };
       public static OperationAuthorizationRequirement Read =
           new OperationAuthorizationRequirement   { Name = "Read" };
       public static OperationAuthorizationRequirement Update =
           new OperationAuthorizationRequirement { Name = "Update" };
       public static OperationAuthorizationRequirement Delete =
           new OperationAuthorizationRequirement { Name = "Delete" };
   }
```

Your handler could then be implemented as follows, using a hypothetical `Document` class as the resource;

```
public class DocumentAuthorizationHandler :
       AuthorizationHandler<OperationAuthorizationRequirement, Document>
   {
       public override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   OperationAuthorizationRequirement requirement,
                                                   Document resource)
       {
           // Validate the operation using the resource, the identity and
           // the Name property value from the requirement.

           return Task.CompletedTask;
       }
   }
```

You can see the handler works on `OperationAuthorizationRequirement`. The code inside the handler must take the Name property of the supplied requirement into account when making its evaluations.

To call an operational resource handler you need to specify the operation when calling `AuthorizeAsync` in your action. For example

```
if (await _authorizationService.AuthorizeAsync(User, document, Operations.Read))
   {
       return View(document);
   }
   else
   {
       return new ChallengeResult();
   }
```

This example checks if the User is able to perform the Read operation for the current `document` instance. If authorization succeeds the view for the document will be returned. If authorization fails returning `ChallengeResult` will inform any authentication middleware authorization has failed and the middleware can take the appropriate response, for example returning a 401 or 403 status code, or redirecting the user to a login page for interactive browser clients.

View Based Authorization

Often a developer will want to show, hide or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via dependency injection. To inject the authorization service into a Razor view use the `@inject` directive, for example `@inject IAuthorizationService AuthorizationService` (requires `@using Microsoft.AspNetCore.Authorization`). If you want the authorization service in every view then place the `@inject` directive into the `_ViewImports.cshtml` file in the `Views` directory. For more information on dependency injection into views see Dependency injection into views.

Once you have injected the authorization service you use it by calling the `AuthorizeAsync` method in exactly the same way as you would check during resource based authorization.

```
@if (await AuthorizationService.AuthorizeAsync(User, "PolicyName"))
   {
       <p>This paragraph is displayed because you fulfilled PolicyName.</p>
   }
```

In some cases the resource will be your view model, and you can call `AuthorizeAsync` in exactly the same way as you would check during resource based authorization;

```
@if (await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit))
   {
       <p><a class="btn btn-default" role="button"
           href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>
   }
```

Here you can see the model is passed as the resource authorization should take into consideration.

 Warning

Do not rely on showing or hiding parts of your UI as your only authorization method. Hiding a UI element does not mean a user cannot access it. You must also authorize the user within your controller code.

Limiting identity by scheme

In some scenarios, such as Single Page Applications it is possible to end up with multiple authentication methods. For example, your application may use cookie-based authentication to log in and bearer authentication for JavaScript requests. In some cases you may have multiple instances of an authentication middleware. For example, two cookie middlewares where one contains a basic identity and one is created when a multi-factor authentication has triggered because the user requested an operation that requires extra security.

Authentication schemes are named when authentication middleware is configured during authentication, for example

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
{
    AuthenticationScheme = "Cookie",
    LoginPath = new PathString("/Account/Unauthorized/"),
    AccessDeniedPath = new PathString("/Account/Forbidden/"),
    AutomaticAuthenticate = false
});

app.UseBearerAuthentication(options =>
{
    options.AuthenticationScheme = "Bearer";
    options.AutomaticAuthenticate = false;
});
```

In this configuration two authentication middlewares have been added, one for cookies and one for bearer.

Note

When adding multiple authentication middleware you should ensure that no middleware is configured to run automatically. You do this by setting the `AutomaticAuthenticate` options property to false. If you fail to do this filtering by scheme will not work.

Selecting the scheme with the Authorize attribute

As no authentication middleware is configured to automatically run and create an identity you must, at the point of authorization choose which middleware will be used. The simplest way to select the middleware you wish to authorize with is to use the `ActiveAuthenticationSchemes` property. This property accepts a comma delimited list of Authentication Schemes to use. For example;

```
[Authorize(ActiveAuthenticationSchemes = "Cookie,Bearer")]
public class MixedController : Controller
```

In the example above both the cookie and bearer middlewares will run and have a chance to create and append an identity for the current user. By specifying a single scheme only the specified middleware will run;

```
[Authorize(ActiveAuthenticationSchemes = "Bearer")]
```

In this case only the middleware with the Bearer scheme would run, and any cookie based identities would be ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in policy you can set the `AuthenticationSchemes` collection when adding your policy.

```
options.AddPolicy("Over18", policy =>
{
    policy.AuthenticationSchemes.Add("Bearer");
    policy.RequireAuthenticatedUser();
    policy.Requirements.Add(new Over18Requirement());
});
```

In this example the Over18 policy will only run against the identity created by the `Bearer` middleware.

Introduction to Data Protection

Web applications often need to store security-sensitive data. Windows provides DPAPI for desktop applications but this is unsuitable for web applications. The ASP.NET Core data protection stack provide a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

The ASP.NET Core data protection stack is designed to serve as the long-term replacement for the element in ASP.NET 1.x - 4.x. It was designed to address many of the shortcomings of the old cryptographic stack while providing an out-of-the-box solution for the majority of use cases modern applications are likely to encounter.

Problem statement

The overall problem statement can be succinctly stated in a single sentence: I need to persist trusted information for later retrieval, but I do not trust the persistence mechanism. In web terms, this might be written as "I need to round-trip trusted state via an untrusted client."

The canonical example of this is an authentication cookie or bearer token. The server generates an "I am Groot and have xyz permissions" token and hands it to the client. At some future date the client will present that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token. Thus the first requirement: authenticity (a.k.a. integrity, tamper-proofing).

Since the persisted state is trusted by the server, we anticipate that this state might contain information that is specific to the operating environment. This could be in the form of a file path, a permission, a handle or other indirect reference, or some other piece of server-specific data. Such information should generally not be disclosed to an untrusted client. Thus the second requirement: confidentiality.

Finally, since modern applications are componentized, what we've seen is that individual components will want to take advantage of this system without regard to other components in the system. For instance, if a bearer token component is using this stack, it should operate without interference from an anti-CSRF mechanism that might also be using the same stack. Thus the final requirement: isolation.

We can provide further constraints in order to narrow the scope of our requirements. We assume that all services operating within the cryptosystem are equally trusted and that the data does not need to be generated or consumed outside of the services under our direct control. Furthermore, we require that operations are as fast as possible since each request to the web service might go through the cryptosystem one or more times. This makes symmetric cryptography ideal for our scenario, and we can discount asymmetric cryptography until such a time that it is needed.

Design philosophy

We started by identifying problems with the existing stack. Once we had that, we surveyed the landscape of existing solutions and concluded that no existing solution quite had the capabilities we sought. We then engineered a solution based on several guiding principles.

The system should offer simplicity of configuration. Ideally the system would be zero-configuration and developers could hit the ground running. In situations where developers need to configure a specific aspect (such as the key repository), consideration should be given to making those specific configurations simple.

Offer a simple consumer-facing API. The APIs should be easy to use correctly and difficult to use incorrectly.

Developers should not learn key management principles. The system should handle algorithm selection and key lifetime on the developer's behalf. Ideally the developer should never even have access to the raw key material.

Keys should be protected at rest when possible. The system should figure out an appropriate default protection mechanism and apply it automatically.

With these principles in mind we developed a simple, easy to use data protection stack.

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data.

Audience

The data protection system is divided into five main packages. Various aspects of these APIs target three main audiences;

The Consumer APIs Overview target application and framework developers.

"I don't want to learn about how the stack operates or about how it is configured. I simply want to perform some operation in as simple a manner as possible with high probability of using the APIs successfully."

The configuration APIs target application developers and system administrators.

"I need to tell the data protection system that my environment requires non-default paths or settings."

The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs would be limited to rare situations and experienced, security aware developers.

"I need to replace an entire component within the system because I have truly unique behavioral requirements. I am willing to learn uncommonly-used parts of the API surface in order to build a plugin that fulfills my requirements."

Package Layout

The data protection stack consists of five packages.

Microsoft.AspNetCore.DataProtection.Abstractions contains the basic IDataProtectionProvider and IDataProtector interfaces. It also contains useful extension methods that can assist working with these types (e.g., overloads of IDataProtector.Protect). See the consumer interfaces section for more information. If somebody else is responsible for instantiating the data protection system and you are simply consuming the APIs, you'll want to reference Microsoft.AspNetCore.DataProtection.Abstractions.

Microsoft.AspNetCore.DataProtection contains the core implementation of the data protection system, including the core cryptographic operations, key management, configuration, and extensibility. If you're responsible for instantiating the data protection system (e.g., adding it to an IServiceCollection) or modifying or extending its behavior, you'll want to reference Microsoft.AspNetCore.DataProtection.

Microsoft.AspNetCore.DataProtection.Extensions contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains a simple "instantiate the system pointing at a specific key storage directory with no dependency injection setup" API (more info). It also contains extension methods for limiting the lifetime of protected payloads (more info).

Microsoft.AspNetCore.DataProtection.SystemWeb can be installed into an existing ASP.NET 4.x application to redirect its operations to instead use the new data protection stack. See compatibility for more information.

Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems which need to handle user passwords securely. See Password Hashing for more information.

Getting Started with the Data Protection APIs

At its simplest protecting data consists of the following steps:

Create a data protector from a data protection provider.

Call the Protect method with the data you want to protect.

Call the Unprotect method with the data you want to turn back into plain text.

Most frameworks such as ASP.NET or SignalR already configure the data protection system and add it to a service container you access via dependency injection. The following sample demonstrates configuring a service container for dependency injection and registering the data protection stack, receiving the data protection provider via DI, creating a protector and protecting then unprotecting data

```
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!
 */
```

When you create a protector you must provide one or more Purpose Strings. A purpose string provides isolation between consumers, for example a protector created with a purpose string of "green" would not be able to unprotect data provided by a protector with a purpose of "purple".

Tip

Instances of IDataProtectionProvider and IDataProtector are thread-safe for multiple callers. It is intended that once a component gets a reference to an IDataProtector via a call to CreateProtector, it will use that reference for multiple calls to Protect and Unprotect.

A call to Unprotect will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Consumer APIs overview

The IDataProtectionProvider and IDataProtector interfaces are the basic interfaces through which consumers use the data protection system. They are located in the Microsoft.AspNetCore.DataProtection.Abstractions package.

IDataProtectionProvider

The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an IDataProtector by calling IDataProtectionProvider.CreateProtector(purpose), where purpose is a string that describes the intended consumer use case. See Purpose Strings for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector

The protector interface is returned by a call to CreateProtector, and it is this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the Protect method. The basic interface defines a method which converts byte[] -> byte[], but there is also an overload (provided as an extension method) which converts string -> string. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for their use case. Irrespective of the overload chosen, the value returned by the Protect method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the Unprotect method. (There are byte[]-based and string-based overloads for developer convenience.) If the protected payload was generated by an earlier call to Protect on this same IDataProtector, the Unprotect method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different IDataProtector, the Unprotect method will throw CryptographicException.

The concept of same vs. different IDataProtector ties back to the concept of purpose. If two IDataProtector instances were generated from the same root IDataProtectionProvider but via different purpose strings in the call to IDataProtectionProvider.CreateProtector, then they are considered different protectors, and one will not be able to unprotect payloads generated by the other.

Consuming these interfaces

For a DI-aware component, the intended usage is that the component take an IDataProtectionProvider parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

▯ Note

Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the Non DI Aware Scenarios document for more information on getting an instance of an IDataProtection provider without going through DI.

The following sample demonstrates three concepts:

Adding the data protection system to the service container,

Using DI to receive an instance of an IDataProtectionProvider, and

Creating an IDataProtector from an IDataProtectionProvider and using it to protect and unprotect data.

```
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!
 */
```

The package Microsoft.AspNetCore.DataProtection.Abstractions contains an extension method IServiceProvider.GetDataProtector as a developer convenience. It encapsulates as a single operation both retrieving an IDataProtectionProvider from the service provider and calling IDataProtectionProvider.CreateProtector. The following sample demonstrates its usage.

```
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}");
    }
}
```

汢 Tip

Instances of IDataProtectionProvider and IDataProtector are thread-safe for multiple callers. It is intended that once a component gets a reference to an IDataProtector via a call to CreateProtector, it will use that reference for multiple calls to Protect and Unprotect.A call to Unprotect will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Purpose Strings

Components which consume IDataProtectionProvider must pass a unique *purposes* parameter to the CreateProtector method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.

In the diagram above IDataProtector instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

踰 Tip

Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.

A Contoso-authored component which is responsible for minting bearer tokens might use Contoso.Security.BearerToken as its purpose string. Or - even better - it might use Contoso.Security.BearerToken.v1 as its purpose string. Appending the version number allows a future version to use Contoso.Security.BearerToken.v2 as its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to CreateProtector is a string array, the above could have been instead specified as ["Contoso.Security.BearerToken", "v1"]. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

 Warning

Components should not allow untrusted user input to be the sole source of input for the purposes chain.

For example, consider a component Contoso.Messaging.SecureMessage which is responsible for storing secure messages. If the secure messaging component were to call CreateProtector([username]), then a malicious user might create an account with username "Contoso.Security.BearerToken" in an attempt to get the component to call CreateProtector(["Contoso.Security.BearerToken"]), thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.

A better purposes chain for the messaging component would be CreateProtector(["Contoso.Messaging.SecureMessage", "User: username"]), which provides proper isolation.

The isolation provided by and behaviors of IDataProtectionProvider, IDataProtector, and purposes are as follows:

For a given IDataProtectionProvider object, the CreateProtector method will create an IDataProtector object uniquely tied to both the IDataProtectionProvider object which created it and the purposes parameter which was passed into the method.

The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.

Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.

Two IDataProtector objects are equivalent if and only if they are created from equivalent IDataProtectionProvider objects with equivalent purposes parameters.

For a given IDataProtector object, a call to Unprotect(protectedData) will return the original unprotectedData if and only if protectedData := Protect(unprotectedData) for an equivalent IDataProtector object.

 Note

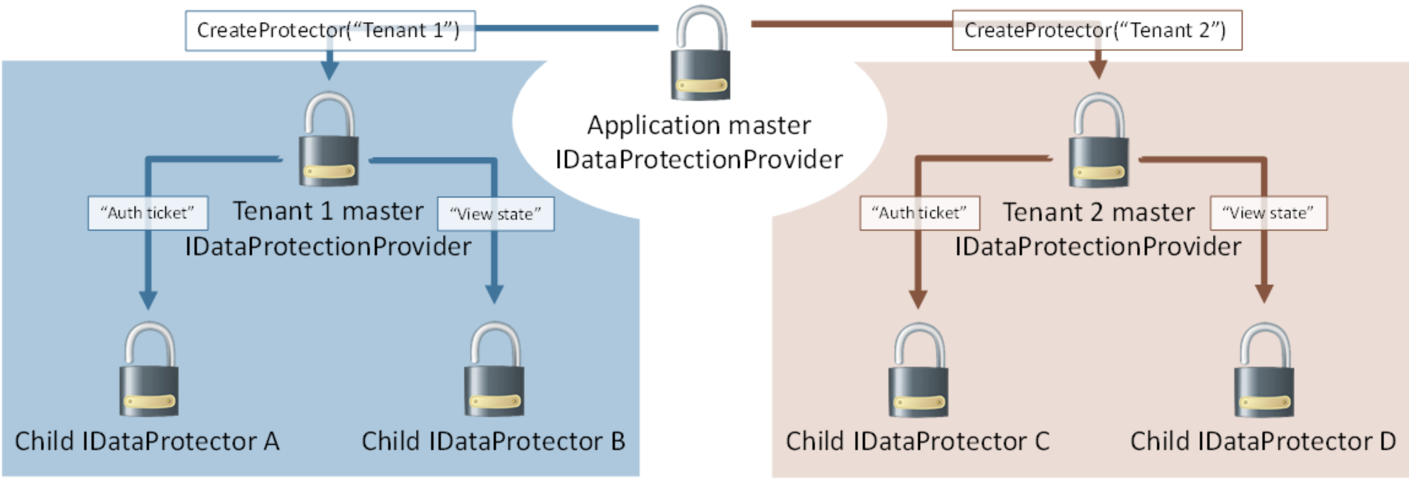
We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system is not intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

Purpose hierarchy and multi-tenancy in ASP.NET Core

Since an IDataProtector is also implicitly an IDataProtectionProvider, purposes can be chained together. In this sense provider.CreateProtector(["purpose1", "purpose2"]) is equivalent to provider.CreateProtector("purpose1").CreateProtector("purpose2").

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of Contoso.Messaging.SecureMessage, the SecureMessage component can call provider.CreateProtector("Contoso.Messaging.SecureMessage") once upfront and cache the result into a private _myProvider field. Future protectors can then be created via calls to _myProvider.CreateProtector("User: username"), and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls provider.CreateProtector("Tenant 1") and provider.CreateProtector("Tenant 2") to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically this is represented as below.



Warning

This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a tenant can execute arbitrary code, they could perform private reflection to break the isolation guarantees, or they could just read the master keying material directly and derive whatever subkeys they desire.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it is actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to isolate individual applications from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Password Hashing

The data protection code base includes a package *Microsoft.AspNetCore.Cryptography.KeyDerivation* which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used completely independently. The source exists alongside the data protection code base as a convenience.

The package currently offers a method `KeyDerivation.Pbkdf2` which allows hashing a password using the PBKDF2 algorithm. This API is very similar to the .NET Framework's existing Rfc2898DeriveBytes type, but there are three important distinctions:

The `KeyDerivation.Pbkdf2` method supports consuming multiple PRFs (currently `HMACSHA1`, `HMACSHA256`, and `HMACSHA512`), whereas the `Rfc2898DeriveBytes` type only supports `HMACSHA1`.

The `KeyDerivation.Pbkdf2` method detects the current operating system and attempts to choose the most optimized implementation of the routine, providing much better performance in certain cases. (On Windows 8, it offers around 10x the throughput of `Rfc2898DeriveBytes`.)

The `KeyDerivation.Pbkdf2` method requires the caller to specify all parameters (salt, PRF, and iteration count). The `Rfc2898DeriveBytes` type provides default values for these.

```
using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("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: /OOoOer10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */
```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

Limiting the lifetime of protected payloads

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It is certainly possible for the developer to create their own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package Microsoft.AspNetCore.DataProtection.Extensions contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs hang off of the ITimeLimitedDataProtector type.

API usage

The ITimeLimitedDataProtector interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an ITimeLimitedDataProtector, you'll first need an instance of a regular IDataProtector constructed with a specific purpose. Once the IDataProtector instance is available, call the IDataProtector.ToTimeLimitedDataProtector extension method to get back a protector with built-in expiration capabilities.

ITimeLimitedDataProtector exposes the following API surface and extension methods:

CreateProtector(string purpose) : ITimeLimitedDataProtector This API is similar to the existing IDataProtectionProvider.CreateProtector in that it can be used to create purpose chains from a root time-limited protector.

Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]

Protect(byte[] plaintext, TimeSpan lifetime) : byte[]

Protect(byte[] plaintext) : byte[]

Protect(string plaintext, DateTimeOffset expiration) : string

Protect(string plaintext, TimeSpan lifetime) : string

Protect(string plaintext) : string

In addition to the core Protect methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a DateTimeOffset) or as a relative time (from the current system time, via a TimeSpan). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]

Unprotect(byte[] protectedData) : byte[]

Unprotect(string protectedData, out DateTimeOffset expiration) : string

Unprotect(string protectedData) : string

The Unprotect methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the Unprotect method will throw CryptographicException.

▯ Warning

It is not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the non-DI code paths for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the Microsoft.AspNetCore.DataProtection.Extensions package.

```
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...nLk7Ok
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at ...'>>

 */
```

Unprotecting payloads whose keys have been revoked

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there is nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so IDataProtector.Unprotect can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as IDataProtector.Unprotect will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having Unprotect throw could lead to unacceptable data loss.

IPersistedDataProtector

To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an IPersistedDataProtector type. To get an instance of IPersistedDataProtector, simply get an instance of IDataProtector in the normal fashion and try casting the IDataProtector to IPersistedDataProtector.

 Note

Not all IDataProtector instances can be cast to IPersistedDataProtector. Developers should use the C# as operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

IPersistedDataProtector exposes the following API surface:

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
     out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There is no string-based overload. The two out parameters are as follows.

requiresMigration: will be set to true if the key used to protect this payload is no longer the active default key, e.g., the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on their business needs.

wasRevoked: will be set to true if the key used to protect this payload was revoked.

 Warning

Exercise extreme caution when passing ignoreRevocationErrors: true to the DangerousUnprotect method. If after calling this method the wasRevoked value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case only continue operating on the unprotected payload if you have some separate assurance that it is authentic, e.g. that it's coming from a secure database rather than being sent by an untrusted web client.

```
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 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.Write("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: CfDJ8LHIzUCX1ZVBn2BZ...
 * Round-tripped payload: Hello!
 * Revoking all keys in the key ring...
 * Calling Unprotect...
```

```
* Calling Unprotect...
* CryptographicException: The key {...} has been revoked.
* Calling DangerousUnprotect...
* Unprotected payload: Hello!
* Requires migration = True, was revoked = True
*/
```

Configuring data protection

When the data protection system is initialized it applies some default settings based on the operational environment. These settings are generally good for applications running on a single machine. There are some cases where a developer may want to change these (perhaps because their application is spread across multiple machines or for compliance reasons), and for these scenarios the data protection system offers a rich configuration API.

There is an extension method AddDataProtection which returns an IDataProtectionBuilder which itself exposes extension methods that you can chain together to configure various data protection options. For instance, to store keys at a UNC share instead of %LOCALAPPDATA% (the default), configure the system as follows:

```
public void ConfigureServices(IServiceCollection services)
   {
       services.AddDataProtection()
           .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));

   }
```

⚠ Warning

If you change the key persistence location, the system will no longer automatically encrypt keys at rest since it doesn't know whether DPAPI is an appropriate encryption mechanism.

You can configure the system to protect keys at rest by calling any of the ProtectKeysWith* configuration APIs. Consider the example below, which stores keys at a UNC share and encrypts those keys at rest with a specific X.509 certificate.

```
public void ConfigureServices(IServiceCollection services)
     {
         services.AddDataProtection()
             .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
             .ProtectKeysWithCertificate("thumbprint");
     }
```

See key encryption at rest for more examples and for discussion on the built-in key encryption mechanisms.

To configure the system to use a default key lifetime of 14 days instead of 90 days, consider the following example:

```
public void ConfigureServices(IServiceCollection services)
   {
       services.AddDataProtection()
           .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
   }
```

By default the data protection system isolates applications from one another, even if they're sharing the same physical key repository. This prevents the applications from understanding each other's protected payloads. To share protected payloads between two different applications, configure the system passing in the same application name for both applications as in the below example:

```
public void ConfigureServices(IServiceCollection services)
   {
       services.AddDataProtection()
           .SetApplicationName("my application");
   }
```

Finally, you may have a scenario where you do not want an application to automatically roll keys as they approach expiration. One example of this might be applications set up in a primary / secondary relationship, where only the primary application is responsible for key management concerns, and all secondary applications simply have a read-only view of the key ring. The secondary applications can be configured to treat the key ring as read-only by configuring the system as below:

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

Per-application isolation

When the data protection system is provided by an ASP.NET Core host, it will automatically isolate applications from one another, even if those applications are running under the same worker process account and are using the same master keying material. This is somewhat similar to the IsolateApps modifier from System.Web's element.

The isolation mechanism works by considering each application on the local machine as a unique tenant, thus the IDataProtector rooted for any given application automatically includes the application ID as a discriminator. The application's unique ID comes from one of two places.

If the application is hosted in IIS, the unique identifier is the application's configuration path. If an application is deployed in a farm environment, this value should be stable assuming that the IIS environments are configured similarly across all machines in the farm.

If the application is not hosted in IIS, the unique identifier is the physical path of the application.

The unique identifier is designed to survive resets - both of the individual application and of the machine itself.

This isolation mechanism assumes that the applications are not malicious. A malicious application can always impact any other application running under the same worker process account. In a shared hosting environment where applications are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between applications, including separating the applications' underlying key repositories.

If the data protection system is not provided by an ASP.NET Core host (e.g., if the developer instantiates it himself via the DataProtectionProvider concrete type), application isolation is disabled by default, and all applications backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide application isolation in this environment, call the SetApplicationName method on the configuration object, see the code sample above.

Changing algorithms

The data protection stack allows changing the default algorithm used by newly-generated keys. The simplest way to do this is to call UseCryptographicAlgorithms from the configuration callback, as in the below example.

```
services.AddDataProtection()
       .UseCryptographicAlgorithms(new AuthenticatedEncryptionSettings()
       {
           EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
           ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
       });
```

The default EncryptionAlgorithm and ValidationAlgorithm are AES-256-CBC and HMACSHA256, respectively. The default policy can be set by a system administrator via Machine Wide Policy, but an explicit call to UseCryptographicAlgorithms will override the default policy.

Calling UseCryptographicAlgorithms will allow the developer to specify the desired algorithm (from a predefined built-in list), and the developer does not need to worry about the implementation of the algorithm. For instance, in the scenario above the data protection system will attempt to use the CNG implementation of AES if running on Windows, otherwise it will fall back to the managed System.Security.Cryptography.Aes class.

The developer can manually specify an implementation if desired via a call to UseCustomCryptographicAlgorithms, as show in the below examples.

汢 Tip

Changing algorithms does not affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms

To specify custom managed algorithms, create a ManagedAuthenticatedEncryptionSettings instance that points to the implementation types.

```
serviceCollection.AddDataProtection()
       .UseCustomCryptographicAlgorithms(new ManagedAuthenticatedEncryptionSettings()
       {
           // a type that subclasses SymmetricAlgorithm
           EncryptionAlgorithmType = typeof(Aes),

           // specified in bits
           EncryptionAlgorithmKeySize = 256,

           // a type that subclasses KeyedHashAlgorithm
           ValidationAlgorithmType = typeof(HMACSHA256)
       });
```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of SymmetricAlgorithm and KeyedHashAlgorithm, though the system special-cases some values like typeof(Aes) for convenience.

 Note

The SymmetricAlgorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of >= 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm is not strictly required to be HMAC.

Specifying custom Windows CNG algorithms

To specify a custom Windows CNG algorithm using CBC-mode encryption + HMAC validation, create a CngCbcAuthenticatedEncryptionSettings instance that contains the algorithmic information.

```
services.AddDataProtection()
       .UseCustomCryptographicAlgorithms(new CngCbcAuthenticatedEncryptionSettings()
       {
           // passed to BCryptOpenAlgorithmProvider
           EncryptionAlgorithm = "AES",
           EncryptionAlgorithmProvider = null,

           // specified in bits
           EncryptionAlgorithmKeySize = 256,

           // passed to BCryptOpenAlgorithmProvider
           HashAlgorithm = "SHA256",
           HashAlgorithmProvider = null
       });
```

 Note

The symmetric block cipher algorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of >= 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption + validation, create a CngGcmAuthenticatedEncryptionSettings instance that contains the algorithmic information.

```
services.AddDataProtection()
       .UseCustomCryptographicAlgorithms(new CngGcmAuthenticatedEncryptionSettings()
       {
           // passed to BCryptOpenAlgorithmProvider
           EncryptionAlgorithm = "AES",
           EncryptionAlgorithmProvider = null,

           // specified in bits
           EncryptionAlgorithmKeySize = 256
       });
   });
```

Note

The symmetric block cipher algorithm must have a key length of ≥ 128 bits and a block size of exactly 128 bits, and it must support GCM encryption. The EncryptionAlgorithmProvider property can be set to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

Specifying other custom algorithms

Though not exposed as a first-class API, the data protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it is possible to keep all keys contained within an HSM and to provide a custom implementation of the core encryption and decryption routines. See IAuthenticatedEncryptorConfiguration in the core cryptography extensibility section for more information.

See also

Non DI Aware Scenarios

Machine Wide Policy

Key management and lifetime

Key Management

The system tries to detect its operational environment and provide good zero-configuration behavioral defaults. The heuristic used is as follows.

If the system is being hosted in Azure Web Sites, keys are persisted to the "%HOME%\ASP.NET\DataProtection-Keys" folder. This folder is backed by network storage and is synchronized across all machines hosting the application. Keys are not protected at rest. This folder supplies the key ring to all instances of an application in a single deployment slot. Separate deployment slots, such as Staging and Production, will not share a key ring. When you swap between deployment slots, for example swapping Staging to Production or using A/B testing, any system using data protection will not be able to decrypt stored data using the key ring inside the previous slot. This will lead to users being logged out of an ASP.NET application that uses the standard ASP.NET cookie middleware, as it uses data protection to protect its cookies. If you desire slot-independent key rings, use an external key ring provider, such as Azure Blob Storage, Azure Key Vault, a SQL store, or Redis cache.

If the user profile is available, keys are persisted to the "%LOCALAPPDATA%\ASP.NET\DataProtection-Keys" folder. Additionally, if the operating system is Windows, they'll be encrypted at rest using DPAPI.

If the application is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that is ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

If none of these conditions matches, keys are not persisted outside of the current process. When the process shuts down, all generated keys will be lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should good defaults for most applications similar to how the ASP.NET auto-generation routines worked in the past. The final, fall back option is the only scenario that truly requires the developer to specify configuration upfront if they want key persistence, but this fall-back would only occur in rare situations.

⚠ Warning

If the developer overrides this heuristic and points the data protection system at a specific key repository, automatic encryption of keys at rest will be disabled. At rest protection can be re-enabled via configuration.

Key Lifetime

Keys by default have a 90-day lifetime. When a key expires, the system will automatically generate a new key and set the new key as the active key. As long as retired keys remain on the system you will still be able to decrypt any data protected with them. See key management for more information.

Default Algorithms

The default payload protection algorithm used is AES-256-CBC for confidentiality and HMACSHA256 for authenticity. A 512-bit master key, rolled every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See subkey derivation for more information.

Machine Wide Policy

When running on Windows, the data protection system has limited support for setting default machine-wide policy for all applications which consume data protection. The general idea is that an administrator might wish to change some default setting (such as algorithms used or key lifetime) without needing to manually update every application on the machine.

 Warning

The system administrator can set default policy, but they cannot enforce it. The application developer can always override any value with one of their own choosing. The default policy only affects applications where the developer has not specified an explicit value for some particular setting.

Setting default policy

To set default policy, an administrator can set known values in the system registry under the following key.

Reg key: `HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection`

If you're on a 64-bit operating system and want to affect the behavior of 32-bit applications, remember also to configure the Wow6432Node equivalent of the above key.

The supported values are:

EncryptionType [string] - specifies which algorithms should be used for data protection. This value must be "CNG-CBC", "CNG-GCM", or "Managed" and is described in more detail below.

DefaultKeyLifetime [DWORD] - specifies the lifetime for newly-generated keys. This value is specified in days and must be ≥ 7.

KeyEscrowSinks [string] - specifies the types which will be used for key escrow. This value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly qualified name of a type which implements IKeyEscrowSink.

Encryption types

If EncryptionType is "CNG-CBC", the system will be configured to use a CBC-mode symmetric block cipher for confidentiality and HMAC for authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngCbcAuthenticatedEncryptionSettings type:

EncryptionAlgorithm [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in CBC mode.

EncryptionAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm EncryptionAlgorithm.

EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.

HashAlgorithm [string] - the name of a hash algorithm understood by CNG. This algorithm will be opened in HMAC mode.

HashAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm HashAlgorithm.

If EncryptionType is "CNG-GCM", the system will be configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngGcmAuthenticatedEncryptionSettings type:

EncryptionAlgorithm [string] - the name of a symmetric block cipher algorithm understood by CNG. This algorithm will be opened in Galois/Counter Mode.

EncryptionAlgorithmProvider [string] - the name of the CNG provider implementation which can produce the algorithm EncryptionAlgorithm.

EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric block cipher algorithm.

If EncryptionType is "Managed", the system will be configured to use a managed SymmetricAlgorithm for confidentiality and KeyedHashAlgorithm for authenticity (see Specifying custom managed algorithms for more details). The following additional values are supported, each of which corresponds to a property on the ManagedAuthenticatedEncryptionSettings type:

EncryptionAlgorithmType [string] - the assembly-qualified name of a type which implements SymmetricAlgorithm.

EncryptionAlgorithmKeySize [DWORD] - the length (in bits) of the key to derive for the symmetric encryption algorithm.

ValidationAlgorithmType [string] - the assembly-qualified name of a type which implements KeyedHashAlgorithm.

If EncryptionType has any other value (other than null / empty), the data protection system will throw an exception at startup.

⚠ Warning

When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the application. In practice, this means that for applications running on Desktop CLR, the assemblies which contain these types should be GACed. For ASP.NET Core applications running on .NET Core, the packages which contain these types should be installed.

Non DI aware scenarios

The data protection system is normally designed to be added to a service container and to be provided to dependent components via a DI mechanism. However, there may be some cases where this is not feasible, especially when importing the system into an existing application.

To support these scenarios the package Microsoft.AspNetCore.DataProtection.Extensions provides a concrete type DataProtectionProvider which offers a simple way to use the data protection system without going through DI-specific code paths. The type itself implements IDataProtectionProvider, and constructing it is as easy as providing a DirectoryInfo where this provider's cryptographic keys should be stored.

For example:

```
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
        string destFolder = Path.Combine(
            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: ");
        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: CfDJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 */
```

⚠ Warning

By default the DataProtectionProvider concrete type does not encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share, in which case the data protection system cannot automatically deduce an appropriate at-rest key encryption mechanism.

Additionally, the DataProtectionProvider concrete type does not isolate applications by default, so all applications pointed at the same key directory can share payloads as long as their purpose parameters match.

The application developer can address both of these if desired. The DataProtectionProvider constructor accepts an optional configuration callback which can be used to tweak the behaviors of the system. The sample below demonstrates restoring isolation via an explicit call to SetApplicationName, and it also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption instead via a call to 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
        string destFolder = Path.Combine(
            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: ");
        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}");
    }
}
```

 Tip

Instances of the DataProtectionProvider concrete type are expensive to create. If an application maintains multiple instances of this type and if they're all pointing at the same key storage directory, application performance may be degraded. The intended usage is that the application developer instantiate this type once then keep reusing this single reference as much as possible. The DataProtectionProvider type and all IDataProtector instances created from it are thread-safe for multiple callers.

Core cryptography extensibility

▯ Warning

Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor

The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There is generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

Decrypt(ArraySegment ciphertext, ArraySegment additionalAuthenticatedData) : byte[]

Encrypt(ArraySegment plaintext, ArraySegment additionalAuthenticatedData) : byte[]

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentNullException and similar) should be homogenized to CryptographicException.

▯ Note

The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

IAuthenticatedEncryptorDescriptor

The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to create an IAuthenticatedEncryptor instance. Its API is as follows.

CreateEncryptorInstance() : IAuthenticatedEncryptor

ExportToXml() : XmlSerializedDescriptorInfo

Like IAuthenticatedEncryptor, an instance of IAuthenticatedEncryptorDescriptor is assumed to wrap one specific key. This means that for any given IAuthenticatedEncryptorDescriptor instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

```
// we have an IAuthenticatedEncryptorDescriptor instance
   IAuthenticatedEncryptorDescriptor descriptor = ...;

   // 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 = descriptor.CreateEncryptorInstance();
   byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

   // get another encryptor instance and perform an authenticated decryption operation
   var encryptor2 = descriptor.CreateEncryptorInstance();
   byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);


   // the 'roundTripped' and 'plaintext' buffers should be equivalent
```

XML Serialization

The primary difference between IAuthenticatedEncryptor and IAuthenticatedEncryptorDescriptor is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an IAuthenticatedEncryptor whose implementation relies on SymmetricAlgorithm and KeyedHashAlgorithm. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (e.g., it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset.

The descriptor can be serialized via its ExportToXml routine. This routine returns an XmlSerializedDescriptorInfo which contains two properties: the XElement representation of the descriptor and the Type which represents an IAuthenticatedEncryptorDescriptorDeserializer which can be used to resurrect this descriptor given the corresponding XElement.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns "http://schemas.asp.net/2015/03/dataProtection"), value "true".

澁 Tip

There's a helper API for setting this attribute. Call the extension method XElement.MarkAsRequiresEncryption() located in namespace Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel.

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM will not expose the material in plaintext form. Instead, the descriptor might write out the key-wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer

The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an IAuthenticatedEncryptorDescriptor instance from an XElement. It exposes a single method:

ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor

The ImportFromXml method takes the XElement that was returned by IAuthenticatedEncryptorDescriptor.ExportToXml and creates an equivalent of the original IAuthenticatedEncryptorDescriptor.

Types which implement IAuthenticatedEncryptorDescriptorDeserializer should have one of the following two public constructors:

.ctor(IServiceProvider)

.ctor()

 Note

The IServiceProvider passed to the constructor may be null.

IAuthenticatedEncryptorConfiguration

The **IAuthenticatedEncryptorConfiguration** interface represents a type which knows how to create IAuthenticatedEncryptorDescriptor instances. It exposes a single API.

CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor

Think of IAuthenticatedEncryptorConfiguration as the top-level factory. The configuration serves as a template. It wraps algorithmic information (e.g., this configuration produces descriptors with an AES-128-GCM master key), but it is not yet associated with a specific key.

When CreateNewDescriptor is called, fresh key material is created solely for this call, and a new IAuthenticatedEncryptorDescriptor is produced which wraps this key material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to CreateNewDescriptor should never create equivalent IAuthenticatedEncryptorDescriptor instances.

The IAuthenticatedEncryptorConfiguration type serves as the entry point for key creation routines such as automatic key rolling. To change the implementation for all future keys, register a singleton IAuthenticatedEncryptorConfiguration in the service container.

Key management extensibility

> 📝 **Tip**
>
> Read the key management section before reading this section, as it explains some of the fundamental concepts behind these APIs.

> ⚠ **Warning**
>
> Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key

The IKey interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

Activation, creation, and expiration dates

Revocation status

Key identifier (a GUID)

Additionally, IKey exposes a CreateEncryptorInstance method which can be used to create an IAuthenticatedEncryptor instance tied to this key.

> 📝 **Note**
>
> There is no API to retrieve the raw cryptographic material from an IKey instance.

IKeyManager

The IKeyManager interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

Create a new key and persist it to storage.

Get all keys from storage.

Revoke one or more keys and persist the revocation information to storage.

> ⚠ **Warning**
>
> Writing an IKeyManager is a very advanced task, and the majority of developers should not attempt it. Instead, most developers should take advantage of the facilities offered by the XmlKeyManager class.

XmlKeyManager

The XmlKeyManager type is the in-box concrete implementation of IKeyManager. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, XElement).

XmlKeyManager depends on several other components in the course of fulfilling its tasks:

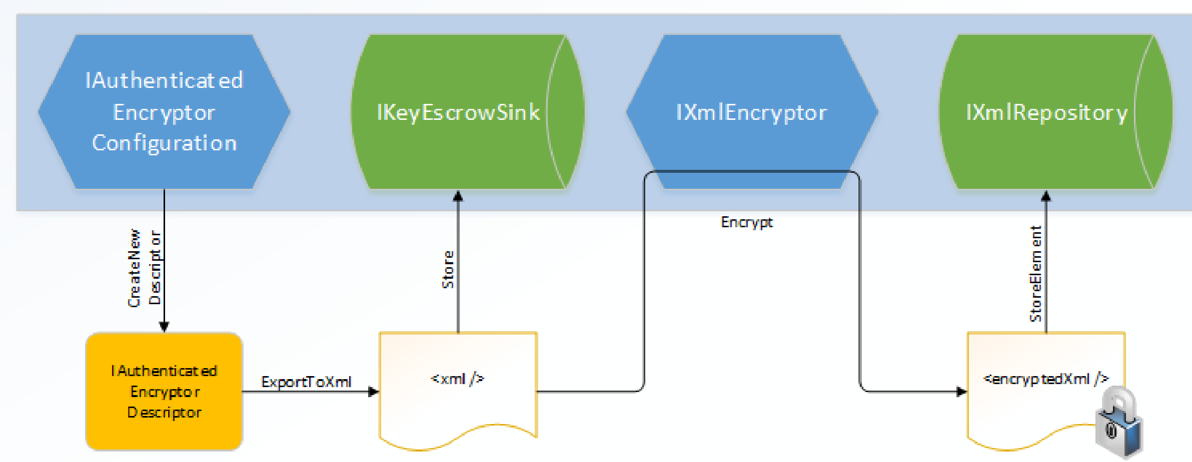
IAuthenticatedEncryptorConfiguration, which dictates the algorithms used by new keys.

IXmlRepository, which controls where keys are persisted in storage.

IXmlEncryptor [optional], which allows encrypting keys at rest.

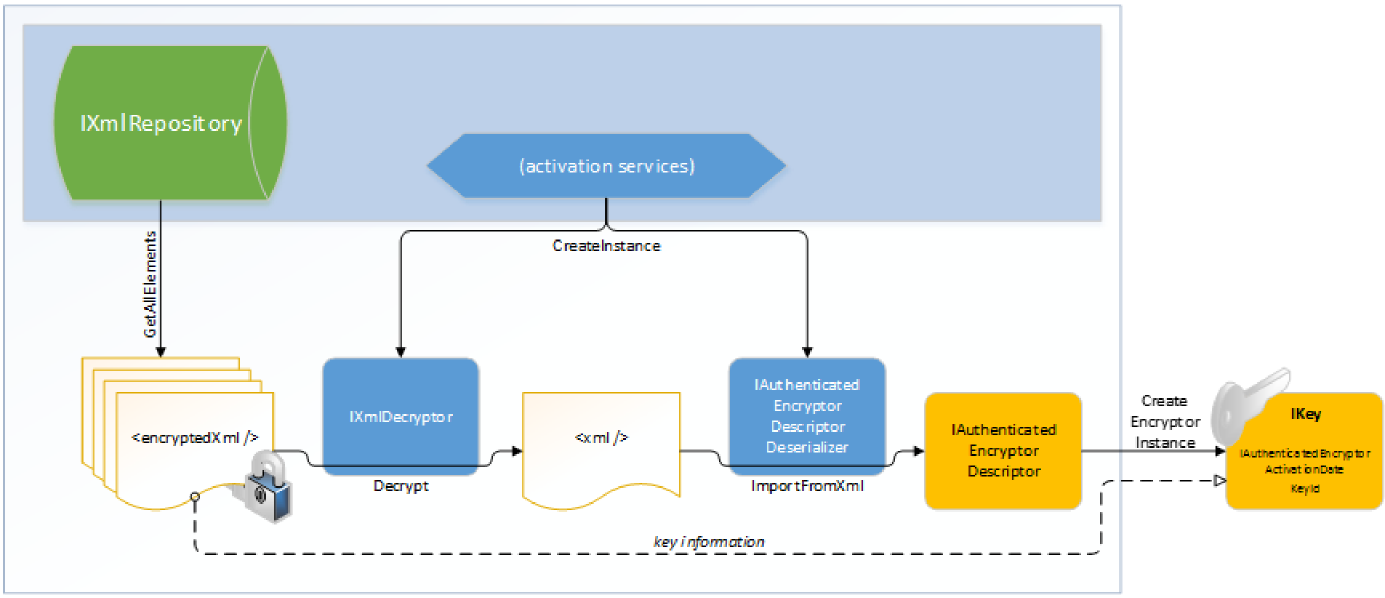
IKeyEscrowSink [optional], which provides key escrow services.

Below are high-level diagrams which indicate how these components are wired together within XmlKeyManager.



Key Creation / CreateNewKey

In the implementation of CreateNewKey, the IAuthenticatedEncryptorConfiguration component is used to create a unique IAuthenticatedEncryptorDescriptor, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an IXmlEncryptor (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the IXmlRepository. (If no IXmlEncryptor is configured, the unencrypted document is persisted in the IXmlRepository.)



Key Retrieval / GetAllKeys

In the implementation of GetAllKeys, the XML documents representing keys and revocations are read from the underlying IXmlRepository. If these documents are encrypted, the system will automatically decrypt them. XmlKeyManager creates the appropriate IAuthenticatedEncryptorDescriptorDeserializer instances to deserialize the documents back into IAuthenticatedEncryptorDescriptor instances, which are then wrapped in individual IKey instances. This collection of IKey instances is returned to the caller.

Further information on the particular XML elements can be found in the key storage format document.

IXmlRepository

The IXmlRepository interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

GetAllElements() : IReadOnlyCollection

StoreElement(XElement element, string friendlyName)

Implementations of IXmlRepository don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are two built-in concrete types which implement IXmlRepository: FileSystemXmlRepository and RegistryXmlRepository. See the key storage providers document for more information. Registering a custom IXmlRepository would be the appropriate manner to use a different backing store, e.g., Azure Blob Storage. To change the default repository application-wide, register a custom singleton IXmlRepository in the service provider.

IXmlEncryptor

The IXmlEncryptor interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

Encrypt(XElement plaintextElement) : EncryptedXmlInfo

If a serialized IAuthenticatedEncryptorDescriptor contains any elements marked as "requires encryption", then XmlKeyManager will run those elements through the configured IXmlEncryptor's Encrypt method, and it will persist the enciphered element rather than the plaintext element to the IXmlRepository. The output of the Encrypt method is an EncryptedXmlInfo object. This object is a wrapper which contains both the resultant enciphered XElement and the Type which represents an IXmlDecryptor which can be used to decipher the corresponding element.

There are four built-in concrete types which implement IXmlEncryptor: CertificateXmlEncryptor, DpapiNGXmlEncryptor, DpapiXmlEncryptor, and NullXmlEncryptor. See the key encryption at rest document for more information. To change the default key-encryption-at-rest mechanism application-wide, register a custom singleton IXmlEncryptor in the service provider.

IXmlDecryptor

The IXmlDecryptor interface represents a type that knows how to decrypt an XElement that was enciphered via an IXmlEncryptor. It exposes a single API:

Decrypt(XElement encryptedElement) : XElement

The Decrypt method undoes the encryption performed by IXmlEncryptor.Encrypt. Generally each concrete IXmlEncryptor implementation will have a corresponding concrete IXmlDecryptor implementation.

Types which implement IXmlDecryptor should have one of the following two public constructors:

.ctor(IServiceProvider)

.ctor()

Note

The IServiceProvider passed to the constructor may be null.

IKeyEscrowSink

The IKeyEscrowSink interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the IXmlEncryptor type in the first place. However, accidents happen, and keyrings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it is transformed by any configured IXmlEncryptor. The interface exposes a single API:

Store(Guid keyId, XElement element)

It is up to the IKeyEscrowSink implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate's private key has been escrowed; the CertificateXmlEncryptor type can assist with this. The IKeyEscrowSink implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can configure this globally. It can also be configured programmatically via the *IDataProtectionBuilder.AddKeyEscrowSink* method as shown in the sample below. The *AddKeyEscrowSink* method overloads mirror the *IServiceCollection.AddSingleton* and *IServiceCollection.AddInstance* overloads, as IKeyEscrowSink instances are intended to be singletons. If multiple IKeyEscrowSink instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There is no API to read material from an IKeyEscrowSink instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an IKeyEscrowSink where keys are escrowed such that only members of "CONTOSODomain Admins" can recover them.

 Note

To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

```
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/en-us/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>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

Miscellaneous APIs

 Warning

Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret

The ISecret interface represents a secret value, such as cryptographic key material. It contains the following API surface.

Length : int

Dispose() : void

WriteSecretIntoBuffer(ArraySegment buffer) : void

The WriteSecretIntoBuffer method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a byte[] directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The Secret type is a concrete implementation of ISecret where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via CryptProtectMemory.

Authenticated encryption details.

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it is tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There is also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be base64url-encoded.)

Protected payload format

The protected payload format consists of three primary components:

A 32-bit magic header that identifies the version of the data protection system.

A 128-bit key id that identifies the key used to protect this particular payload.

The remainder of the protected payload is specific to the encryptor encapsulated by this key. In the example below the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows: * A 128-bit key modifier. * A 128-bit initialization vector. * 48 bytes of AES-256-CBC output. * An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
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
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

 Warning

All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {0c819c80-6619-4019-9536-53f8aaffee57}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it is reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey Derivation and Authenticated Encryption

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

▯ Note

Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of IAuthenticatedEncryptor rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation

The IAuthenticatedEncryptor interface serves as the core interface for all authenticated encryption operations. Its Encrypt method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to IDataProtector.Protect, but the AAD is generated by the system and consists of three components:

The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.

The 128-bit key id.

A variable-length string formed from the purpose chain that created the IDataProtector that is performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to IAuthenticatedEncryptor.Encrypt, the following key derivation process takes place:

(K_E, K_H) = SP800_108_CTR_HMACSHA512(K_M, AAD, contextHeader || keyModifier)

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with the following parameters:

Key derivation key (KDK) = K_M

PRF = HMACSHA512

label = additionalAuthenticatedData

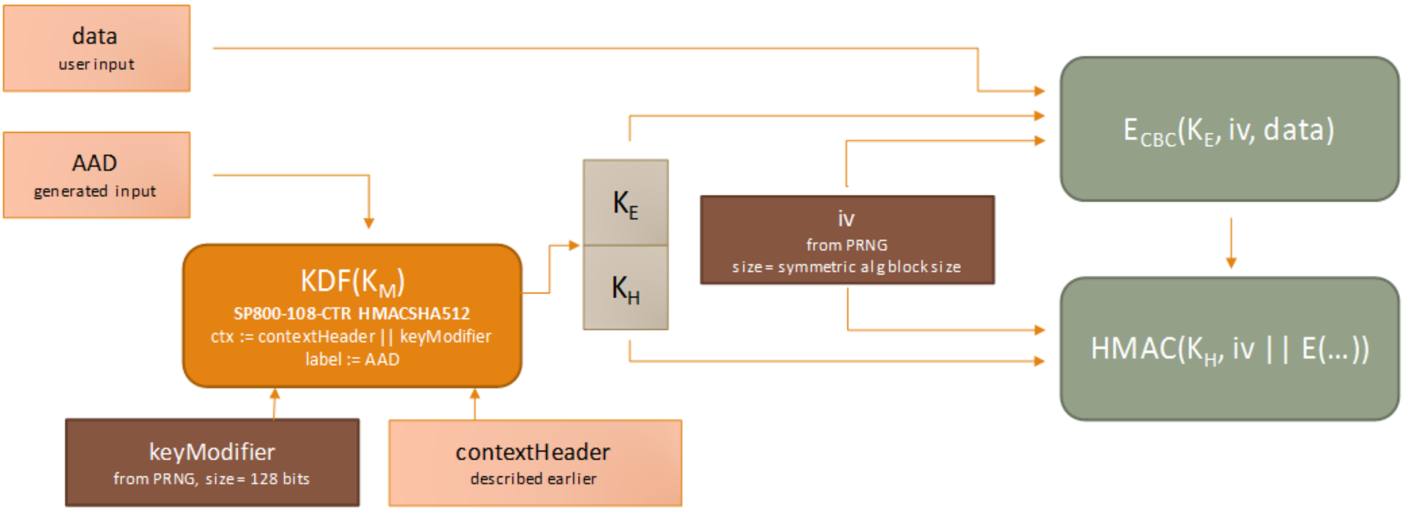
context = contextHeader || keyModifier

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving K_E and K_H. The key modifier is a 128-bit string randomly generated for each call to Encrypt and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, | K_E | is the length of the symmetric block cipher key, and | K_H | is the digest size of the HMAC routine. For GCM encryption + validation operations, | K_H | = 0.

CBC-mode encryption + HMAC validation

Once K_E is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key K_H to produce the MAC. This process and the return value is represented graphically below.



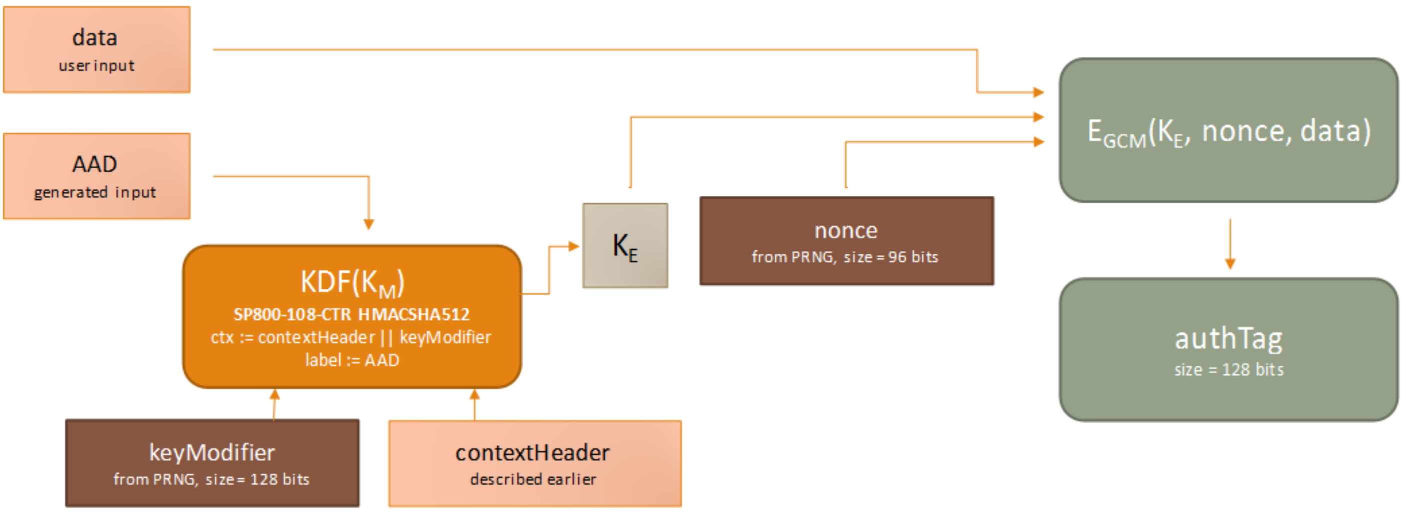
output:= keyModifier || iv || E_cbc (K_E,iv,data) || HMAC(K_H, iv || E_cbc (K_E,iv,data))

Note

The IDataProtector.Protect implementation will prepend the magic header and key id to output before returning it to the caller. Because the magic header and key id are implicitly part of AAD, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation

Once K_E is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.



output := keyModifier || nonce || E_gcm (K_E,nonce,data) || authTag

Note

Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, to support agility we never want to use K_M directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^32. If we fix K_E we cannot perform more than 2^32 encryption operations before we run afoul of the 2^-32 limit. This might seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^-32 probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given K_M. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there is no need to forward it to the GCM routine.

Context headers

Background and theory

In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It is intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of cryptographic agility, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), they'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the subkey derivation process. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication

The context header consists of the following components:

[16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".

[32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.

[32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.

[32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)

[32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.

EncCBC(K_E, IV, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of K_E is described below.

MAC(K_H, ""), which is the output of the HMAC algorithm given an empty string input. The construction of K_H is described below.

Ideally we could pass all-zero vectors for K_E and K_H. However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive | K_E | + | K_H | bytes of output, then decompose the result into K_E and K_H themselves. Mathematically, this is represented as follows.

(K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

Example: AES-192-CBC + HMACSHA256

As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let (K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 192 bits and | K_H | = 256 bits per the specified algorithms. This leads to K_E = 5BB6..21DD and K_H = A04A..00A9 in the example below:

```
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
```

Next, compute Enc_CBC (K_E, IV, "") for AES-192-CBC given IV = 0* and K_E as above.

result := F474B1872B3B53E4721DE19C0841DB6F

Next, compute MAC(K_H, "") for HMACSHA256 given K_H as above.

result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C

This produces the full context header below:

```
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
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described above are:

the marker (00 00)

the block cipher key length (00 00 00 18)

the block cipher block size (00 00 00 10)

the HMAC key length (00 00 00 20)

the HMAC digest size (00 00 00 20)

the block cipher PRP output (F4 74 - DB 6F) and

the HMAC PRF output (D4 79 - end).

Note

The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1

First, let (K_E || K_H) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 192 bits and | K_H | = 160 bits per the specified algorithms. This leads to K_E = A219..E2BB and K_H = DC4A..B464 in the example below:

```
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
```

Next, compute Enc_CBC (K_E, IV, "") for 3DES-192-CBC given IV = 0* and K_E as above.

result := ABB100F81E53E10E

Next, compute MAC(K_H, "") for HMACSHA1 given K_H as above.

result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
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
```

The components break down as follows:

the marker (00 00)

the block cipher key length (00 00 00 18)

the block cipher block size (00 00 00 08)

the HMAC key length (00 00 00 14)

the HMAC digest size (00 00 00 14)

the block cipher PRP output (AB B1 - E1 0E) and

the HMAC PRF output (76 EB - end).

Galois/Counter Mode encryption + authentication

The context header consists of the following components:

[16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".

[32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.

[32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)

[32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)

[32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)

[128 bits] The tag of Enc_GCM (K_E, nonce, ""), which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

K_E is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there is no K_H in play here, we essentially have | K_H | = 0, and the algorithm collapses to the below form.

K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")

Example: AES-256-GCM

First, let K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = ""), where | K_E | = 256 bits.

K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8

Next, compute the authentication tag of Enc_GCM (K_E, nonce, "") for AES-256-GCM given nonce = 096 and K_E as above.

result := E7DCCE66DF855A323A6BB7BD7A59BE45

This produces the full context header below:

```
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
```

The components break down as follows:

the marker (00 01)

the block cipher key length (00 00 00 20)

the nonce size (00 00 00 0C)

the block cipher block size (00 00 00 10)

the authentication tag size (00 00 00 10) and

the authentication tag from running the block cipher (E7 DC - end).

Key Management

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages.

Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.

Active - the key exists in the key ring and should be used for all new Protect operations.

Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.

Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can override this behavior if necessary.

> **Warning**
>
> The developer might be tempted to delete a key from the key ring (e.g., by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there is no emergency override like there is with revoked keys. Deleting a key is truly destructive behavior, and consequently the data protection system exposes no first-class API for performing this operation.

Default key selection

When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has not disabled automatic key generation, then a new key will be generated with immediate activation per the key expiration and rolling policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There is an exception. If the application developer has disabled automatic key generation, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling

When a key is created, it is automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring does not already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

```
services.AddDataProtection()
       // use 14-day lifetime instead of 90-day lifetime
       .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to SetDefaultKeyLifetime will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic keyring refresh

When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

⚠ Warning

Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface IKeyManager that can be used to inspect and make changes to the key ring. The DI system that provided the instance of IDataProtectionProvider can also provide an instance of IKeyManager for your consumption. Alternatively, you can pull the IKeyManager straight from the IServiceProvider as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to Protect or Unprotect will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the IKeyManager interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

```
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:B}: 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:B}: 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
 */
```

Key storage

The data protection system has a heuristic whereby it tries to deduce an appropriate key storage location and encryption at rest mechanism automatically. This is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

In-box key storage providers

In-box key encryption at rest providers

Key storage providers

By default the data protection system employs a heuristic to determine where cryptographic key material should be persisted. The developer can override the heuristic and manually specify the location.

Note

If you specify an explicit key persistence location, the data protection system will deregister the default key encryption at rest mechanism that the heuristic provided, so keys will no longer be encrypted at rest. It is recommended that you additionally specify an explicit key encryption mechanism for production applications.

The data protection system ships with several in-box key storage providers.

File system

We anticipate that many apps will use a file system-based key repository. To configure this, call the PersistKeysToFileSystem configuration routine as shown below. Provide a `DirectoryInfo` pointing to the repository where keys should be stored.

```
sc.AddDataProtection()
       // persist keys to a specific directory
       .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only applications on the local machine will need to use this repository), consider using Windows DPAPI to encrypt the keys at rest. Otherwise consider using an X.509 certificate to encrypt keys at rest.

Azure and Redis

The `Microsoft.AspNetCore.DataProtection.AzureStorage` and `Microsoft.AspNetCore.DataProtection.Redis` packages allow storing your data protection keys in Azure Storage or a Redis cache. Keys can be shared across several instances of a web app. Your ASP.NET Core app can share authentication cookies or CSRF protection across multiple servers. To configure on Azure, call one of the PersistKeysToAzureBlobStorage overloads as shown below.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS token>"));

    services.AddMvc();
}
```

See also the Azure test code.

To configure on Redis, call one of the PersistKeysToRedis overloads as shown below.

```
public void ConfigureServices(IServiceCollection services)
{
    // Connect to Redis database.
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToRedis(redis, "DataProtection-Keys");

    services.AddMvc();
}
```

See the following for more information:

StackExchange.Redis ConnectionMultiplexer

Azure Redis Cache
Redis test code.

Registry

Sometimes the app might not have write access to the file system. Consider a scenario where an app is running as a virtual service account (such as w3wp.exe's app pool identity). In these cases, the administrator may have provisioned a registry key that is appropriate ACLed for the service account identity. Call the PersistKeysToRegistry configuration routine as shown below. Provide a `RegistryKey` pointing to the location where cryptographic keys/values should be stored.

```
sc.AddDataProtection()
    // persist keys to a specific location in the system registry
    .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
```

If you use the system registry as a persistence mechanism, consider using Windows DPAPI to encrypt the keys at rest.

Custom key repository

If the in-box mechanisms are not appropriate, the developer can specify their own key persistence mechanism by providing a custom `IXmlRepository`.

Key Encryption At Rest

By default the data protection system employs a heuristic to determine how cryptographic key material should be encrypted at rest. The developer can override the heuristic and manually specify how keys should be encrypted at rest.

 Note

If you specify an explicit key encryption at rest mechanism, the data protection system will deregister the default key storage mechanism that the heuristic provided. You must specify an explicit key storage mechanism, otherwise the data protection system will fail to start.

The data protection system ships with three in-box key encryption mechanisms.

Windows DPAPI

This mechanism is available only on Windows.

When Windows DPAPI is used, key material will be encrypted via CryptProtectData before being persisted to storage. DPAPI is an appropriate encryption mechanism for data that will never be read outside of the current machine (though it is possible to back these keys up to Active Directory; see DPAPI and Roaming Profiles). For example to configure DPAPI key-at-rest encryption.

```
sc.AddDataProtection()
       // only the local user account can decrypt the keys
       .ProtectKeysWithDpapi();
```

If ProtectKeysWithDpapi is called with no parameters, only the current Windows user account can decipher the persisted key material. You can optionally specify that any user account on the machine (not just the current user account) should be able to decipher the key material, as shown in the below example.

```
sc.AddDataProtection()
       // all user accounts on the machine can decrypt the keys
       .ProtectKeysWithDpapi(protectToLocalMachine: true);
```

X.509 certificate

This mechanism is not available on `.NET Core 1.0` or `1.1`.

If your application is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and to configure applications to use this certificate for encryption of keys at rest. See below for an example.

```
sc.AddDataProtection()
       // searches the cert store for the cert with this thumbprint
       .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
```

Due to .NET Framework limitations only certificates with CAPI private keys are supported. See Certificate-based encryption with Windows DPAPI-NG below for possible workarounds to these limitations.

Windows DPAPI-NG

This mechanism is available only on Windows 8 / Windows Server 2012 and later.

Beginning with Windows 8, the operating system supports DPAPI-NG (also called CNG DPAPI). Microsoft lays out its usage scenario as follows.

Cloud computing, however, often requires that content encrypted on one computer be decrypted on another. Therefore, beginning with Windows 8, Microsoft extended the idea of using a relatively straightforward API to encompass cloud scenarios. This new API, called DPAPI-NG, enables you to securely share secrets (keys, passwords, key material) and messages by protecting them to a set of principals that can be used to unprotect them on different computers after proper authentication and authorization.

From https://msdn.microsoft.com/en-us/library/windows/desktop/hh706794(v=vs.85).aspx

The principal is encoded as a protection descriptor rule. Consider the below example, which encrypts key material such that only the domain-joined user with the specified SID can decrypt the key material.

```
sc.AddDataProtection()
     // uses the descriptor rule "SID=S-1-5-21-..."
     .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
       flags: DpapiNGProtectionDescriptorFlags.None);
```

There is also a parameterless overload of ProtectKeysWithDpapiNG. This is a convenience method for specifying the rule "SID=mine", where mine is the SID of the current Windows user account.

```
sc.AddDataProtection()
     // uses the descriptor rule "SID={current account SID}"
     .ProtectKeysWithDpapiNG();
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user will be able to decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG

If you're running on Windows 8.1 / Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption, even if the application is running on .NET Core. To take advantage of this, use the rule descriptor string "CERTIFICATE=HashId:thumbprint", where thumbprint is the hex-encoded SHA1 thumbprint of the certificate to use. See below for an example.

```
sc.AddDataProtection()
       // searches the cert store for the cert with this thumbprint
       .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0",
           flags: DpapiNGProtectionDescriptorFlags.None);
```

Any application which is pointed at this repository must be running on Windows 8.1 / Windows Server 2012 R2 or later to be able to decipher this key.

Custom key encryption

If the in-box mechanisms are not appropriate, the developer can specify their own key encryption mechanism by providing a custom IXmlEncryptor.

Key Immutability and changing settings

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it is immutable. Its creation, activation, and expiration dates can never be changed, though it can revoked by using IKeyManager. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes will not go into effect until the next time a key is generated, either via an explicit call to IKeyManager.CreateNewKey or via the data protection system's own automatic key generation behavior. The settings that affect key persistence are as follows:

The default key lifetime

The key encryption at rest mechanism

The algorithmic information contained within the key

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to IKeyManager.CreateNewKey to force the creation of a new key. Remember to provide an explicit activation date ({ now + 2 days } is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

 Tip

All applications touching the repository should specify the same settings with the IDataProtectionBuilder extension methods, otherwise the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key Storage Format

Objects are stored at rest in XML representation. The default directory for key storage is %LOCALAPPDATA%\ASP.NET\DataProtection-Keys.

The <key> element

Keys exist as top-level objects in the key repository. By convention keys have the filename **key-{guid}.xml**, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

```
<?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>
```

The <key> element contains the following attributes and child elements:

The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.

The version of the <key> element, currently fixed at 1.

The key's creation, activation, and expiration dates.

A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {80732141-ec8f-4b80-af9c-c4d2d1ff8901}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element

The outer <descriptor> element contains an attribute deserializerType, which is the assembly-qualified name of a type which implements IAuthenticatedEncryptorDescriptorDeserializer. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element

An element which contains the encrypted form of the secret key material may be present if encryption of secrets at rest is enabled. The attribute decryptorType will be the assembly-qualified name of a type which implements IXmlDecryptor. This type is responsible for reading the inner element and decrypting it to recover the original plaintext.

As with <descriptor>, the particular format of the element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element

Revocations exist as top-level objects in the key repository. By convention revocations have the filename **revocation-{timestamp}.xml** (for revoking all keys before a specific date) or **revocation-{guid}.xml** (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```
<?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>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

```
<?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>
```

The <reason> element is never read by the system. It is simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers

There are scenarios where an application needs a throwaway IDataProtectionProvider. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the package Microsoft.AspNetCore.DataProtection includes a type EphemeralDataProtectionProvider. This type provides a basic implementation of IDataProtectionProvider whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of EphemeralDataProtectionProvider uses its own unique master key. Therefore, if an IDataProtector rooted at an EphemeralDataProtectionProvider generates a protected payload, that payload can only be unprotected by an equivalent IDataProtector (given the same purpose chain) rooted at the same EphemeralDataProtectionProvider instance.

The following sample demonstrates instantiating an EphemeralDataProtectionProvider and using it to protect and unprotect data.

```
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: CfDJ8AAAAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
    * Unprotect returned: Hello!
    * << throws CryptographicException >>
    */
```

Sharing cookies between applications

Web sites commonly consist of many individual web applications, all working together harmoniously. If an application developer wants to provide a good single-sign-on experience, they'll often need all of the different web applications within the site to share authentication tickets between each other.

To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

Sharing authentication cookies between applications

To share authentication cookies between two different ASP.NET Core applications, configure each application that should share cookies as follows.

In your configure method use the CookieAuthenticationOptions to set up the data protection service for cookies and the AuthenticationScheme to match ASP.NET 4.X.

If you're using identity:

```
app.AddIdentity<ApplicationUser, IdentityRole>(options =>
   {
       options.Cookies.ApplicationCookie.AuthenticationScheme = "ApplicationCookie";
       options.Cookies.ApplicationCookie.DataProtectionProvider = DataProtectionProvider.Create(new
DirectoryInfo(@"c:\shared-auth-ticket-keys\"));
   });
```

If you're using cookies directly:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
   {
       DataProtectionProvider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-
keys\"))
   });
```

The `DataProtectionProvider` requires the `Microsoft.AspNetCore.DataProtection.Extensions` NuGet package.

When used in this manner, the DirectoryInfo should point to a key storage location specifically set aside for authentication cookies. The cookie authentication middleware will use the explicitly provided implementation of the DataProtectionProvider, which is now isolated from the data protection system used by other parts of the application. The application name is ignored (intentionally so, since you're trying to get multiple applications to share payloads).

⚠ Warning

You should consider configuring the DataProtectionProvider such that keys are encrypted at rest, as in the below example.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    DataProtectionProvider = DataProtectionProvider.Create(
        new DirectoryInfo(@"c:\shared-auth-ticket-keys\"),
        configure =>
        {
            configure.ProtectKeysWithCertificate("thumbprint");
        })
});
```

Sharing authentication cookies between ASP.NET 4.x and ASP.NET Core applications

ASP.NET 4.x applications which use Katana cookie authentication middleware can be configured to generate authentication cookies which are compatible with the ASP.NET Core cookie authentication middleware. This allows upgrading a large site's individual applications piecemeal while still providing a smooth single sign on experience across the site.

 Tip

You can tell if your existing application uses Katana cookie authentication middleware by the existence of a call to UseCookieAuthentication in your project's Startup.Auth.cs. ASP.NET 4.x web application projects created with Visual Studio 2013 and later use the Katana cookie authentication middleware by default.

 Note

Your ASP.NET 4.x application must target .NET Framework 4.5.1 or higher, otherwise the necessary NuGet packages will fail to install.

To share authentication cookies between your ASP.NET 4.x applications and your ASP.NET Core applications, configure the ASP.NET Core application as stated above, then configure your ASP.NET 4.x applications by following the steps below.

Install the package Microsoft.Owin.Security.Interop into each of your ASP.NET 4.x applications.

In Startup.Auth.cs, locate the call to UseCookieAuthentication, which will generally look like the following.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
   // ...
});
```

Modify the call to UseCookieAuthentication as follows, changing the CookieName to match the name used by the ASP.NET Core cookie authentication middleware, providing an instance of a DataProtectionProvider that has been initialized to a key storage location, and set CookieManager to interop ChunkingCookieManager so the chunking format is compatible.

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
   {
       AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
       CookieName = ".AspNetCore.Cookies",
       // CookiePath = "...", (if necessary)
       // ...
       TicketDataFormat = new AspNetTicketDataFormat(
           new DataProtectorShim(
               DataProtectionProvider.Create(new DirectoryInfo(@"c:\shared-auth-ticket-keys\"))
               .CreateProtector("Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware",
               "Cookies", "v2"))),
       CookieManager = new ChunkingCookieManager()
   });
```

The DirectoryInfo has to point to the same storage location that you pointed your ASP.NET Core application to and should be configured using the same settings.

The ASP.NET 4.x and ASP.NET Core applications are now configured to share authentication cookies.

 Note

You'll need to make sure that the identity system for each application is pointed at the same user database. Otherwise the identity system will produce failures at runtime when it tries to match the information in the authentication cookie against the information in its database.

Replacing `<machineKey>` in ASP.NET

The implementation of the `<machineKey>` element in ASP.NET is replaceable. This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

 Note

The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or higher. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNetCore.DataProtection.SystemWeb. This will instantiate the data protection system using the default configuration settings.

When you install the package, it inserts a line into *Web.config* that tells ASP.NET to use it for most cryptographic operations, including forms authentication, view state, and calls to MachineKey.Protect. The line that's inserted reads as follows.

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

 Tip

You can tell if the new data protection system is active by inspecting fields like `__VIEWSTATE`, which should begin with "CfDJ8" as in the example below. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZOpk..." />
```

Package configuration

The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses DataProtectionStartup and overrides its ConfigureServices method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

```
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");
        }
    }
}
```

 Tip

You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to SetApplicationName. This is a convenience mechanism to avoid forcing the developer to create a DataProtectionStartup-derived type if all they wanted to configure was setting the application name.

To enable this custom configuration, go back to Web.config and look for the `<appSettings>` element that the package install added to the config file. It will look like the following markup:

```
<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>
```

Fill in the blank value with the assembly-qualified name of the DataProtectionStartup-derived type you just created. If the name of the application is DataProtectionDemo, this would look like the below.

```
<add key="aspnet:dataProtectionStartupType"
     value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

The newly-configured data protection system is now ready for use inside the application.

Enforcing SSL in an ASP.NET Core app

This document shows how to:

Require SSL for all requests (HTTPS requests only).
Redirect all HTTP requests to HTTPS.
Set up IIS Express to use SSL/HTTPS.

Require SSL

The RequireHttpsAttribute is used to require SSL. You can decorate controllers or methods with this attribute or you can apply it globally as shown below:

Add the following code to `ConfigureServices` in `Startup`:

```
// Requires using Microsoft.AspNetCore.Mvc;
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new RequireHttpsAttribute());
    });
```

The highlighted code above requires all requests use `HTTPS`, therefore HTTP requests are ignored. The following highlighted code redirects all HTTP requests to HTTPS:

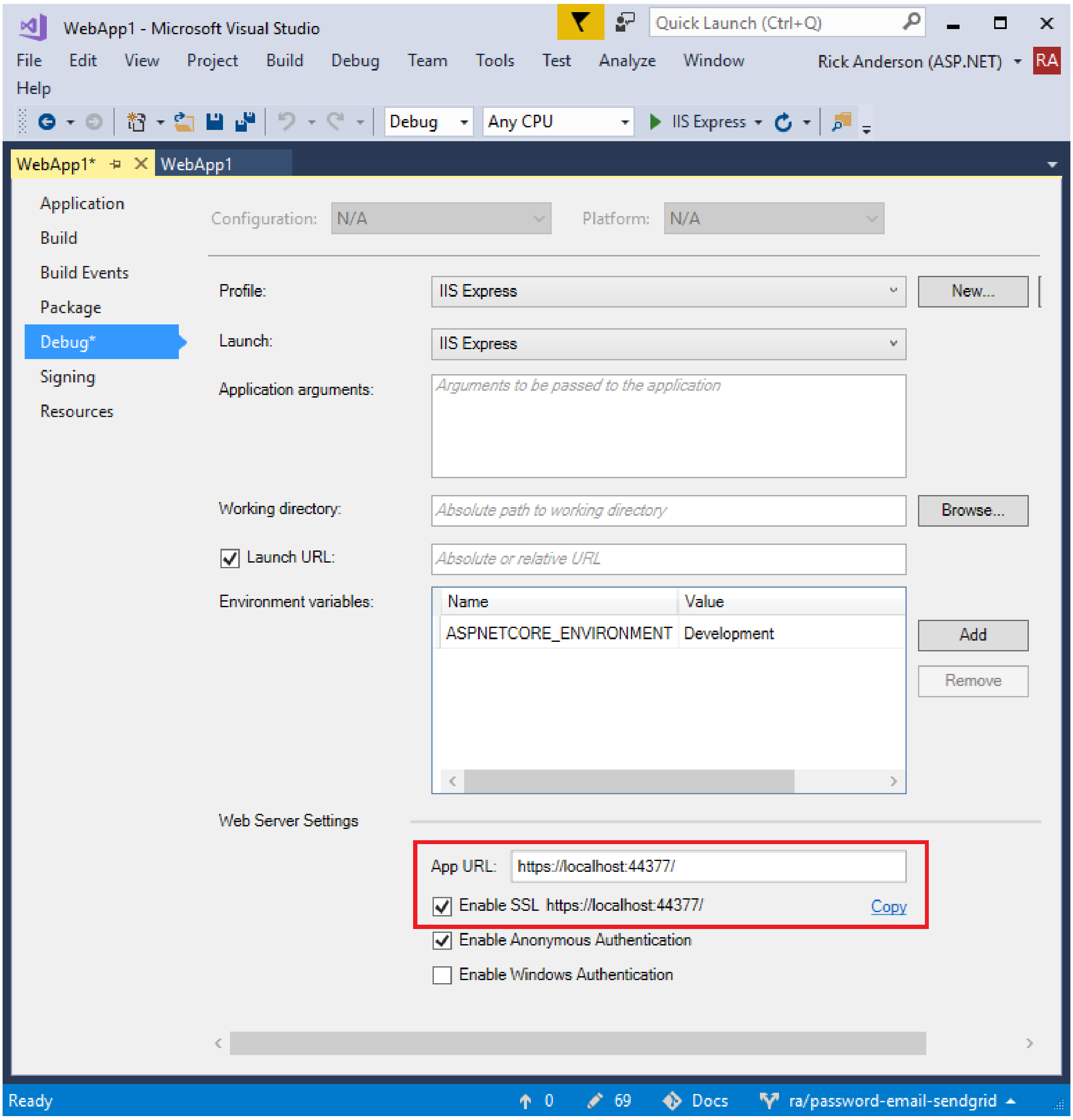
```
// Requires using Microsoft.AspNetCore.Rewrite;
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    var options = new RewriteOptions()
       .AddRedirectToHttps();

```

See URL Rewriting Middleware for more information.

Requiring HTTPS globally (`options.Filters.Add(new RequireHttpsAttribute());`) is a security best practice. Applying the `[RequireHttps]` to controllers has the drawback that you're not guaranteed new controllers added to you project will get this protection.

Set up IIS Express for SSL/HTTPS

In Solution Explorer, right click the project and select **Properties**.
On the left pane, select **Debug**.
Check **Enable SSL**
Copy the SSL URL and paste it into the **App URL**



WebApp1 - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window
Help
Rick Anderson (ASP.NET)
RA
Debug
Any CPU
IIS Express
WebApp1*
WebApp1
Application
Build
Build Events
Package
Debug*
Signing
Resources
Configuration: N/A
Platform: N/A
Profile: IIS Express
New...
Launch: IIS Express
Application arguments:
Arguments to be passed to the application
Working directory:
Absolute path to working directory
Browse...
Launch URL:
Absolute or relative URL
Environment variables:
Name
Value
ASPNETCORE_ENVIRONMENT
Development
Add
Remove
Web Server Settings
App URL: https://localhost:44377/
Enable SSL https://localhost:44377/
Copy
Enable Anonymous Authentication
Enable Windows Authentication
Ready
0
69
Docs
ra/password-email-sendgrid

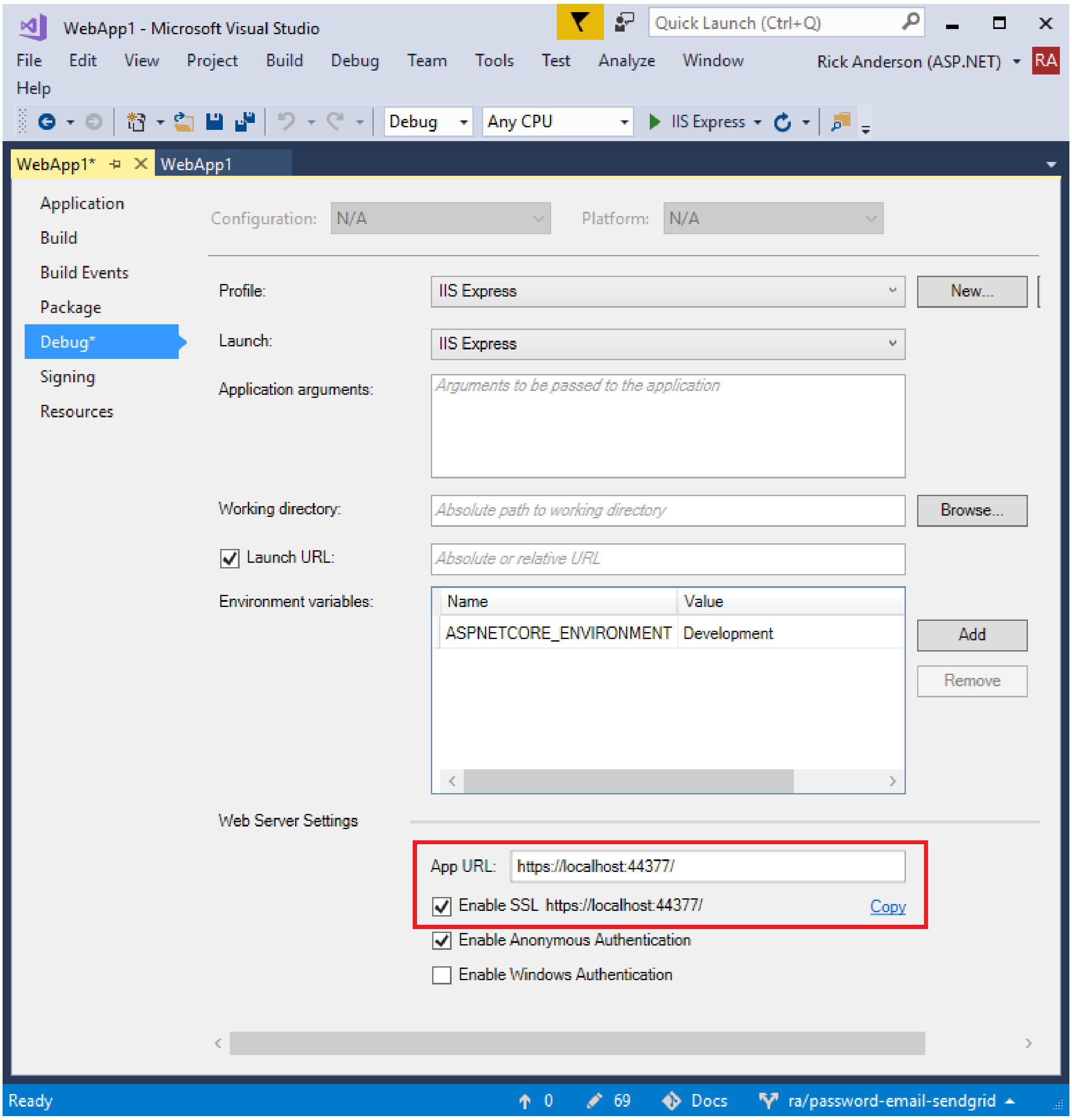
Setting up HTTPS for development in ASP.NET Core

Note

This topic applies to ASP.NET Core 2.0 Preview 1

You can configure your application to use HTTPS during development to simulate HTTPS in your production environment. Enabling HTTPS may be required to enable integration with various identity providers (like Azure AD and Azure AD B2C).

On Windows if you've installed Visual Studio or IIS Express, the IIS Express Development Certificate will be in your LocalMachine certificate store. You can update your project properties in Visual Studio to use this certificate when running behind IIS Express.



You can also configure Kestrel to listen over HTTPS by configuring an endpoint with the desired IP address, port, and certificate. The certificate can be configured inline, or in the top level `Certificates` section and then referenced by name:

```
{
  "Kestrel": {
    "Endpoints": {
      "LocalhostHttps": {
        "Address": "127.0.0.1",
        "Port": "43434",
        "Certificate": "HTTPS"
      }
    }
  }
}
```

For development you can use the IIS Express Development Certificate if it is available, or create a new certificate for development purposes. The development certificate should be configured in the `appsettings.Development.json` file so that it is not used in production:

```
{
  "Certificates": {
    "HTTPS": {
      "Source": "Store",
      "StoreLocation": "LocalMachine",
      "StoreName": "My",
      "Subject": "CN=localhost",
      "AllowInvalid": true
    }
  }
}
```

An app with this configuration running in production will throw an exception saying "No certificate named 'HTTPS' found in configuration for the current environment (Production)". To switch the environment to `Development`, set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development`.

If you do not have the IIS Express Development Certificate installed, you can create a development certificate yourself. On Windows you can create a development certificate and add it to the trusted root store for the current user by running the following PowerShell commands in an elevated prompt:

```
$cert = New-SelfSignedCertificate -Subject localhost -DnsName localhost -FriendlyName "ASP.NET Core Development" -KeyUsage DigitalSignature -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.3.1")
Export-Certificate -Cert $cert -FilePath cert.cer
Import-Certificate -FilePath cert.cer -CertStoreLocation cert:/CurrentUser/Root
```

On macOS and Linux you can create a self-signed certificate for HTTPS using OpenSSL:

```
openssl req -new -x509 -newkey rsa:2048 -keyout localhost.key -out localhost.cer -days 365 -subj /CN=localhost
openssl pkcs12 -export -out certificate.pfx -inkey localhost.key -in localhost.cer
```

Once the `certificate.pfx` file has been generated, configure the HTTPS certificate in your `appsettings.Development.json` file:

```
{
  "Certificates": {
    "HTTPS": {
      "Source": "File",
      "Path": "certificate.pfx"
    }
  }
}
```

You will also need to specify the passphrase for the certificate by setting the “Certificates:HTTPS:Password” config property. Passwords should not be stored in plain text. See Safe Storage of App Secrets During Development for appropriate handling of the certificate passphrase.

On macOS you can add the certificate to your keychain and change its trust settings so that it is trusted for HTTPS during development. To add the certificate to your keychain (the equivalent of the `CurrentUser/My` store on Windows) run the following command:

```
security import certificate.pfx -k ~/Library/Keychains/login.keychain-db
```

And then to trust the certificate:

```
security add-trusted-cert localhost.cer
```

You can then configure your app to use this certificate in development like this:

```
{
  "Certificates": {
    "HTTPS": {
      "Source": "Store",
      "StoreLocation": "CurrentUser",
      "StoreName": "My",
      "Subject": "CN=localhost",
      "AllowInvalid": true
    }
  }
}
```

Safe storage of app secrets during development

By Rick Anderson, Daniel Roth, and Scott Addie

This document shows how you can use the Secret Manager tool in development to keep secrets out of your code. The most important point is you should never store passwords or other sensitive data in source code, and you shouldn't use production secrets in development and test mode. You can instead use the configuration system to read these values from environment variables or from values stored using the Secret Manager tool. The Secret Manager tool helps prevent sensitive data from being checked into source control. The configuration system can read secrets stored with the Secret Manager tool described in this article.

The Secret Manager tool is used only in development. You can safeguard Azure test and production secrets with the Microsoft Azure Key Vault configuration provider. See Azure Key Vault configuration provider for more information.

Environment variables

To avoid storing app secrets in code or in local configuration files, you store secrets in environment variables. You can setup the configuration framework to read values from environment variables by calling `AddEnvironmentVariables`. You can then use environment variables to override configuration values for all previously specified configuration sources.

For example, if you create a new ASP.NET Core web app with individual user accounts, it will add a default connection string to the *appsettings.json* file in the project with the key `DefaultConnection`. The default connection string is setup to use LocalDB, which runs in user mode and doesn't require a password. When you deploy your application to a test or production server, you can override the `DefaultConnection` key value with an environment variable setting that contains the connection string (potentially with sensitive credentials) for a test or production database server.

> Warning
>
> Environment variables are generally stored in plain text and are not encrypted. If the machine or process is compromised, then environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may still be required.

Secret Manager

The Secret Manager tool stores sensitive data for development work outside of your project tree. The Secret Manager tool is a project tool that can be used to store secrets for a .NET Core project during development. With the Secret Manager tool, you can associate app secrets with a specific project and share them across multiple projects.

> Warning
>
> The Secret Manager tool does not encrypt the stored secrets and should not be treated as a trusted store. It is for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

Visual Studio 2017: Installing the Secret Manager tool

Right-click the project in Solution Explorer, and select **Edit <project_name>.csproj** from the context menu. Add the highlighted line to the *.csproj* file, and save to restore the associated NuGet package:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <PropertyGroup>
    <UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.0-msbuild3-final"
/>
  </ItemGroup>
</Project>
```

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new `UserSecretsId` node within a `PropertyGroup` of the *.csproj* file. It also opens a `secrets.json` file in the text editor.

Add the following to `secrets.json`:

```json
{
    "MySecret": "ValueOfMySecret"
}
```

Visual Studio 2015: Installing the Secret Manager tool

Open the project's `project.json` file. Add a reference to `Microsoft.Extensions.SecretManager.Tools` within the `tools` property, and save to restore the associated NuGet package:

```json
"tools": {
    "Microsoft.Extensions.SecretManager.Tools": "1.0.0-preview2-final",
    "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
},
```

Right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a new `userSecretsId` property to `project.json`. It also opens a `secrets.json` file in the text editor.

Add the following to `secrets.json`:

```json
{
    "MySecret": "ValueOfMySecret"
}
```

Visual Studio Code or Command Line: Installing the Secret Manager tool

Add `Microsoft.Extensions.SecretManager.Tools` to the *.csproj* file and run `dotnet restore`.

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <PropertyGroup>
    <UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Configuration.UserSecrets" Version="1.1.1" />
  </ItemGroup>

  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="1.0.0-msbuild3-final"
/>
  </ItemGroup>
</Project>
```

Test the Secret Manager tool by running the following command:

```
dotnet user-secrets -h
```

The Secret Manager tool will display usage, options and command help.

Note

You must be in the same directory as the *.csproj* file to run tools defined in the *.csproj* file's `DotNetCliToolReference` nodes.

The Secret Manager tool operates on project-specific configuration settings that are stored in your user profile. To use user secrets, the project must specify a `UserSecretsId` value in its *.csproj* file. The value of `UserSecretsId` is arbitrary, but is generally unique to the project. Developers typically generate a GUID for the `UserSecretsId`.

Add a `UserSecretsId` for your project in the *.csproj* file:

```
<PropertyGroup>
  <UserSecretsId>My-USER-SECRET-ID-HERE-c23d27a4-eb88</UserSecretsId>
</PropertyGroup>
```

Use the Secret Manager tool to set a secret. For example, in a command window from the project directory, enter the following:

```
dotnet user-secrets set MySecret ValueOfMySecret
```

You can run the Secret Manager tool from other directories, but you must use the `--project` option to pass in the path to the *.csproj* file:

```
dotnet user-secrets set MySecret ValueOfMySecret --project c:\work\WebApp1\src\webapp1
```

You can also use the Secret Manager tool to list, remove and clear app secrets.

Accessing user secrets via configuration

You access Secret Manager secrets through the configuration system. Add the `Microsoft.Extensions.Configuration.UserSecrets` package and run `dotnet restore`.

Add the user secrets configuration source to the `Startup` method:

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

namespace UserSecrets
{
    public class Startup
    {
        string _testSecret = null;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder();

            if (env.IsDevelopment())
            {
                builder.AddUserSecrets<Startup>();
            }

            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            _testSecret = Configuration["MySecret"];
        }

        public void Configure(IApplicationBuilder app)
        {
            var result = string.IsNullOrEmpty(_testSecret) ? "Null" : "Not Null";
            app.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Secret is {result}");
            });
        }
    }
}
```

You can access user secrets via the configuration API:

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

namespace UserSecrets
{
    public class Startup
    {
        string _testSecret = null;
        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder();

            if (env.IsDevelopment())
            {
                builder.AddUserSecrets<Startup>();
            }

            Configuration = builder.Build();
        }

        public IConfigurationRoot Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            _testSecret = Configuration["MySecret"];
        }

        public void Configure(IApplicationBuilder app)
        {
            var result = string.IsNullOrEmpty(_testSecret) ? "Null" : "Not Null";
            app.Run(async (context) =>
            {
                await context.Response.WriteAsync($"Secret is {result}");
            });
        }
    }
}
```

How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. In the current version, the values are stored in a JSON configuration file in the user profile directory:

Windows: `%APPDATA%\microsoft\UserSecrets\<userSecretsId>\secrets.json`

Linux: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`

Mac: `~/.microsoft/usersecrets/<userSecretsId>/secrets.json`

The value of `userSecretsId` comes from the value specified in *.csproj* file.

You should not write code that depends on the location or format of the data saved with the Secret Manager tool, as these implementation details might change. For example, the secret values are currently *not* encrypted today, but could be someday.

Additional Resources

Configuration

Azure Key Vault configuration provider

By Luke Latham and Andrew Stanton-Nurse

View or download sample code

This document explains how to use the Microsoft Azure Key Vault configuration provider to load application configuration values from Azure Key Vault. Azure Key Vault is a cloud-based service that helps you safeguard cryptographic keys and secrets used by apps and services. Common scenarios include controlling access to sensitive configuration data and meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSM's) when storing configuration data.

Package

To use the provider, add a reference to the `Microsoft.Extensions.Configuration.AzureKeyVault` package. The provider depends on .NET Framework 4.5.1 or .NET Standard 1.5 or higher. This feature is available for applications that target ASP.NET Core 1.1.0 or higher.

Application configuration

You can explore the provider with the sample application. Once you establish a key vault and create a pair of secrets in the vault by following the guidance below, the sample app securely loads the secret values into its configuration and displays them in a webpage.

The provider is added to the `ConfigurationBuilder` with the `AddAzureKeyVault` extension. In the sample app, the extension uses three configuration values loaded from the *appsettings.json* file.

APP SETTING	DESCRIPTION	EXAMPLE
`Vault`	Azure Key Vault name	contosovault
`ClientId`	Azure Active Directory App Id	627e911e-43cc-61d4-992e-12db9c81b413
`ClientSecret`	Azure Active Directory App Key	g58K3dtg59o1Pa+e59v2Tx829w6VxTB2yv9sv/101di=

```
public Startup()
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", optional: false)
        .AddEnvironmentVariables();

    var config = builder.Build();

    builder.AddAzureKeyVault(
            $"https://{config["Vault"]}.vault.azure.net/",
            config["ClientId"],
            config["ClientSecret"]);

    Configuration = builder.Build();
}
```

`AddAzureKeyVault` also provides an overload that accepts an implementation of `IKeyVaultSecretManager`, which allows you to control how key vault secrets are converted into configuration keys. For example, you can implement the interface to load configuration values by application, where you prefix app names to configuration secrets you've stored in the key vault. This allows you to maintain secrets for multiple apps in one key vault.

Assume we have several `ConnectionString` key vault secrets with the application name prefixed. For the sample app, we create a secret in the key vault for `KeyVaultConfigProviderSample-ConnectionString` and its value. For a second app, we create a secret for `SomeOtherApplicationName-ConnectionString` and its value. We want each app to load its own `ConnectionString` secret into its configuration as `ConnectionString`. An example of this implementation is shown below:

```
public class EnvironmentSecretManager : IKeyVaultSecretManager
{
    private readonly string _appNamePrefix;

    public EnvironmentSecretManager(string appName)
    {
        _appNamePrefix = appName + "-";
    }

    public bool Load(SecretItem secret)
    {
        return secret.Identifier.Name.StartsWith(_appNamePrefix);
    }

    public string GetKey(SecretBundle secret)
    {
        return secret.SecretIdentifier.Name.Substring(_appNamePrefix.Length);
    }
}
```

```
builder.AddAzureKeyVault(
    $"https://{config["Vault"]}.vault.azure.net/",
    config["ClientId"],
    config["ClientSecret"],
    new EnvironmentSecretManager(env.ApplicationName));

Configuration = builder.Build();
```

The `Load` method is called by a provider algorithm that iterates through the secrets to find the one that matches the application name as a prefix to the secret's name. When a match is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret name. It strips off the app name prefix from the secret's name and returns the name for loading into the app's configuration name-value pairs.

If you implement this approach:

The key vault secrets are loaded.
The string secret for `KeyVaultConfigProviderSample-ConnectionString` is matched.
The application name `KeyVaultConfigProviderSample` (with the dash) is stripped off and used to load `ConnectionString` with its value into the app's configuration.

You can also provide your own `KeyVaultClient` implementation to `AddAzureKeyVault`. Supplying a custom client allows you to share a single instance of the client between the configuration provider and other parts of your app.

Controlling access to the ClientSecret

Use the Secret Manager tool to maintain the `ClientSecret` outside of your project source tree. With Secret Manager, you associate app secrets with a specific project and share them across multiple projects.

When developing a .NET Framework app in an environment that supports certificates, you can authenticate to Azure Key Vault with an X.509 certificate. The X.509 certificate's private key is managed by the OS. For more information, see Authenticate with a Certificate instead of a Client Secret. Use the `AddAzureKeyVault` overload that accepts an `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();
```

Creating key vault secrets and loading configuration values

Create a key vault and set up Azure Active Directory (Azure AD) for the application following the guidance in Get started with Azure Key Vault.

Add secrets to the key vault using the Azure PowerShell Module, the Azure Management API, or the Azure Portal. Secrets are uploaded as either *Manual* or *Certificate* secrets. *Certificate* secrets are certificates for use by apps and services but are not supported by the configuration provider. You should use the *Manual* option to create name-value pair secrets for use with the configuration provider.

Hierarchical values (configuration sections) use `--` (two dashes) as a separator.

For the sample app, create two *Manual* secrets with the following name-value pairs:

`MySecret`: `secret_value_1`

`Section--MySecret`: `secret_value_2`

Register the sample app with Azure Active Directory.

Authorize the app to access the key vault. When you use the `Set-AzureRmKeyVaultAccessPolicy` PowerShell cmdlet to authorize the app to access the key vault, provide `List` and `Get` access to secrets with `-PermissionsToKeys list,get`.

Update the app's *appsettings.json* file with the values of `Vault`, `ClientId`, and `ClientSecret`.

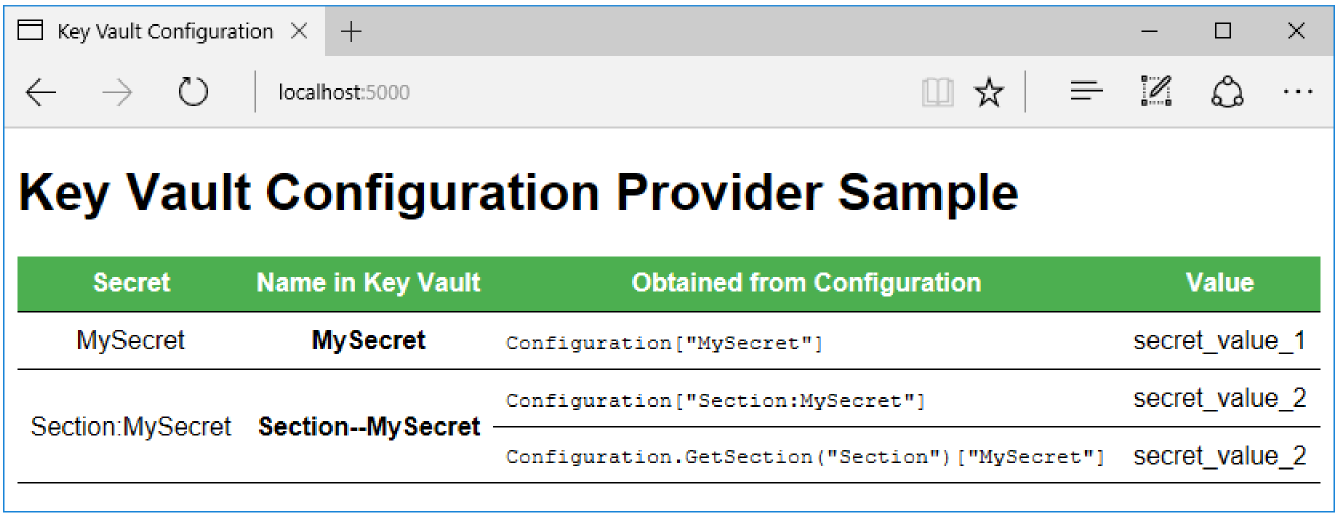
Run the sample app, which obtains its configuration values from `IConfigurationRoot` with the same name as the secret name.

Non-hierarchical Values: The value for `MySecret` is obtained with `config["MySecret"]`.

Hierarchical Values (sections): Use `:` (colon) notation or the `GetSection` extension method.

`config["Section:MySecret"]`

`config.GetSection("Section")["MySecret"]`



Reloading secrets

Secrets are cached until `IConfigurationRoot.Reload()` is called. Expired, disabled, and updated secrets in the key vault are not respected by the application until `Reload` is executed.

```
Configuration.Reload();
```

Disabled and expired secrets

Disabled and expired secrets throw a `KeyVaultClientException`. To prevent your app from throwing, replace your app or update the disabled/expired secret.

Troubleshooting

When the application fails to load configuration using the provider, an error message is written to the ASP.NET Logging infrastructure. The following conditions will prevent configuration from loading:

The app isn't configured correctly in Azure Active Directory.
The key vault doesn't exist in Azure Key Vault.
The app isn't authorized to access the key vault.
The access policy doesn't include `Get` and `List` permissions.
In the key vault, the configuration data (name-value pair) is incorrectly named, missing, disabled, or expired.
The app has the wrong key vault name (`Vault`), Azure AD App Id (`ClientId`), or Azure AD Key (`ClientSecret`).
The Azure AD Key (`ClientSecret`) is expired.
The configuration key (name) is incorrect in the app for the value you're trying to load.

`System.TypeLoadException` exception during startup using `.NET 4.6.x`:

Add a NuGet reference to the package for `System.Net.Http` version 4.3.1 or later.

Additional resources

Configuration in ASP.NET Core | Microsoft Docs
Microsoft Azure: Key Vault
Microsoft Azure: Key Vault Documentation
How to generate and transfer HSM-protected keys for Azure Key Vault
KeyVaultClient Class

Preventing Cross-Site Request Forgery (XSRF/CSRF) Attacks in ASP.NET Core

Steve Smith, Fiyaz Hasan

What attack does anti-forgery prevent?

Cross-site request forgery (also known as XSRF or CSRF, pronounced *see-surf*) is an attack against web-hosted applications whereby a malicious web site can influence the interaction between a client browser and a web site that trusts that browser. These attacks are made possible because web browsers send some types of authentication tokens automatically with every request to a web site. This form of exploit is also known as a *one-click attack* or as *session riding*, because the attack takes advantage of the user's previously authenticated session.

An example of a CSRF attack:

A user logs into `www.example.com`, using forms authentication.

The server authenticates the user and issues a response that includes an authentication cookie.

The user visits a malicious site.

This malicious site contains the following HTML form:

```
<h1>You Are a Winner!</h1>
  <form action="http://example.com/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw" />
    <input type="hidden" name="Amount" value="1000000" />
  <input type="submit" value="Click Me"/>
</form>
```

Notice that the form action posts to the vulnerable site, not to the malicious site. This is the “cross-site” part of CSRF.

The user clicks the submit button. The browser automatically includes the authentication cookie for the requested domain (the vulnerable site in this case) with the request.

The request runs on the server with the user’s authentication context, and can do anything that an authenticated user is allowed to do.

Although this example requires the user to click the form button, the malicious page could just as easily run a script that automatically submits the form or sends a form submission as an AJAX request. The form could also be hidden using CSS so the user never realizes it's present. Moreover, using SSL does not prevent a CSRF attack, because the malicious site can send an `https://` request. Some attacks can target site endpoints that respond to `GET` requests, in which case even an image tag can be used to perform the action (this form of attack is common on forum sites that permit images but block JavaScript). If your application uses `GET` requests to significantly change the state of the application, you should switch to `POST` if possible (in addition to protecting against CSRF attacks).

Typically, CSRF attacks are possible against web sites that use cookies for authentication, because browsers send all relevant cookies to the destination web site. However, CSRF attacks are not limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user logs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

Note

In this context, *session* refers to the client-side session during which the user is authenticated. It is unrelated to server-side sessions or session middleware.

Users can guard against CSRF vulnerabilities by:

Logging off of web sites when they have finished using them
Clearing their browser's cookies periodically

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

How does ASP.NET Core MVC address CSRF?

The most common approach to defending against CSRF attacks is the synchronizer token pattern (STP). STP is a technique used when the user requests a page with form data. The server sends a token associated with the current user's identity to the client. The client must send back the token to the server for verification. If the server receives a token that doesn't match the authenticated user's identity, the request should be rejected. The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (ensuring page 1 precedes page 2 which precedes page 3). ASP.NET Core MVC will generate Antiforgery Tokens by default on all forms it generates. The following two examples of view logic will generate antiforgery tokens automatically:

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">

</form>

@using (Html.BeginForm("ChangePassword", "Manage"))
{

}
```

You can also explicitly add an antiforgery token to a `<form>` element you create without using tag helpers or HTML helpers by using `@Html.AntiForgeryToken()`:

```
<form action="/" method="post">
    @Html.AntiForgeryToken()
</form>
```

```
In each of the above cases, ASP.NET Core will add a hidden form field like the following:

<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8NrAkSldwD9CpLRyOtm6FiJB1Jr_F3FQJQDvhlHoLNJJrLA6zaMUmhjMsisu2D2tFkAiYgyWQawJk9vNm36sYP1esHOtamBEPv
Sk1_x--Sg8Ey2a-d9CV2zHVWIN9MVhvKHOSyKqdZFlYDVd69XYx-rOWPw3ilHGLN6K0Km-1p83jZzF0E4WU5OGg5ns2-m9Yw" />
```

ASP.NET Core includes three filters for working with antiforgery tokens: `ValidateAntiForgeryToken`, `AutoValidateAntiforgeryToken`, and `IgnoreAntiforgeryToken`.

ValidateAntiForgeryToken

The `ValidateAntiForgeryToken` is an action filter that can be applied to an individual action, a controller, or globally for the app. Requests made to actions that have this filter applied will be blocked unless the request includes a valid antiforgery token.

```
[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 });
}
```

The `ValidateAntiForgeryToken` attribute will require a token for requests to action methods it decorates, including *GET* requests. If you apply it broadly, you can override it with the `IgnoreAntiforgeryToken` attribute.

AutoValidateAntiforgeryToken

In most cases, your application will not receive antiforgery tokens for certain kinds of HTTP requests, such as GET requests. Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, you can use the `AutoValidateAntiforgeryToken` attribute. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

GET
HEAD
OPTIONS
TRACE

We recommend you use `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This ensures your POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to the individual action method. It's more likely in this scenario for a POST action method to be left unprotected, leaving your app vulnerable to CSRF attacks. Even anonymous POSTS should send the antiforgery token.

Note

APIs don't have an automatic mechanism for sending the non-cookie part of the token; your implementation will likely depend on your client code implementation. Some examples are shown below.

Example (class level):

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
```

Example (global):

```
services.AddMvc(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

IgnoreAntiforgeryToken

The `IgnoreAntiforgeryToken` filter is used to eliminate the need for an antiforgery token to be present for a given action (or controller). When applied, this filter will override `ValidateAntiForgeryToken` and/or `AutoValidateAntiforgeryToken` filters specified at a higher level (globally or on a controller).

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
  [HttpPost]
  [IgnoreAntiforgeryToken]
  public async Task<IActionResult> DoSomethingSafe(SomeViewModel model)
  {
    // no antiforgery token required
  }
}
```

JavaScript, AJAX, and SPAs

In traditional HTML-based applications, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and single page applications (SPAs), many requests are made programmatically. These AJAX requests may use other techniques (such as request headers or cookies) to send the token. If cookies are used to store authentication tokens and to authenticate API requests on the server, then CSRF will be a potential problem. However, if local storage is used to store the token, CSRF vulnerability may be mitigated, since values from local storage are not sent automatically to the server with every new request. Thus, using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

AngularJS

AngularJS uses a convention to address CSRF. If the server sends a cookie with the name `XSRF-TOKEN`, the Angular `$http` service will add the value from this cookie to a header when it sends a request to this server. This process is automatic; you don't need to set the header explicitly. The header name is `X-XSRF-TOKEN`. The server should detect this header and validate its contents.

For ASP.NET Core API work with this convention:

Configure your app to provide a token in a cookied called `XSRF-TOKEN`
Configure the antiforgery service to look for a header named `X-XSRF-TOKEN`

```
services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
```

View sample.

JavaScript

Using JavaScript with views, you can create the token using a service from within your view. To do so, you inject the `Microsoft.AspNetCore.Antiforgery.IAntiforgery` service into the view and call `GetAndStoreTokens`, as shown:

```
@{
    ViewData["Title"] = "AJAX Demo";
}
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <input type="button" id="antiforgery" value="Antiforgery" />
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"></script>
    <script>
                $("#antiforgery").click(function () {
                    $.ajax({
                        type: "post",
                        dataType: "html",
                        headers:
                        {
                            "RequestVerificationToken": '@GetAntiXsrfRequestToken()'
                        },
                        url: '@Url.Action("Antiforgery", "Home")',
                        success: function (result) {
                            alert(result);
                        },
                        error: function (err, scnd) {
                            alert(err.statusText);
                        }
                    });
                });
    </script>
</div>
```

This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client.

JavaScript can also access tokens provided in cookies, and then use the cookie's contents to create a header with the token's value, as shown below.

```
context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken,
  new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });
```

Then, assuming you construct your script requests to send the token in a header called `X-CSRF-TOKEN`, configure the antiforgery service to look for the `X-CSRF-TOKEN` header:

```
services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");
```

The following example uses jQuery to make an AJAX request with the appropriate header:

Configuring Antiforgery

`IAntiforgery` provides the API to configure the antiforgery system. It can be requested in the `Configure` method of the `Startup` class. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie (using the default Angular naming convention described above):

```
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))
        {
            // We can send the request token as a JavaScript-readable cookie,
            // and Angular will use it by default.
            var tokens = antiforgery.GetAndStoreTokens(context);
            context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
                new CookieOptions() { HttpOnly = false });
        }

        return next(context);
    });
    //
}
```

Options

You can customize antiforgery options in `ConfigureServices`:

```
services.AddAntiforgery(options =>
{
  options.CookieDomain = "mydomain.com";
  options.CookieName = "X-CSRF-TOKEN-COOKIENAME";
  options.CookiePath = "Path";
  options.FormFieldName = "AntiforgeryFieldname";
  options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
  options.RequireSsl = false;
  otpions.SuppressXFrameOptionsHeader = false;
});
```

OPTION	DESCRIPTION
CookieDomain	The domain of the cookie. Defaults to `null`.
CookieName	The name of the cookie. If not set, the system will generate a unique name beginning with the `DefaultCookiePrefix` (".AspNetCore.Antiforgery.").
CookiePath	The path set on the cookie.
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If `null`, the system will consider only form data.
RequireSsl	Specifies whether SSL is required by the antiforgery system. Defaults to `false`. If `true`, non-SSL requests will fail.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the `X-Frame-Options` header. By default the header is generated with a value of "SAMEORIGIN". Defaults to `false`.

See https://docs.microsoft.com/aspnet/core/api/microsoft.aspnetcore.builder.cookieauthenticationoptions for more info.

Extending Antiforgery

The IAntiForgeryAdditionalDataProvider type allows developers to extend the behavior of the anti-XSRF system by round-tripping additional data in each token. The GetAdditionalData method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call ValidateAdditionalData to validate this data when the token is validated. The client's username is already embedded in the generated tokens, so there is no need to include this information. If a token includes supplemental data but no `IAntiForgeryAdditionalDataProvider` has been configured, the supplemental data is not validated.

Fundamentals

CSRF attacks rely on the default browser behavior of sending cookies associated with a domain with every request made to that domain. These cookies are stored within the browser. They frequently include session cookies for authenticated users. Cookie-based authentication is the most popular form of authentication used by web applications. However, token-based authentication systems have been growing in popularity in recent years, especially for SPAs and other "smart client" scenarios.

Cookie based authentication

Once a user has authenticated using their username and password, they are issued a token that can be used to identify them and validate that they have been authenticated. The token is stored as a cookie that accompanies every request the client makes. Generating and validating this cookie is done by the cookie authentication middleware. ASP.NET Core provides cookie middleware which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal and assigns it to the `User` property on `HttpContext`.

When a cookie is used, The authentication cookie is just a container for the forms authentication ticket. The ticket is passed as the value of the forms authentication cookie with each request and is used by forms authentication, on the server, to identify an authenticated user.

When a user is logged in to a system, a user session is created on the server side and is stored in a database or some other persistent store. The system generates a session key that points to the actual session in the data store and it is sent as a client side cookie. The web server will check this session key any time a user requests a resource that requires authorization. The system checks whether the associated user session has the privilege to access the requested resource. If so, the request continues. Otherwise, the request returns as not authorized. In this approach, cookies are used to make the application appear to be stateful, since it is able to "remember" that the user has previously authenticated with the server.

User tokens

Token based authentication doesn’t store any kind of session on the server or in a server-side data store. Instead when a user is logged in they are issued a token (not an antiforgery token). This token holds all the data that is required to validate the token. It also contains user information, in the form of claims. When a user wants to access a server resource requiring authentication, the token is sent to the server with an additional authorization header in form of Bearer {token}. This makes the application stateless since in each subsequent request the token is passed in the request for server side validation. One thing to remember is this token is not *encrypted*; rather it is *encoded*. On the server side the token can be decoded to access the raw information within the token. To send the token in subsequent requests, you can either store it in browser’s local storage or in a cookie. You don’t have to worry about XSRF vulnerability if your token is stored in the local storage, but it is still an issue if the token is stored in a cookie.

Multiple applications are hosted in one domain

Even though `example1.cloudapp.net` and `example2.cloudapp.net` are different hosts, there is an implicit trust relationship between all hosts under the `*.cloudapp.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other’s cookies (the same-origin policies that govern AJAX requests do not necessarily apply to HTTP cookies). The ASP.NET Core runtime provides some mitigation in that the username is embedded into the field token, so even if a malicious subdomain is able to overwrite a session token it will be unable to generate a valid field token for the user. However, when hosted in such an environment the built-in anti-XSRF routines still cannot defend against session hijacking or login CSRF attacks.

Note

You should only host live and preproduction apps in domains you fully control, rather than shared domains like `azurewebsites.net` or `cloudapp.net`. This will better protect your app's users from session hijacking and/or CSRF attacks.

Additional Resources

XSRF on Open Web Application Security Project (OWASP).

Preventing Cross-Site Scripting

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attackers scripts will run, enabling the attacher to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it in a page without validating, encoding or escaping it.

Protecting your application against XSS

At a basic level XSS works by tricking your application into inserting a `<script>` tag into your rendered page, or by inserting an `On*` event into an element. Developers should use the following prevention steps to avoid introducing XSS into their application.

Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.

Before putting untrusted data inside an HTML element ensure it is HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like <

Before putting untrusted data into an HTML attribute ensure it is HTML attribute encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and '.

Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this is not possible then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example < would be encoded as `\u003C`.

Before putting untrusted data into a URL query string ensure it is URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML Attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view;

```
@{
       var untrustedInput = "<\"123\">";
   }

   @untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

```
&lt;&quot;123&quot;&gt;
```

Warning

ASP.NET Core MVC provides an `HtmlString` class which is not automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

Javascript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert simple values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

```
@{
       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>
```

This will produce the following HTML

```
<div
     id="injectedData"
     data-untrustedinput="&lt;&quot;123&quot;&gt;" />

   <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>
```

Which, when it runs, will render the following;

```
<"123">
   <"123">
```

You can also call the JavaScript encoder directly,

```
@using System.Text.Encodings.Web;
   @inject JavaScriptEncoder encoder;

   @{
       var untrustedInput = "<\"123\">";
   }

   <script>
       document.write("@encoder.Encode(untrustedInput)");
   </script>
```

This will render in the browser as follows;

```
<script>
       document.write("\u003C\u0022123\u0022\u003E");
   </script>
```

 Warning

Do not concatenate untrusted input in JavaScript to create DOM elements. You should use `createElement()` and assign property values appropriately such as `node.TextContent=`, or use `element.SetAttribute()` / `element[attribute]=` otherwise you expose yourself to DOM-based XSS.

Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways, you can inject them via dependency injection or you can use the default encoders contained in the `System.Text.Encodings.Web` namespace. If you use the default encoders then any you applied to character ranges to be treated as safe will not take effect - the default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

```
public class HomeController : Controller
   {
       HtmlEncoder _htmlEncoder;
       JavaScriptEncoder _javaScriptEncoder;
       UrlEncoder _urlEncoder;

       public HomeController(HtmlEncoder htmlEncoder,
                             JavaScriptEncoder javascriptEncoder,
                             UrlEncoder urlEncoder)
       {
           _htmlEncoder = htmlEncoder;
           _javaScriptEncoder = javascriptEncoder;
           _urlEncoder = urlEncoder;
       }
   }
```

Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the `UrlEncoder` to encode the value. For example,

```
var example = "\"Quoted Value with spaces and &\"";
   var encodedValue = _urlEncoder.Encode(example);
```

After encoding the encodedValue variable will contain `%22Quoted%20Value%20with%20spaces%20and%20%26%22`. Spaces, quotes, punctuation and other unsafe characters will be percent encoded to their hexadecimal value, for example a space character will become %20.

Warning

Do not use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it will use the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.

You can customize the encoder safe lists to include Unicode ranges appropriate to your application during startup, in `ConfigureServices()`.

For example, using the default configuration you might use a Razor HtmlHelper like so;

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

When you view the source of the web page you will see it has been rendered as follows, with the Chinese text encoded;

```
<p>This link text is in Chinese: <a href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder you would insert the following line into the `ConfigureServices()` method in `startup.cs`;

```
services.AddSingleton<HtmlEncoder>(
     HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
                                               UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range CjkUnifiedIdeographs. The rendered output would now become

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The Unicode standard has a list of code charts you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be configured separately.

Note

Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via `System.Text.Encodings.Web.*Encoder.Default` then the default, Basic Latin only safelist will be used.

Where should encoding take place?

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a simple numeric string containing only the characters 0-9 will not trigger an XSS attack. Validation becomes more complicated should you wish to accept HTML in user input - parsing HTML input is difficult, if not impossible. MarkDown and other text formats would be a safer option for rich input. You should never rely on validation alone. Always encode untrusted input before output, no matter what validation you have performed.

Enabling Cross-Origin Requests (CORS)

By Mike Wasson, Shayne Boyer, and Tom Dykstra

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy*, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites make cross-origin requests to your web API.

Cross Origin Resource Sharing (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. CORS is safer and more flexible than earlier techniques such as JSONP. This topic shows how to enable CORS in an ASP.NET Core application.

What is "same origin"?

Two URLs have the same origin if they have identical schemes, hosts, and ports. (RFC 6454)

These two URLs have the same origin:

`http://example.com/foo.html`

`http://example.com/bar.html`

These URLs have different origins than the previous two:

`http://example.net` - Different domain

`http://www.example.com/foo.html` - Different subdomain

`https://example.com/foo.html` - Different scheme

`http://example.com:9000/foo.html` - Different port

Note

Internet Explorer does not consider the port when comparing origins.

Setting up CORS

To set up CORS for your application add the `Microsoft.AspNetCore.Cors` package to your project.

Add the CORS services in Startup.cs:

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

Enabling CORS with middleware

To enable CORS for your entire application add the CORS middleware to your request pipeline using the `UseCors` extension method. Note that the CORS middleware must precede any defined endpoints in your app that you want to support cross-origin requests (ex. before any call to `UseMvc`).

You can specify a cross-origin policy when adding the CORS middleware using the `CorsPolicyBuilder` class. There are two ways to do this. The first is to call UseCors with a lambda:

```
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!");
    });
```

The lambda takes a `CorsPolicyBuilder` object. You'll find a list of the configuration options later in this topic. In this example, the policy allows cross-origin requests from `http://example.com` and no other origins.

Note that CorsPolicyBuilder has a fluent API, so you can chain method calls:

```
app.UseCors(builder =>
    builder.WithOrigins("http://example.com")
           .AllowAnyHeader()
    );
```

The second approach is to define one or more named CORS policies, and then select the policy by name at run time.

```
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!");
    });
}
```

This example adds a CORS policy named "AllowSpecificOrigin". To select the policy, pass the name to `UseCors`.

Enabling CORS in MVC

You can alternatively use MVC to apply specific CORS per action, per controller, or globally for all controllers. When using MVC to enable CORS the same CORS services are used, but the CORS middleware is not.

Per action

To specify a CORS policy for a specific action add the `[EnableCors]` attribute to the action. Specify the policy name.

```
[HttpGet]
[EnableCors("AllowSpecificOrigin")]
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

Per controller

To specify the CORS policy for a specific controller add the `[EnableCors]` attribute to the controller class. Specify the policy name.

```
[Route("api/[controller]")]
[EnableCors("AllowSpecificOrigin")]
public class ValuesController : Controller
```

Globally

You can enable CORS globally for all controllers by adding the `CorsAuthorizationFilterFactory` filter to the global filter collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new CorsAuthorizationFilterFactory("AllowSpecificOrigin"));
    });
}
```

The precedence order is: Action, controller, global. Action-level policies take precedence over controller-level policies, and controller-level policies take precedence over global policies.

Disable CORS

To disable CORS for a controller or action, use the `[DisableCors]` attribute.

```
[HttpGet("{id}")]
[DisableCors]
public string Get(int id)
{
    return "value";
}
```

CORS policy options

This section describes the various options that you can set in a CORS policy.

Set the allowed origins

Set the allowed HTTP methods

Set the allowed request headers

Set the exposed response headers

Credentials in cross-origin requests

Set the preflight expiration time

For some options it may be helpful to read How CORS works first.

Set the allowed origins

To allow one or more specific origins:

```
options.AddPolicy("AllowSpecificOrigins",
builder =>
{
    builder.WithOrigins("http://example.com", "http://www.contoso.com");
});
```

To allow all origins:

```
options.AddPolicy("AllowAllOrigins",
    builder =>
    {
        builder.AllowAnyOrigin();
    });
```

Consider carefully before allowing requests from any origin. It means that literally any website can make AJAX calls to your API.

Set the allowed HTTP methods

To allow all HTTP methods:

```
options.AddPolicy("AllowAllMethods",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyMethod();
    });
```

This affects pre-flight requests and Access-Control-Allow-Methods header.

Set the allowed request headers

A CORS preflight request might include an Access-Control-Request-Headers header, listing the HTTP headers set by the application (the so-called "author request headers").

To whitelist specific headers:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders("accept", "content-type", "origin", "x-custom-header");
    });
```

To allow all author request headers:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

Browsers are not entirely consistent in how they set Access-Control-Request-Headers. If you set headers to anything other than "*", you should include at least "accept", "content-type", and "origin", plus any custom headers that you want to support.

Set the exposed response headers

By default, the browser does not expose all of the response headers to the application. (See http://www.w3.org/TR/cors/#simple-response-header.) The response headers that are available by default are:

Cache-Control

Content-Language

Content-Type

Expires

Last-Modified

Pragma

The CORS spec calls these *simple response headers*. To make other headers available to the application:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser does not send any credentials with a cross-origin request. Credentials include cookies as well as HTTP authentication schemes. To send credentials with a cross-origin request, the client must set XMLHttpRequest.withCredentials to true.

Using XMLHttpRequest directly:

In jQuery:

```
$.ajax({
  type: 'get',
  url: 'http://www.example.com/home',
  xhrFields: {
    withCredentials: true
}
```

In addition, the server must allow the credentials. To allow cross-origin credentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowCredentials();
    });
```

Now the HTTP response will include an Access-Control-Allow-Credentials header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials, but the response does not include a valid Access-Control-Allow-Credentials header, the browser will not expose the response to the application, and the AJAX request fails.

Be very careful about allowing cross-origin credentials, because it means a website at another domain can send a logged-in user's credentials to your app on the user's behalf, without the user being aware. The CORS spec also states that setting origins to "*" (all origins) is invalid if the Access-Control-Allow-Credentials header is present.

Set the preflight expiration time

The Access-Control-Max-Age header specifies how long the response to the preflight request can be cached. To set this header:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request, at the level of the HTTP messages. It's important to understand how CORS works, so that you can configure the your CORS policy correctly, and troubleshoot if things don't work as you expect.

The CORS specification introduces several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests; you don't need to do anything special in your JavaScript code.

Here is an example of a cross-origin request. The "Origin" header gives the domain of the site that is making the request:

```
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
```

If the server allows the request, it sets the Access-Control-Allow-Origin header. The value of this header either matches the Origin header, or is the wildcard value "*", meaning that any origin is allowed.:

```
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
```

If the response does not include the Access-Control-Allow-Origin header, the AJAX request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser does not make the response available to the client application.

Preflight Requests

For some CORS requests, the browser sends an additional request, called a "preflight request", before it sends the actual request for the resource. The browser can skip the preflight request if the following conditions are true:

The request method is GET, HEAD, or POST, and

The application does not set any request headers other than Accept, Accept-Language, Content-Language, Content-Type, or Last-Event-ID, and

The Content-Type header (if set) is one of the following:

application/x-www-form-urlencoded

multipart/form-data

text/plain

The rule about request headers applies to headers that the application sets by calling setRequestHeader on the XMLHttpRequest object. (The CORS specification calls these "author request headers".) The rule does not apply to headers the browser can set, such as User-Agent, Host, or Content-Length.

Here is an example of a preflight request:

```
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
```

The pre-flight request uses the HTTP OPTIONS method. It includes two special headers:

Access-Control-Request-Method: The HTTP method that will be used for the actual request.

Access-Control-Request-Headers: A list of request headers that the application set on the actual request. (Again, this does not include headers that the browser sets.)

Here is an example response, assuming that the server allows the request:

```
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
```

The response includes an Access-Control-Allow-Methods header that lists the allowed methods, and optionally an Access-Control-Allow-Headers header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request, as described earlier.

Introduction to in-memory caching in ASP.NET Core

By Rick Anderson, John Luo, and Steve Smith

View or download sample code

Caching basics

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently. Caching makes a copy of data that can be returned much faster than from the original source. You should write and test your app to never depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the IMemoryCache, which represents a cache stored in the memory of the web server. Apps which run on a server farm of multiple servers should ensure that sessions are sticky when using the in-memory cache. Sticky sessions ensure that subsequent requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all subsequent requests to the same server.

Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The `IMemoryCache` cache will evict cache entries under memory pressure unless the cache priority is set to `CacheItemPriority.NeverRemove`. You can set the `CacheItemPriority` to adjust the priority the cache evicts items under memory pressure.

The in-memory cache can store any object; the distributed cache interface is limited to `byte[]`.

Using IMemoryCache

In-memory caching is a *service* that is referenced from your app using Dependency Injection. Call `AddMemoryCache` in `ConfigureServices`:

```
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();
    }
}
```

Request the `IMemoryCache` instance in the constructor:

```
public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
```

`IMemoryCache` requires NuGet package "Microsoft.Extensions.Caching.Memory".

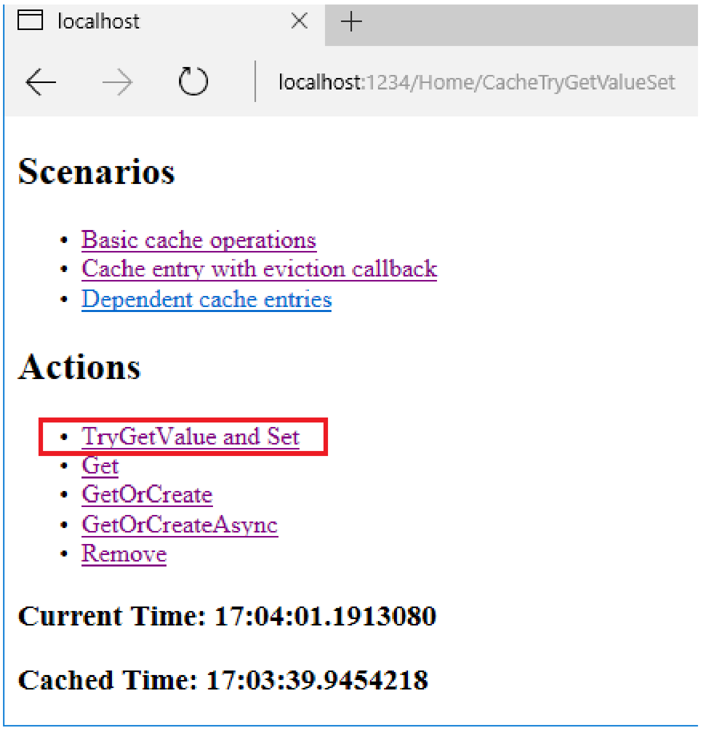
The following code uses TryGetValue to check if the current time is in the cache. If the item is not cached, a new entry is created and added to the cache with Set.

```
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);
}
```

The current time and the cached time is displayed:

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>
```

The cached `DateTime` value will remain in the cache while there are requests within the timeout period (and no eviction due to memory pressure). The image below shows the current time and an older time retrieved from cache:



The following code uses GetOrCreate and GetOrCreateAsync to cache data.

```
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);
}
```

The following code calls Get to fetch the cached time:

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

See IMemoryCache methods and CacheExtensions methods for a description of the cache methods.

Using MemoryCacheEntryOptions

The following sample:

Sets the absolute expiration time. This is the maximum time the entry can be cached and prevents the item from becoming too stale when the sliding expiration is continuously renewed.
Sets a sliding expiration time. Requests that access this cached item will reset the sliding expiration clock.
Sets the cache priority to `CacheItemPriority.NeverRemove`.
Sets a PostEvictionDelegate that will be called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

```
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);
}
```

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A `CancellationChangeToken` is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted.

```
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);
}
```

Using a `CancellationTokenSource` allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` block will inherit triggers and expiration settings.

Additional notes

When using a callback to repopulate a cache item:

Multiple requests can find the cached key value empty because the callback hasn't completed.
This can result in several threads repopulating the cached item.

When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child is not expired by manual removal or updating of the parent entry.

Other Resources

[Working with a Distributed Cache](#)
[Response caching middleware](#)

Working with a distributed cache

By Steve Smith

Distributed caches can improve the performance and scalability of ASP.NET Core apps, especially when hosted in a cloud or server farm environment. This article explains how to work with ASP.NET Core's built-in distributed cache abstractions and implementations.

View or download sample code

What is a Distributed Cache

A distributed cache is shared by multiple app servers (see Caching Basics). The information in the cache is not stored in the memory of individual web servers, and the cached data is available to all of the app's servers. This provides several advantages:

Cached data is coherent on all web servers. Users don't see different results depending on which web server handles their request

Cached data survives web server restarts and deployments. Individual web servers can be removed or added without impacting the cache.

The source data store has fewer requests made to it (than with multiple in-memory caches or no cache at all).

Note

If using a SQL Server Distributed Cache, some of these advantages are only true if a separate database instance is used for the cache than for the app's source data.

Like any cache, a distributed cache can dramatically improve an app's responsiveness, since typically data can be retrieved from the cache much faster than from a relational database (or web service).

Cache configuration is implementation specific. This article describes how to configure both Redis and SQL Server distributed caches. Regardless of which implementation is selected, the app interacts with the cache using a common `IDistributedCache` interface.

The IDistributedCache Interface

The `IDistributedCache` interface includes synchronous and asynchronous methods. The interface allows items to be added, retrieved, and removed from the distributed cache implementation. The `IDistributedCache` interface includes the following methods:

Get, GetAsync

Takes a string key and retrieves a cached item as a `byte[]` if found in the cache.

Set, SetAsync

Adds an item (as `byte[]`) to the cache using a string key.

Refresh, RefreshAsync

Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).

Remove, RemoveAsync

Removes a cache entry based on its key.

To use the `IDistributedCache` interface:

Add the required NuGet packages to your project file.

Configure the specific implementation of `IDistributedCache` in your `Startup` class's `ConfigureServices` method, and add it to the container there.

From the app's `Middleware or MVC controller classes, request an instance of `IDistributedCache` from the constructor. The instance will be provided by Dependency Injection (DI).

Note

There is no need to use a Singleton or Scoped lifetime for `IDistributedCache` instances (at least for the built-in implementations). You can also create an instance wherever you might need one (instead of using Dependency Injection), but this can make your code harder to test, and violates the Explicit Dependencies Principle.

The following example shows how to use an instance of `IDistributedCache` in a simple middleware component:

```
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>();
        }
    }
}
```

In the code above, the cached value is read, but never written. In this sample, the value is only set when a server starts up, and doesn't change. In a multi-server scenario, the most recent server to start will overwrite any previous values that were set by other servers. The `Get` and `Set` methods use the `byte[]` type. Therefore, the string value must be converted using `Encoding.UTF8.GetString` (for `Get`) and `Encoding.UTF8.GetBytes` (for `Set`).

The following code from *Startup.cs* shows the value being set:

```
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);
```

Note

Since `IDistributedCache` is configured in the `ConfigureServices` method, it is available to the `Configure` method as a parameter. Adding it as a parameter will allow the configured instance to be provided through DI.

Using a Redis Distributed Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can use it locally, and you can configure an Azure Redis Cache for your Azure-hosted ASP.NET Core apps. Your ASP.NET Core app configures the cache implementation using a `RedisDistributedCache` instance.

You configure the Redis implementation in `ConfigureServices` and access it in your app code by requesting an instance of `IDistributedCache` (see the code above).

In the sample code, a `RedisCache` implementation is used when the server is configured for a `Staging` environment. Thus the `ConfigureStagingServices` method configures the `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";
    });
}
```

Note

To install Redis on your local machine, install the chocolatey package http://chocolatey.org/packages/redis-64/ and run `redis-server` from a command prompt.

Using a SQL Server Distributed Cache

The SqlServerCache implementation allows the distributed cache to use a SQL Server database as its backing store. To create SQL Server table you can use sql-cache tool, the tool creates a table with the name and schema you specify.

To use the sql-cache tool, add `SqlConfig.Tools` to the <ItemGroup> element of the *.csproj* file and run dotnet restore.

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.Extensions.Caching.SqlConfig.Tools" Version="1.0.0-msbuild3-final" />
</ItemGroup>
```

Test SqlConfig.Tools by running the following command

```
C:\DistCacheSample\src\DistCacheSample>dotnet sql-cache create --help
```

sql-cache tool will display usage, options and command help, now you can create tables into sql server, running "sql-cache create" command :

```
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.
```

The created table has the following schema:

	Column Name	Data Type	Allow Nulls
▶🔑	Id	nvarchar(900)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSecon...	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑
			☐

Like all cache implementations, your app should get and set cache values using an instance of `IDistributedCache`, not a `SqlServerCache`. The sample implements `SqlServerCache` in the `Production` environment (so it is configured in `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";
    });

}
```

Note

The `ConnectionString` (and optionally, `SchemaName` and `TableName`) should typically be stored outside of source control (such as UserSecrets), as they may contain credentials.

Recommendations

When deciding which implementation of `IDistributedCache` is right for your app, choose between Redis and SQL Server based on your existing infrastructure and environment, your performance requirements, and your team's experience. If your team is more comfortable working with Redis, it's an excellent choice. If your team prefers SQL Server, you can be confident in that implementation as well. Note that A traditional caching solution stores data in-memory which allows for fast retrieval of data. You should store commonly used data in a cache and store the entire data in a backend persistent store such as SQL Server or Azure Storage. Redis Cache is a caching solution which gives you high throughput and low latency as compared to SQL Cache.

Additional resources:

In memory caching
Redis Cache on Azure
SQL Database on Azure

Response Caching

By John Luo, Rick Anderson, and Steve Smith

View or download sample code

What is Response Caching

Response caching adds cache-related headers to responses. These headers specify how you want client, proxy and middleware to cache responses. Response caching can reduce the number of requests a client or proxy makes to the web server. Response caching can also reduce the amount of work the web server performs to generate the response.

The primary HTTP header used for caching is `Cache-Control`. See the HTTP 1.1 Caching for more information. Common cache directives:

public
private
no-cache
Pragma
Vary

The web server can cache responses by adding the response caching middleware. See Response caching middleware for more information.

ResponseCache Attribute

The `ResponseCacheAttribute` specifies the parameters necessary for setting appropriate headers in response caching. See ResponseCacheAttribute for a description of the parameters.

⚠ Warning

Disable caching for content that contains information for authenticated clients. Caching should only be enabled for content that does not change based on a user's identity, or whether a user is logged in.

`VaryByQueryKeys string[]` (requires ASP.NET Core 1.1.0 and higher): When set, the response caching middleware will vary the stored response by the values of the given list of query keys. The response caching middleware must be enabled to set the `VaryByQueryKeys` property, otherwise a runtime exception will be thrown. There is no corresponding HTTP header for the `VaryByQueryKeys` property. This property is an HTTP feature handled by the response caching middleware. For the middleware to serve a cached response, the query string and query string value must match a previous request. For example, consider the following sequence:

REQUEST	RESULT
`http://example.com?key1=value1`	returned from server
`http://example.com?key1=value1`	returned from middleware
`http://example.com?key1=value2`	returned from server

The first request is returned by the server and cached in middleware. The second request is returned by middleware because the query string matches the previous request. The third request is not in the middleware cache because the query string value doesn't match a previous request.

The `ResponseCacheAttribute` is used to configure and create (via `IFilterFactory`) a `ResponseCacheFilter`. The

`ResponseCacheFilter` performs the work of updating the appropriate HTTP headers and features of the response. The filter:

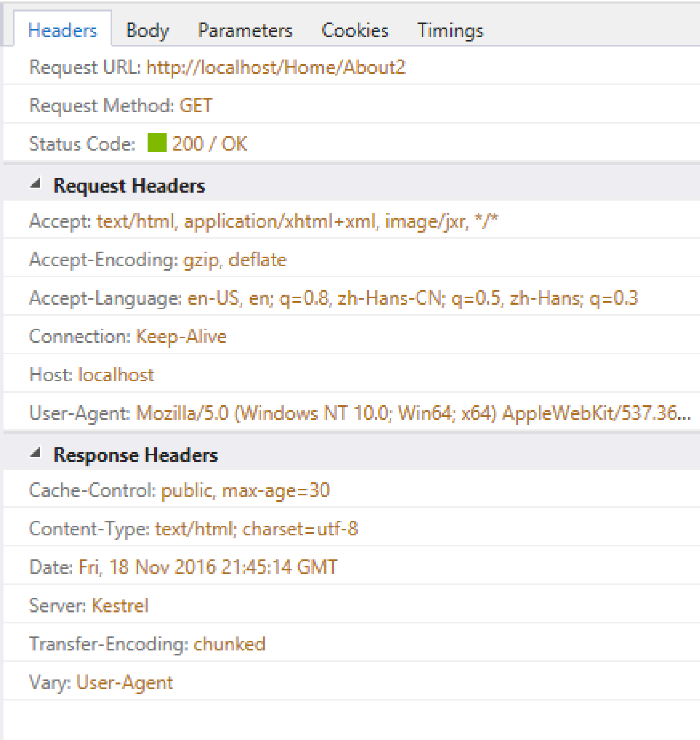
Removes any existing headers for `Vary`, `Cache-Control`, and `Pragma`.
Writes out the appropriate headers based on the properties set in the `ResponseCacheAttribute`.
Updates the response caching HTTP feature if `VaryByQueryKeys` is set.

Vary

This header is only written when the `VaryByHeader` property is set. It is set to the `Vary` property's value. The following sample uses the `VaryByHeader` property.

```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public IActionResult About2()
{
```

You can view the response headers with your browsers network tools. The following image shows the Edge F12 output on the **Network** tab when the `About2` action method is refreshed.



NoStore and Location.None

`NoStore` overrides most of the other properties. When this property is set to `true`, the `Cache-Control` header will be set to "no-store". If `Location` is set to `None`:

`Cache-Control` is set to `"no-store, no-cache"`.
`Pragma` is set to `no-cache`.

If `NoStore` is `false` and `Location` is `None`, `Cache-Control` and `Pragma` will be set to `no-cache`.

You typically set `NoStore` to `true` on error pages. For example:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

This will result in the following headers:

Location and Duration

To enable caching, `Duration` must be set to a positive value and `Location` must be either `Any` (the default) or `Client`. In this case, the `Cache-Control` header will be set to the location value followed by the "max-age" of the response.

Note

`Location`'s options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted previously, setting `Location` to `None` will set both `Cache-Control` and `Pragma` headers to `no-cache`.

Below is an example showing the headers produced by setting `Duration` and leaving the default `Location` value.

```
[ResponseCache(Duration = 60)]
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    return View();
}
```

Produces the following headers:

Cache Profiles

Instead of duplicating `ResponseCache` settings on many controller action attributes, cache profiles can be configured as options when setting up MVC in the `ConfigureServices` method in `Startup`. Values found in a referenced cache profile will be used as the defaults by the `ResponseCache` attribute, and will be overridden by any properties specified on the attribute.

Setting up a cache profile:

```
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
            });
    });
}
```

Referencing a cache profile:

```
[ResponseCache(Duration = 30)]
public class HomeController : Controller
{
    [ResponseCache(CacheProfileName = "Default")]
    public IActionResult Index()
    {
        return View();
    }
```

The `ResponseCache` attribute can be applied both to actions (methods) as well as controllers (classes). Method-level attributes will override the settings specified in class-level attributes.

In the above example, a class-level attribute specifies a duration of 30 seconds while a method-level attributes references a cache profile with a duration set to 60 seconds.

The resulting header:

```
Cache-Control: public,max-age=60
```

Additional Resources

Caching in HTTP from the specification
Cache-Control

Response Caching Middleware

By Luke Latham and John Luo

View or download sample code

This document provides details on how to configure the Response Caching Middleware in ASP.NET Core applications. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the `ResponseCache` attribute, see Response Caching.

Package

To include the middleware in your project, add a reference to the `Microsoft.AspNetCore.ResponseCaching` package. The middleware depends on .NET Framework 4.5.1 or .NET Standard 1.3 or later. This feature is available for apps that target ASP.NET Core 1.1.0 or later.

Configuration

In `ConfgureServices`, add the middleware to your service collection.

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

Configure the application to use the middleware when processing requests. The sample application adds a `Cache-Control` header to the response that will cache cacheable responses for up to 10 seconds. The sample also sends a `Vary` header to configure the cache to serve the response only if the `Accept-Encoding` header of subsequent requests matches that from the original request.

```
public void Configure(IApplicationBuilder app)
{
    app.UseResponseCaching();
    app.Run(async (context) =>
    {
        context.Response.GetTypedHeaders().CacheControl = new CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
        context.Response.Headers[HeaderNames.Vary] = new string[] { "Accept-Encoding" };

        await context.Response.WriteAsync("Hello World! " + DateTime.UtcNow);
    });
}
```

The Response Caching Middleware only caches 200 (OK) server responses. Any other responses, including error pages, will be ignored by the middleware.

Warning

Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See Conditions for caching for details on how the middleware determines if a response is cacheable.

Options

The middleware offers two options for controlling response caching.

OPTION	DEFAULT VALUE
UseCaseSensitivePaths	Determines if responses will be cached on case-sensitive paths. The default value is `false`.
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` [64 MB (67,108,864 bytes)].

The following example configures the middleware to cache responses smaller than or equal to 1,024 bytes using case-sensitive paths, storing the responses to `/page1` and `/Page1` separately.

```
services.AddResponseCaching(options =>
{
    options.UseCaseSensitivePaths = true;
    options.MaximumBodySize = 1024;
});
```

VaryByQueryKeys feature

When using MVC, the `ResponseCache` attribute specifies the parameters necessary for setting appropriate headers for response caching. The only parameter of the `ResponseCache` attribute that strictly requires the middleware is `VaryByQueryKeys`, which does not correspond to an actual HTTP header. For more information, see ResponseCache Attribute.

When not using MVC, you can vary response caching with the `VaryByQueryKeys` feature by using the `ResponseCachingFeature` directly from the `IFeatureCollection` of the `HttpContext`.

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();
if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

HTTP headers used by Response Caching Middleware

Response caching by the middleware is configured via your HTTP response headers. The relevant headers are listed below with notes on how they affect caching.

HEADER	DETAILS
Authorization	The response is not cached if the header exists.

HEADER	DETAILS
Cache-Control	The middleware will only consider caching responses explicitly marked with the `public` cache directive. You can control caching with the following parameters: • max-age • max-stale • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate For more information, see [RFC 7231: Request Cache-Control Directives].
Pragma	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header if present. Considered for backward compatibility with HTTP/1.0.
Set-Cookie	The response is not cached if the header exists.
Vary	You can vary the cached response by another header. For example, you can cache responses by encoding by including the `Vary: Accept-Encoding` header, which would cache responses to requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
Expires	A response deemed stale by this header will not be stored or retrieved unless overridden by other `Cache-Control` headers.
If-None-Match	The full response will be served from cache if the value is not `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response will be served.
If-Modified-Since	If the `If-None-Match` header is not present, a full response will be served from cache if the cached response date is newer than the value provided. Otherwise, a 304 (Not Modified) response will be served.
Date	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
Content-Length	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.

HEADER	DETAILS
Age	The `Age` header sent in the original response will be ignored. The middleware will compute a new value when serving a cached response.

Troubleshooting

If caching behavior is not as you expect, confirm that responses are cacheable and capable of being served from the cache by examining the request's incoming headers and the response's outgoing headers. The conditions by which a response will be cached are listed below.

Enabling logging can help when debugging. For example, the middleware logs why a response is or is not cached and whether it was retrieved from cache. See Logging in ASP.NET Core for more information on enabling logging in your application.

When testing and troubleshooting caching behavior, a browser may set request headers that affect caching in undesirable ways. For example, a browser may set the `Cache-Control` header to `no-cache` when you refresh the page. Instead of using a browser, use a tool like Fiddler, Firebug, or Postman, all of which allow you to explicitly set request headers.

Conditions for caching

The request must result in a 200 (OK) response from the server.
The request method must be GET or HEAD.
Terminal middleware, such as Static File Middleware, must not process the response prior to the Response Caching Middleware.
The `Authorization` header must not be present.
`Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
The `Pragma: no-cache` header/value must not be present if the `Cache-Control` header is not present, as the `Cache-Control` header overrides the `Pragma` header when present.
The `Set-Cookie` header must not be present.
`Vary` header parameters must be valid and not equal to `*`.
The `Content-Length` header value (if set) must match the size of the response body.
The `HttpSendFileFeature` is not used.
The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
Response buffering is successful, and the total length of the response is smaller than the configured limit.
The response must be cacheable according to the RFC 7234 specifications. For example, the `no-store` directive must not exist in request or response header fields. See *Section 3: Storing Responses in Caches* of the RFC document for details.

Note

The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks will set the `Cache-Control` and `Pragma` headers to `no-cache` so that responses will not be cached.

Additional resources

Application Startup
Middleware

Response Compression Middleware

By Luke Latham

View or download sample code

Network bandwidth is a limited resource. If you can reduce response payload sizes and thus send less data to clients, you can usually increase the responsiveness of your application, sometimes dramatically. One way to reduce payload sizes is to compress an application's responses.

When to use Response Compression Middleware

Use Response Compression Middleware when you're unable to use the Dynamic Compression module in IIS, the Apache mod_deflate module, NGINX Compression and Decompression, or your application is hosted directly on WebListener server or Kestrel. The main reason to use the server-based response compression technologies in IIS, Apache, or Nginx is that the performance of the middleware probably won't match that of the server modules.

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include: CSS, JavaScript, HTML, XML, and JSON. You shouldn't compress natively compressed assets, such as PNG files. If you attempt to further compress a natively compressed response, any small additional reduction in size and transmission time will likely be overshadowed by the time it took to process the compression. You also shouldn't compress files smaller than about 150-1000 bytes (depends on the file's content and the efficiency of compression), as doing so may produce a compressed file larger than the file itself.

When a client can process compressed content, the client must inform the server of its capabilities by sending the `Accept-Encoding` header with the request. When a server sends compressed content, it must include information in the `Content-Encoding` header on how the compressed response is encoded. Content encoding designations are shown below indicating which ones are supported by the middleware.

`ACCEPT-ENCODING` HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
`br`	No	Brotli Compressed Data Format
`compress`	No	UNIX "compress" data format
`deflate`	No	"deflate" compressed data inside the "zlib" data format
`exi`	No	W3C Efficient XML Interchange
`gzip`	Yes (default)	Gzip file format
`identity`	Yes	"No encoding" identifier: The response must not be encoded.
`pack200-gzip`	No	Network Transfer Format for Java Archives
`*`	Yes	Any available content encoding not explicitly requested

For more information, see the IANA Official Content Coding List.

The middleware will allow you to add additional compression providers for custom `Accept-Encoding` header values. For more information, see Custom Providers below.

The middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more information, see RFC 7231: Accept-Encoding.

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the most *optimal* compression. The Gzip compression provider defaults to the fastest compression level, which might not produce the most efficient compression. If the most efficient compression is desired, the middleware can be configured for optimal compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described below.

HEADER	ROLE
`Accept-Encoding`	Sent by the client to the server to indicate which types of content encoding are acceptable.
`Content-Encoding`	Sent from the server to the client to indicate the encoding of the content in the payload.
`Content-Length`	When compression occurs, the `Content-Length` header is removed, since the body content changes when the response is compressed.
`Content-MD5`	When compression occurs, the `Content-MD5` header is removed, since the body content has changed and the hash is no longer valid.
`Content-Type`	Specifies the MIME type of the content. Each response should have a content-type. The middleware will check this value to determine if the response should be compressed. The middleware includes a set of default MIME types that it will encode, but you can replace or add MIME types.
`Vary`	When sent by the server with a value of `Accept-Encoding` to clients and proxies, it indicates that they should cache (vary) responses based on the value of the `Accept-Encoding` header of the request. The result of returning content with the `Vary: Accept-Encoding` header is that both compressed and uncompressed responses will be cached separately.

You can explore the features of the Response Compression Middleware with the sample application. The sample illustrates the compression of application responses using Gzip and custom compression providers. It also shows you how to add a MIME type to the default list of MIME types for compression.

Package

To include the middleware in your project, add a reference to the `Microsoft.AspNetCore.ResponseCompression` package. The middleware depends on .NET Framework 4.5.1 or .NET Standard 1.3 or later. This feature is available for apps that target ASP.NET Core 1.1.0 or later.

Configuration

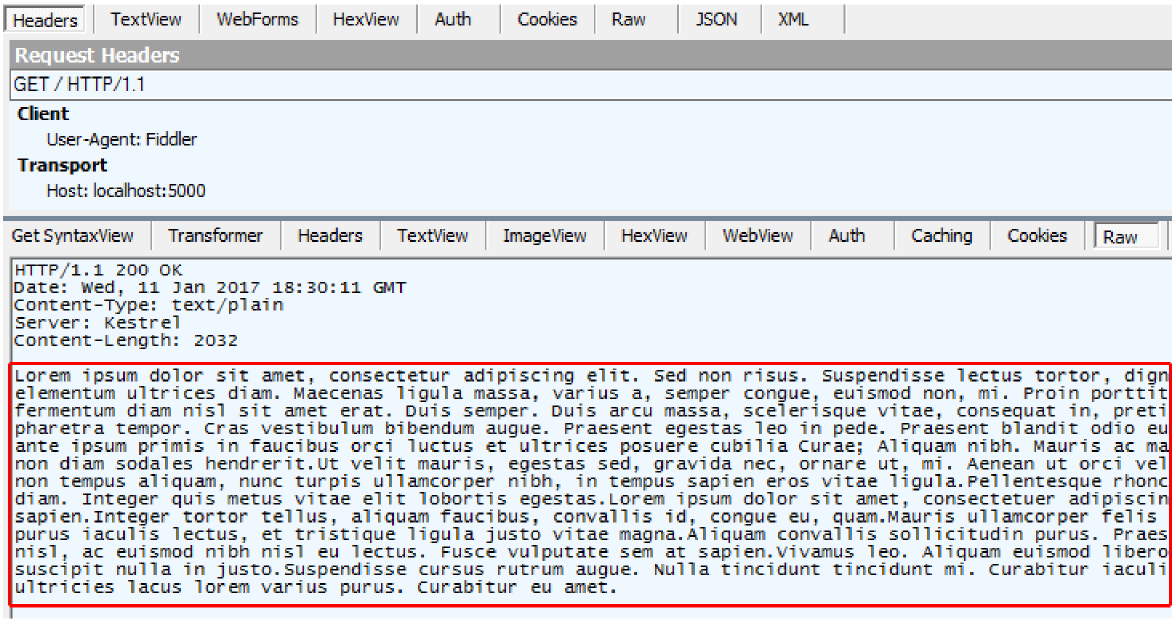
The following highlighted code shows how to enable the Response Compression Middleware with the with the default Gzip compression and for default MIME types.

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

public void Configure(IApplicationBuilder app)
{
    app.UseResponseCompression();

    app.Run(async context =>
    {
        // If the Accept-Encoding header is present, always add the Vary header
        // This will be added as a feature in the next release of the middleware.
        // https://github.com/aspnet/BasicMiddleware/issues/187
        var accept = context.Request.Headers[HeaderNames.AcceptEncoding];
        if (!StringValues.IsNullOrEmpty(accept))
        {
            context.Response.Headers.Append(HeaderNames.Vary, HeaderNames.AcceptEncoding);
        }
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync(LoremIpsum.Text);
    });
}
```

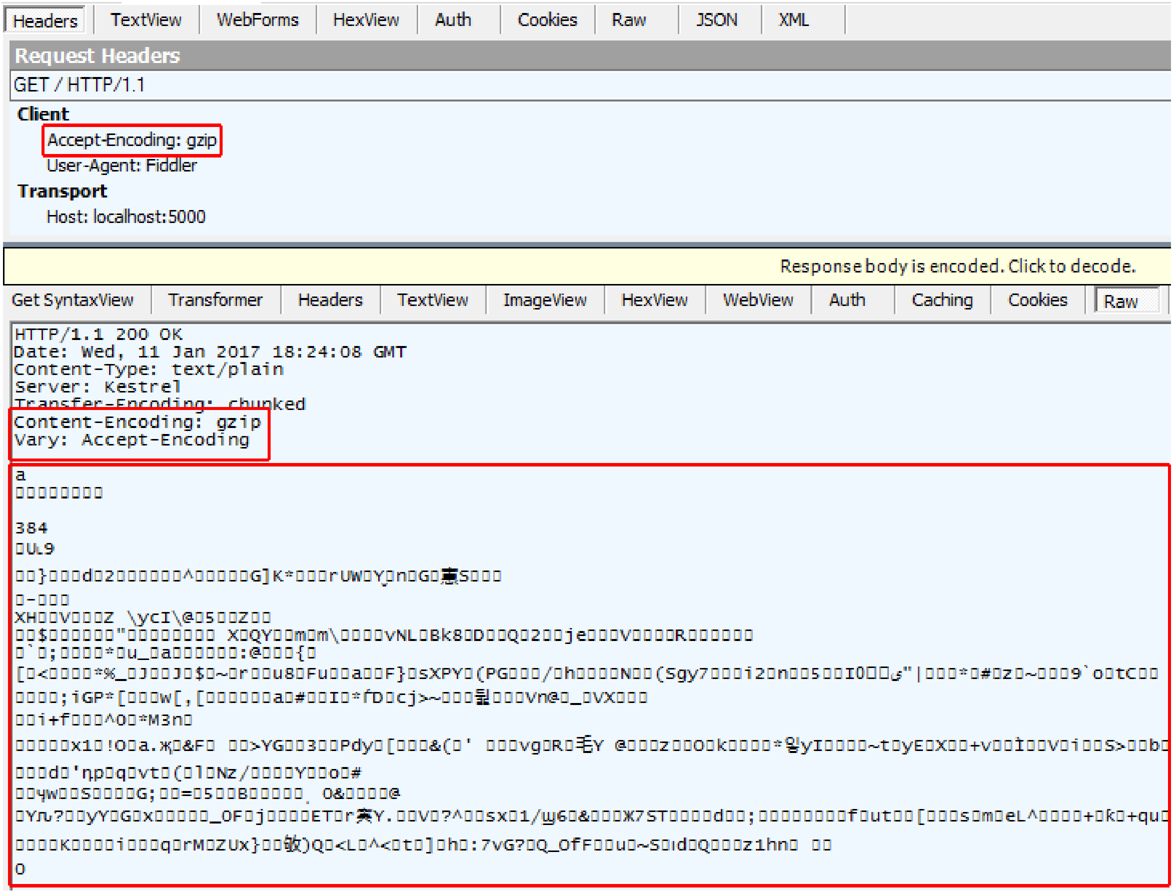
Note

Use a tool like Fiddler, Firebug, or Postman to set the `Accept-Encoding` request header and study the response headers, size, and body.

Submit a request to the sample application without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` and `Vary` headers are not present on the response.



Submit a request to the sample application with the `Accept-Encoding: gzip` header and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.



Providers

GzipCompressionProvider

Use the `GzipCompressionProvider` to compress responses with Gzip. This is the default compression provider if none are specified. You can set the compression level with the `GzipCompressionProviderOptions`. The default is `CompressionLevel.Fastest`.

COMPRESSION LEVEL	DESCRIPTION
`CompressionLevel.Fastest`	Compression should complete as quickly as possible, even if the resulting output is not optimally compressed.
`CompressionLevel.NoCompression`	No compression should be performed.
`CompressionLevel.Optimal`	Responses should be optimally compressed, even if the compression takes more time to complete.

```
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 types

The middleware specifies a default set of MIME types for compression:

`text/plain`
`text/css`
`application/javascript`
`text/html`
`application/xml`
`text/xml`
`application/json`
`text/json`

You can replace or append MIME types with the Response Compression Middleware options. Note that wildcard MIME types, such as `text/*` are not supported. The sample application adds a MIME type for `image/svg+xml` and will compress and serve the ASP.NET Core banner image (*banner.svg*).

```
services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});
```

Custom providers

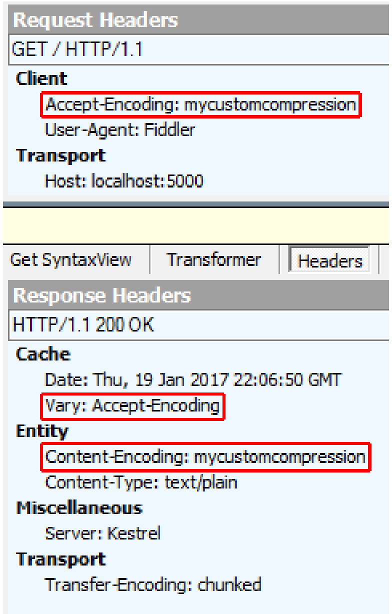
You can create custom compression implementations with `ICompressionProvider`. The `EncodingName` represents the content encoding that this `ICompressionProvider` produces. The middleware will use this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

Using the sample application, the client would submit a request with the `Accept-Encoding: mycustomcompression` header. The middleware will use the custom compression implementation and return the response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

```
services.AddResponseCompression(options =>
{
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(new[] { "image/svg+xml" });
});
```

```
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;
    }
}
```

Submit a request to the sample application with the `Accept-Encoding: mycustomcompression` header and observe the response headers. The `Vary` and `Content-Encoding` headers are present on the response. The response body (not shown) isn't compressed by the sample, as there is no compression implementation in the `CustomCompressionProvider` class of the sample. However, the sample shows where you would implement such a compression algorithm.



Compression with secure protocol

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default. Using compression with dynamically generated pages can lead to security problems such as the CRIME and BREACH attacks.

Adding the Vary header

When compressing responses based on the `Accept-Encoding` header, there are potentially multiple compressed versions of the response and an uncompressed version. In order to instruct client and proxy caches that multiple versions exist and should be stored, you should supply a `Vary` header with an `Accept-Encoding` value. The sample application adds a `Vary` header using a method; however, the middleware will be upgraded soon to provide this feature (Basic Middleware #187).

```
private void ManageVaryHeader(HttpContext context)
{
    // If the Accept-Encoding header is present, always add the Vary header
    // This will be added as a feature in the next release of the middleware.
    // https://github.com/aspnet/BasicMiddleware/issues/187
    var accept = context.Request.Headers[HeaderNames.AcceptEncoding];
    if (!StringValues.IsNullOrEmpty(accept))
    {
        context.Response.Headers.Append(HeaderNames.Vary, HeaderNames.AcceptEncoding);
    }
}
```

Middlware issue when behind an Nginx reverse-proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. This prevents the middleware from compressing the response. For more information, see NGINX: Compression and Decompression. This issue is tracked by Figure out pass-through compression for nginx (BasicMiddleware #123).

Working with IIS dynamic compression

If you have an active IIS Dynamic Compression Module configured at the server level that you would like to disable for an application, you can do so with an addition to your *web.config* file. For more information, see Disabling IIS modules.

Troubleshooting

Use a tool like Fiddler, Firebug, or Postman, which allow you to set the `Accept-Encoding` request header and study the response headers, size, and body. The Response Compression Middleware will compress responses that meet the following conditions:

The `Accept-Encoding` header is present with a value of `gzip`, `*`, or custom encoding that matches a custom compression provider that you've established. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).
The MIME type (`Content-Type`) must be set and must match a MIME type configured on the `ResponseCompressionOptions`.
The request must not include the `Content-Range` header.
The request must use *insecure protocol (http)*, unless secure protocol (https) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Additional resources

Application Startup
Middleware
Mozilla Developer Network: Accept-Encoding
RFC 7231 Section 3.1.2.1: Content Codings
RFC 7230 Section 4.2.3: Gzip Coding
GZIP file format specification version 4.3

Migrating From ASP.NET MVC to ASP.NET Core MVC

By Rick Anderson, Daniel Roth, Steve Smith, and Scott Addie

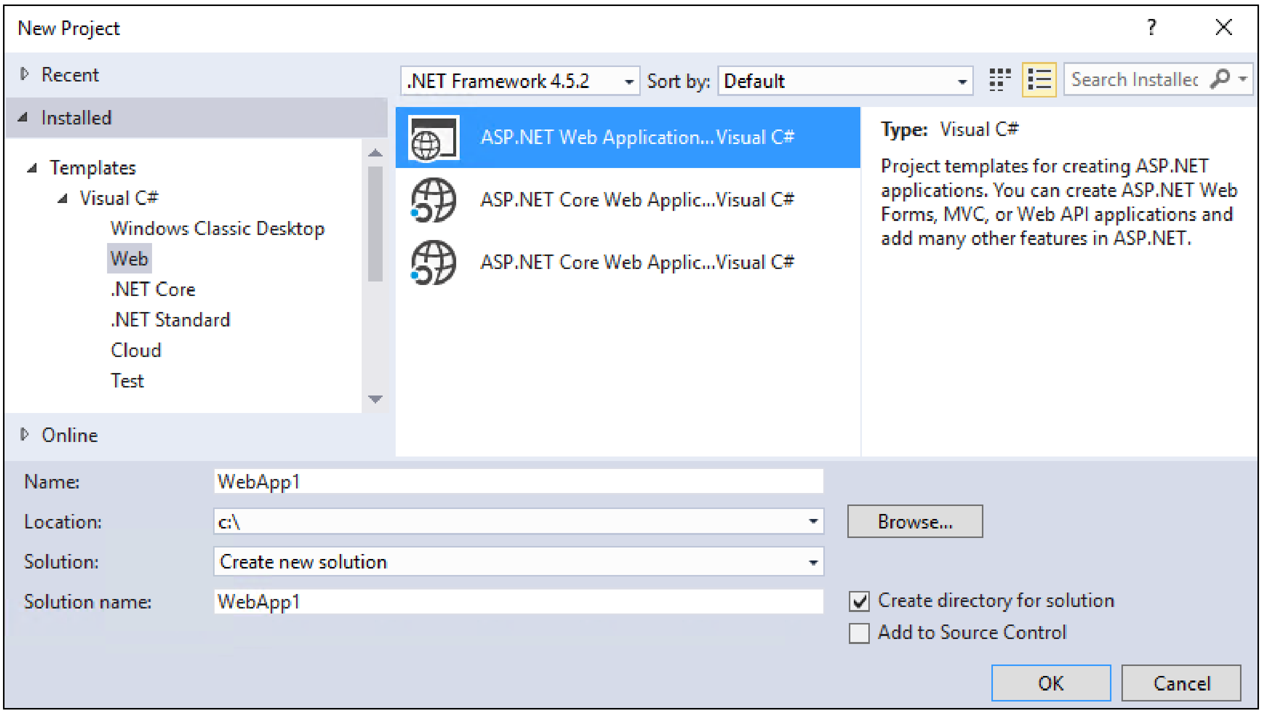
This article shows how to get started migrating an ASP.NET MVC project to ASP.NET Core MVC. In the process, it highlights many of the things that have changed from ASP.NET MVC. Migrating from ASP.NET MVC is a multiple step process and this article covers the initial setup, basic controllers and views, static content, and client-side dependencies. Additional articles cover migrating configuration and identity code found in many ASP.NET MVC projects.

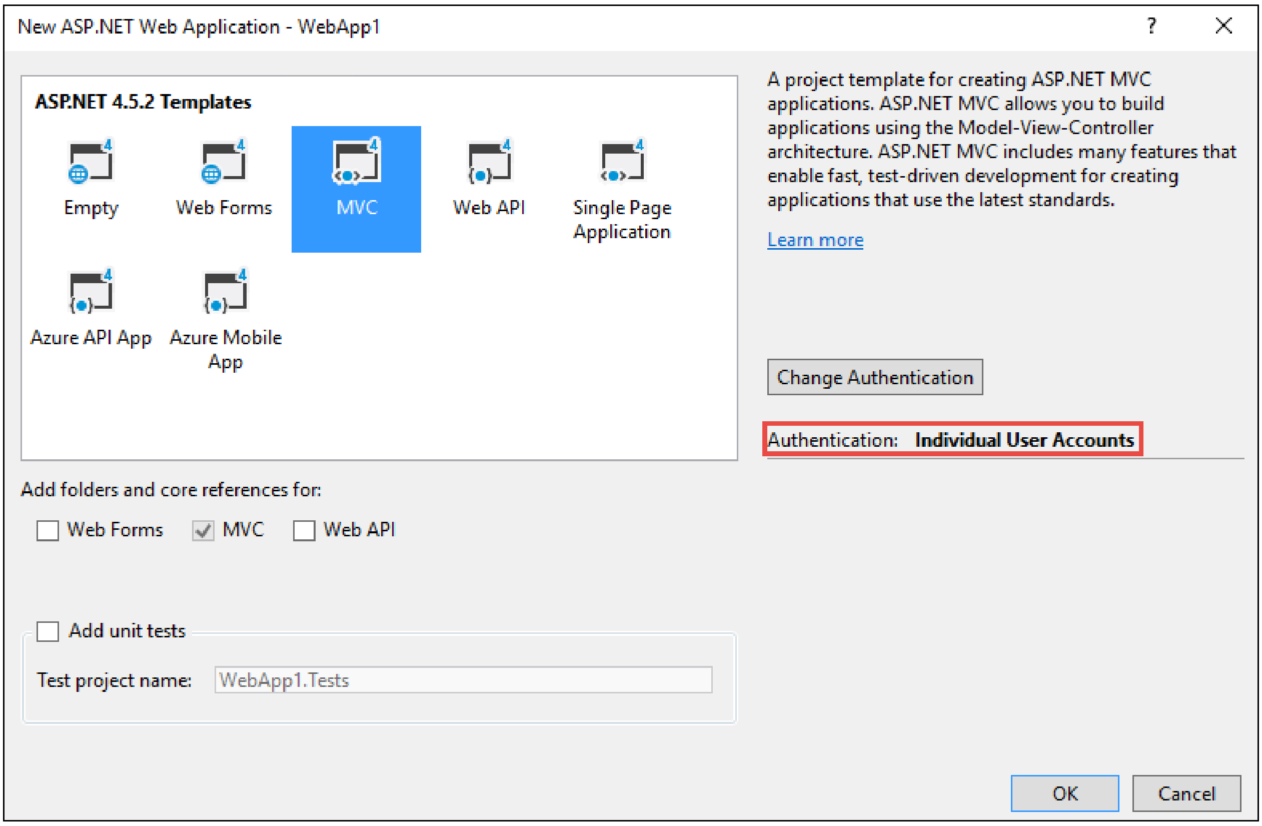
Note

The version numbers in the samples might not be current. You may need to update your projects accordingly.

Create the starter ASP.NET MVC project

To demonstrate the upgrade, we'll start by creating a ASP.NET MVC app. Create it with the name *WebApp1* so the namespace will match the ASP.NET Core project we create in the next step.

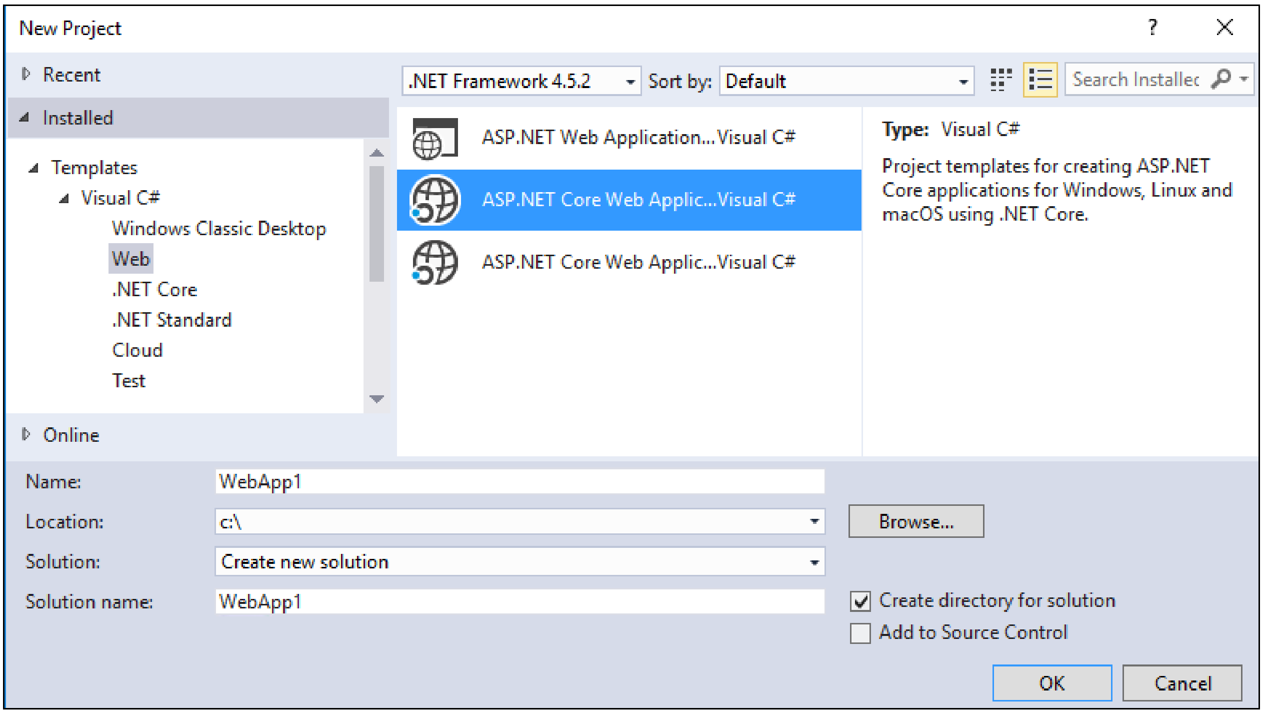


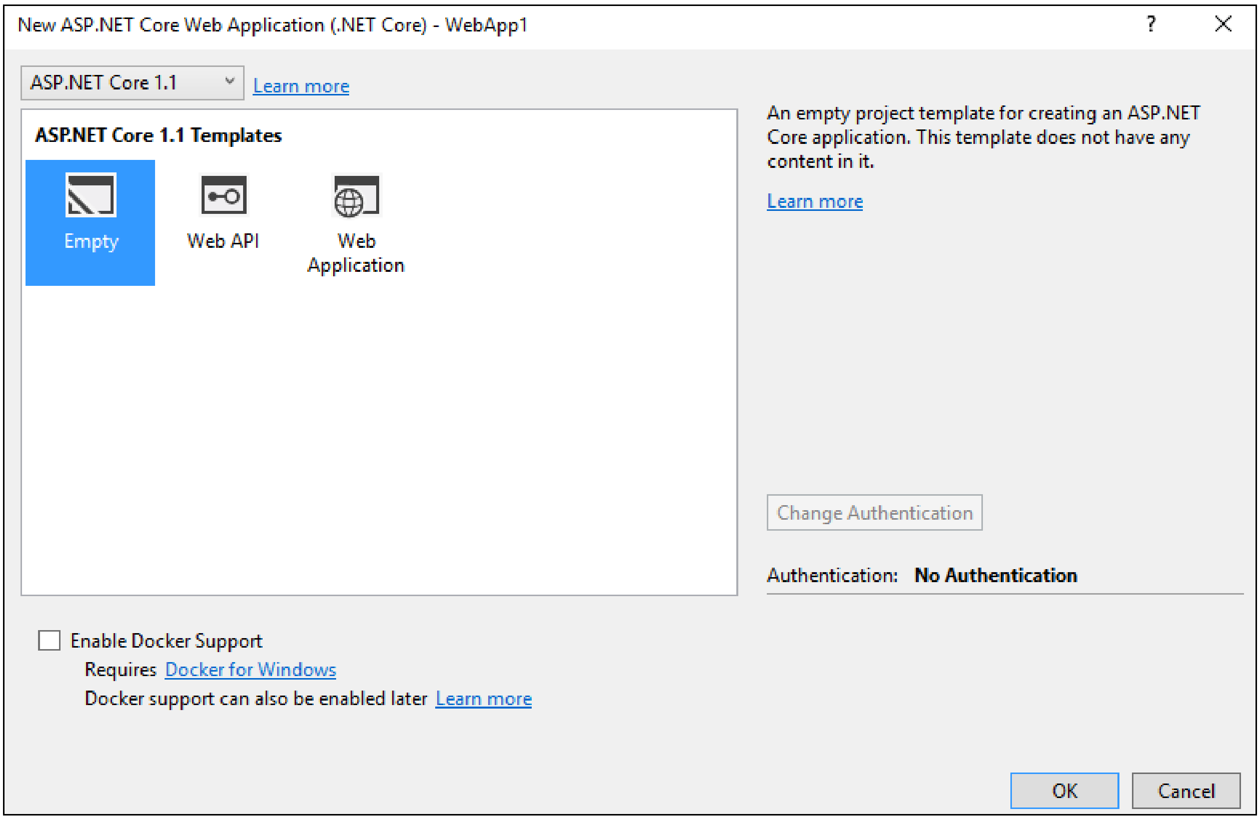


Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio will display the new solution name (*Mvc5*), which will make it easier to tell this project from the next project.

Create the ASP.NET Core project

Create a new *empty* ASP.NET Core web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. You'll have to create this project in a different directory than the previous project to use the same name.





Optional: Create a new ASP.NET Core app using the *Web Application* project template. Name the project *WebApp1*, and select an authentication option of **Individual User Accounts**. Rename this app to *FullAspNetCore*. Creating this project will save you time in the conversion. You can look at the template-generated code to see the end result or to copy code to the conversion project. It's also helpful when you get stuck on a conversion step to compare with the template-generated project.

Configure the site to use MVC

Install the `Microsoft.AspNetCore.Mvc` and `Microsoft.AspNetCore.StaticFiles` NuGet packages.

`Microsoft.AspNetCore.Mvc` is the ASP.NET Core MVC framework. `Microsoft.AspNetCore.StaticFiles` is the static file handler. The ASP.NET runtime is modular, and you must explicitly opt in to serve static files (see Working with Static Files).

Open the *.csproj* file (right-click the project in **Solution Explorer** and select **Edit WebApp1.csproj**) and add a `PrepareForPublish` target:

```
<Target Name="PrepublishScript" BeforeTargets="PrepareForPublish">
  <Exec Command="bower install" />
</Target>
```

The `PrepareForPublish` target is needed for acquiring client-side libraries via Bower. We'll talk about that later.

Open the *Startup.cs* file and change the code to match the following:

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

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, ILoggerFactory loggerFactory)
        {
            loggerFactory.AddConsole();

            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();

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

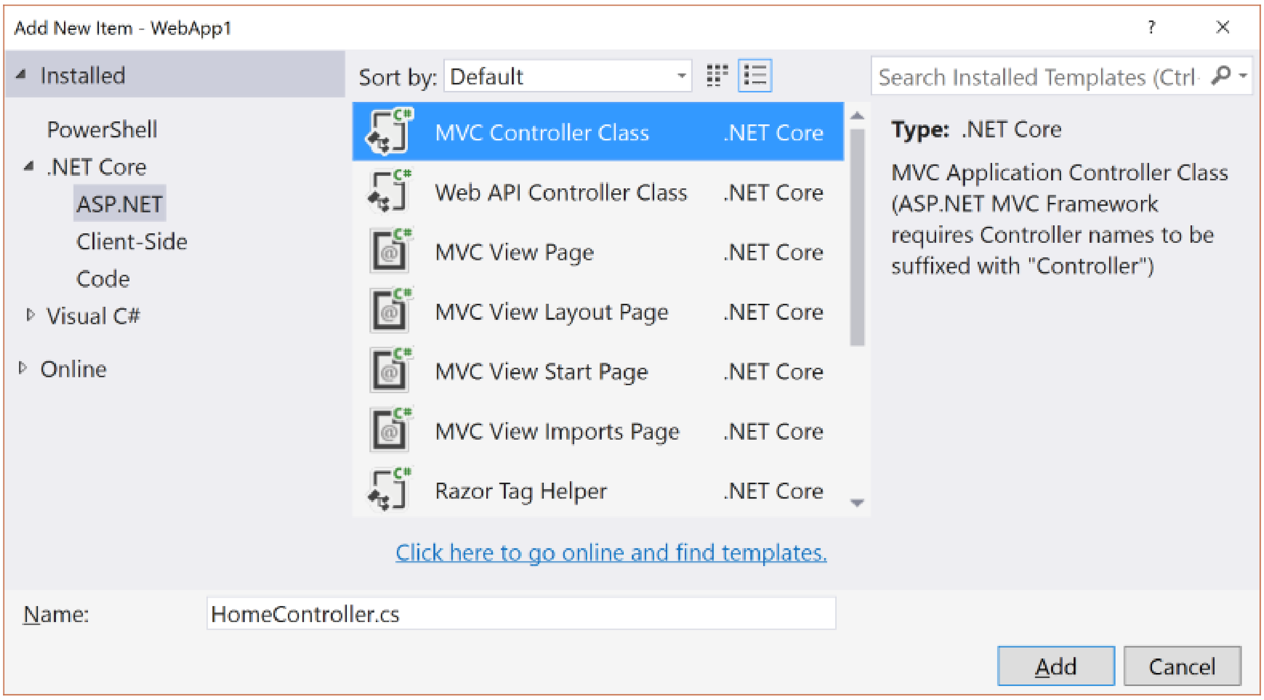
The `UseStaticFiles` extension method adds the static file handler. As mentioned previously, the ASP.NET runtime is modular, and you must explicitly opt in to serve static files. The `UseMvc` extension method adds routing. For more information, see Application Startup and Routing.

Add a controller and view

In this section, you'll add a minimal controller and view to serve as placeholders for the ASP.NET MVC controller and views you'll migrate in the next section.

Add a *Controllers* folder.

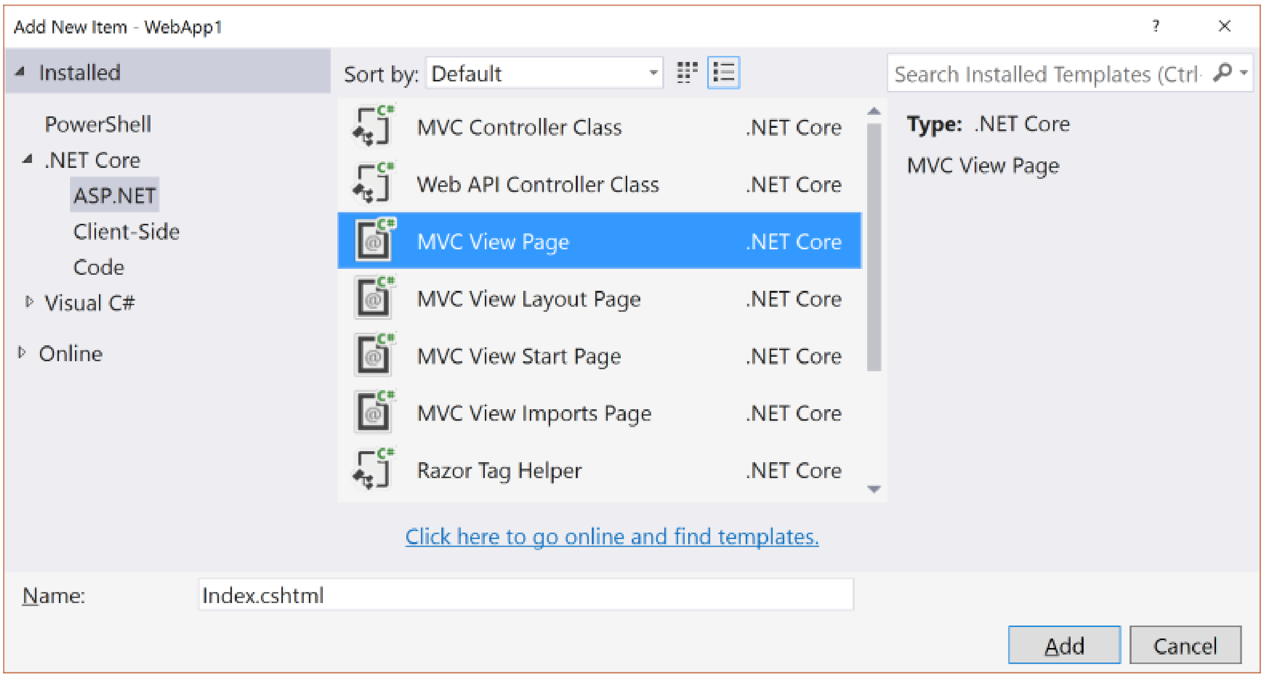
Add an **MVC controller class** with the name *HomeController.cs* to the *Controllers* folder.



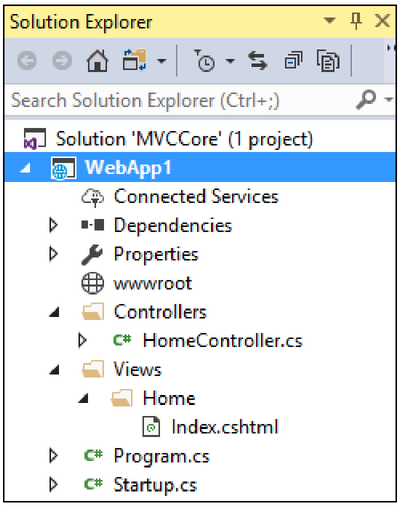
Add a *Views* folder.

Add a *Views/Home* folder.

Add an *Index.cshtml* MVC view page to the *Views/Home* folder.



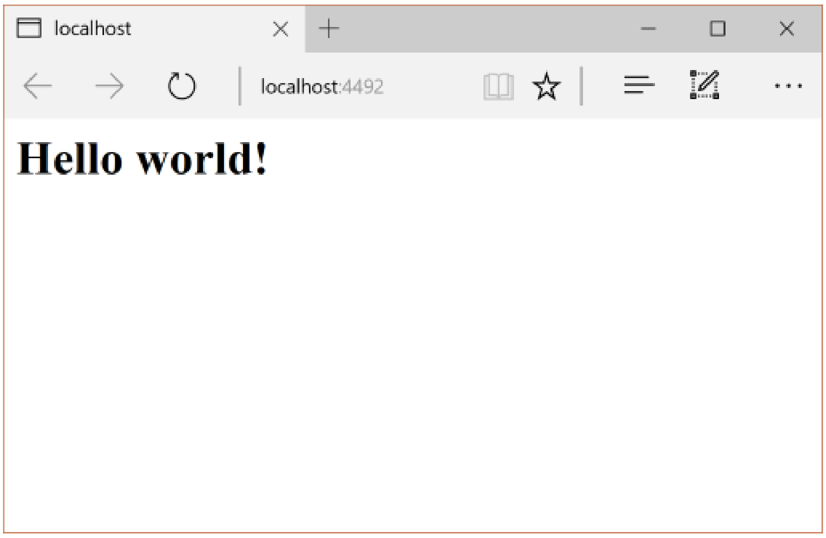
The project structure is shown below:



Replace the contents of the *Views/Home/Index.cshtml* file with the following:

```
<h1>Hello world!</h1>
```

Run the app.



See Controllers and Views for more information.

Now that we have a minimal working ASP.NET Core project, we can start migrating functionality from the ASP.NET MVC project. We will need to move the following:

client-side content (CSS, fonts, and scripts)

controllers

views

models

bundling

filters

Log in/out, identity (This will be done in the next tutorial.)

Controllers and views

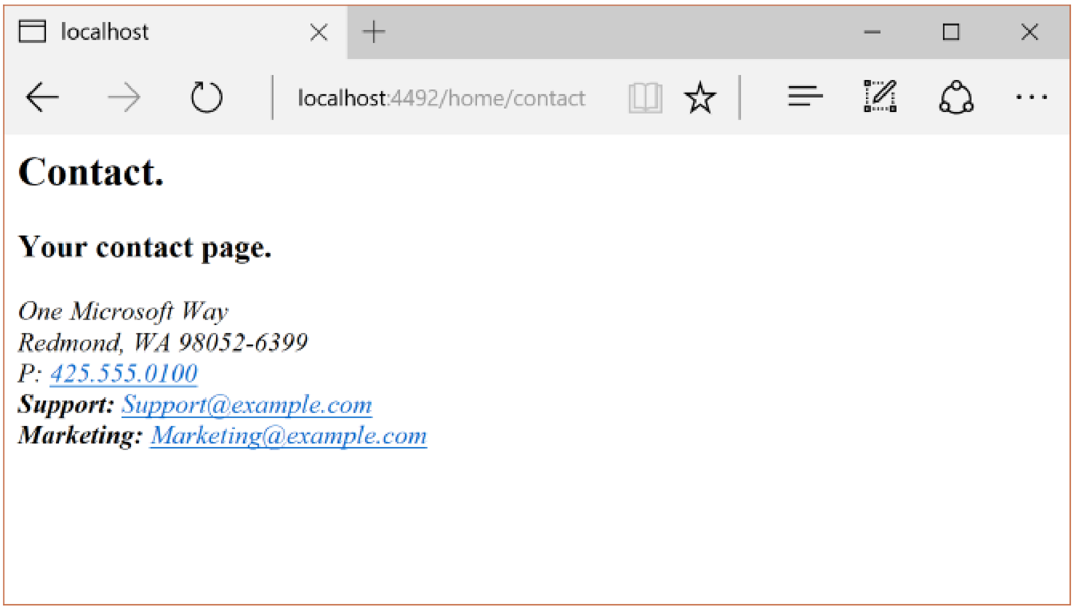
Copy each of the methods from the ASP.NET MVC `HomeController` to the new `HomeController`. Note that in ASP.NET MVC, the built-in template's controller action method return type is ActionResult; in ASP.NET Core MVC, the action methods return `IActionResult` instead. `ActionResult` implements `IActionResult`, so there is no need to change the return type of your action methods.

Copy the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files from the ASP.NET MVC project to the ASP.NET Core project.

Run the ASP.NET Core app and test each method. We haven't migrated the layout file or styles yet, so the rendered views will only contain the content in the view files. You won't have the layout file generated links for the `About` and `Contact` views, so you'll have to invoke them from the browser (replace **4492** with the port number used in your project).

`http://localhost:4492/home/about`

`http://localhost:4492/home/contact`



Note the lack of styling and menu items. We'll fix that in the next section.

Static content

In previous versions of ASP.NET MVC, static content was hosted from the root of the web project and was intermixed with server-side files. In ASP.NET Core, static content is hosted in the *wwwroot* folder. You'll want to copy the static content from your old ASP.NET MVC app to the *wwwroot* folder in your ASP.NET Core project. In this sample conversion:

Copy the *favicon.ico* file from the old MVC project to the *wwwroot* folder in the ASP.NET Core project.

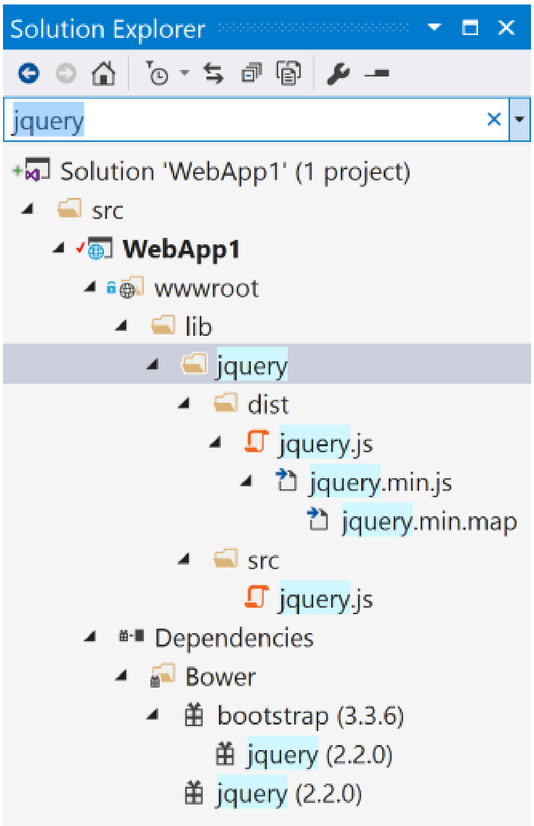
The old ASP.NET MVC project uses Bootstrap for its styling and stores the Bootstrap files in the *Content* and *Scripts* folders. The template, which generated the old ASP.NET MVC project, references Bootstrap in the layout file (*Views/Shared/_Layout.cshtml*). You could copy the *bootstrap.js* and *bootstrap.css* files from the ASP.NET MVC project to the *wwwroot* folder in the new project, but that approach doesn't use the improved mechanism for managing client-side dependencies in ASP.NET Core.

In the new project, we'll add support for Bootstrap (and other client-side libraries) using Bower:

Add a Bower configuration file named *bower.json* to the project root (Right-click on the project, and then **Add > New Item > Bower Configuration File**). Add Bootstrap and jQuery to the file (see the highlighted lines below).

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "2.2.0"
  }
}
```

Upon saving the file, Bower will automatically download the dependencies to the *wwwroot/lib* folder. You can use the **Search Solution Explorer** box to find the path of the assets:



See Manage Client-Side Packages with Bower for more information.

Migrate the layout file

Copy the *_ViewStart.cshtml* file from the old ASP.NET MVC project's *Views* folder into the ASP.NET Core project's *Views* folder. The *_ViewStart.cshtml* file has not changed in ASP.NET Core MVC.

Create a *Views/Shared* folder.

Optional: Copy *_ViewImports.cshtml* from the *FullAspNetCore* MVC project's *Views* folder into the ASP.NET Core project's *Views* folder. Remove any namespace declaration in the *_ViewImports.cshtml* file. The *_ViewImports.cshtml* file provides namespaces for all the view files and brings in Tag Helpers. Tag Helpers are used in the new layout file. The *_ViewImports.cshtml* file is new for ASP.NET Core.

Copy the *_Layout.cshtml* file from the old ASP.NET MVC project's *Views/Shared* folder into the ASP.NET Core project's *Views/Shared* folder.

Open *_Layout.cshtml* file and make the following changes (the completed code is shown below):

Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load *bootstrap.css* (see below).

Remove `@Scripts.Render("~/bundles/modernizr")`.

Comment out the `@Html.Partial("_LoginPartial")` line (surround the line with `@*...*@`). We'll return to it in a future tutorial.

Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below).

Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below)..

The replacement CSS link:

```
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
```

The replacement script tags:

```
<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
```

The updated *_Layout.cshtml* file is shown below:

```
<!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="~/lib/bootstrap/dist/css/bootstrap.css" />

</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>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

View the site in the browser. It should now load correctly, with the expected styles in place.

Optional: You might want to try using the new layout file. For this project you can copy the layout file from the *FullAspNetCore* project. The new layout file uses Tag Helpers and has other improvements.

Configure Bundling & Minification

For information about how to configure bundling and minification, see Bundling and Minification.

Solving HTTP 500 errors

There are many problems that can cause a HTTP 500 error message that contain no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in your project, you'll get a HTTP 500 error. To get a detailed error message, add the following code:

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

    app.UseStaticFiles();

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

See **Using the Developer Exception Page** in Error Handling for more information.

Additional Resources

Client-Side Development

Tag Helpers

Migrating Configuration

By Steve Smith and Scott Addie

In the previous article, we began migrating an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate configuration.

View or download sample code

Setup Configuration

ASP.NET Core no longer uses the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC, a *Startup.cs* file was included in the root of the project; and, it was called when the application started. ASP.NET Core has adopted this approach completely by placing all startup logic in the *Startup.cs* file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in *Startup.cs*. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as *appsettings.json*. ASP.NET Core's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that should not be checked into source control. See Configuration to learn more about configuration in ASP.NET Core.

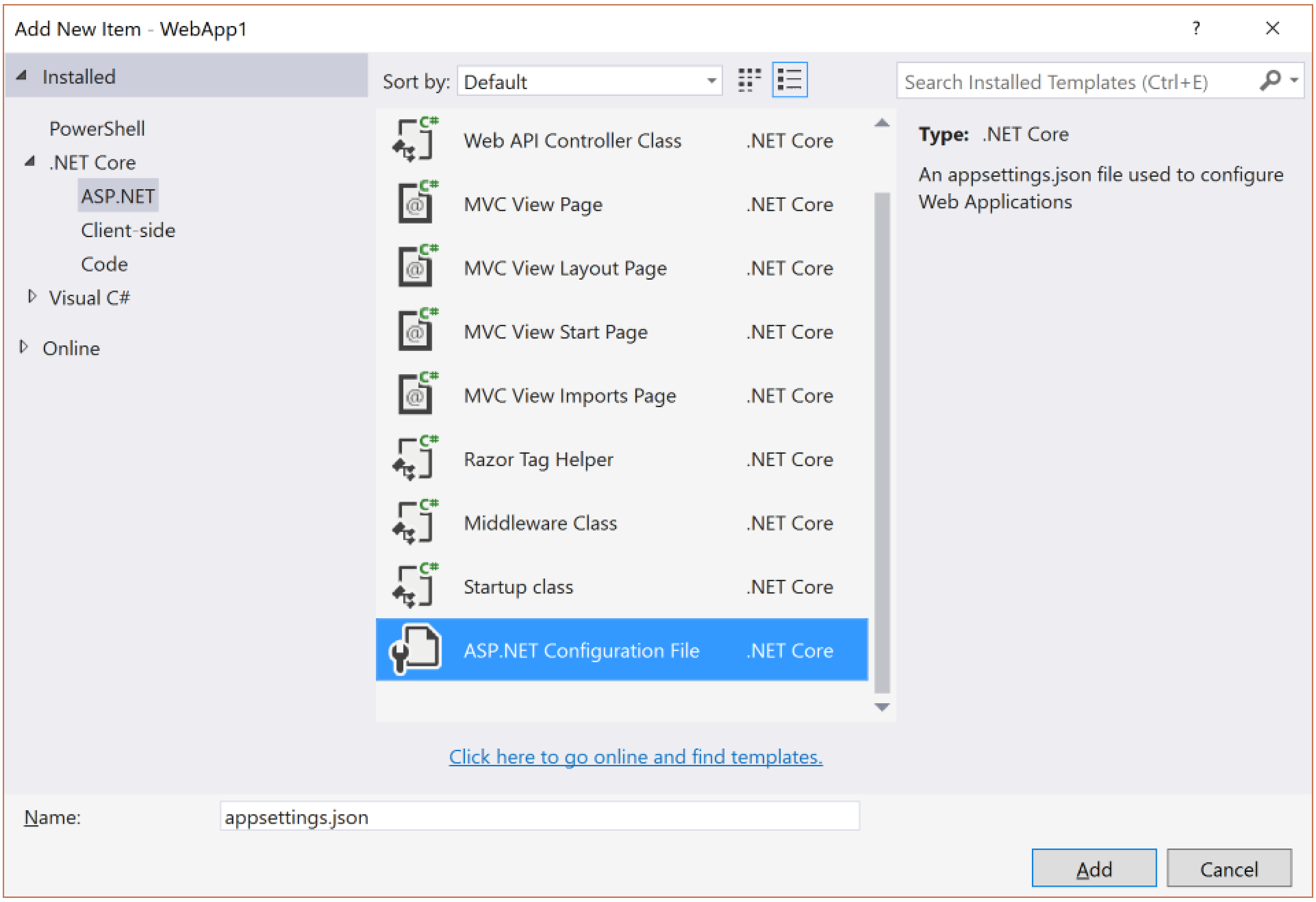
For this article, we are starting with the partially-migrated ASP.NET Core project from the previous article. To setup configuration, add the following constructor and property to the *Startup.cs* file located in the root of the project:

```
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();
}

public IConfigurationRoot Configuration { get; }
```

Note that at this point, the *Startup.cs* file will not compile, as we still need to add the following `using` statement:

```
using Microsoft.Extensions.Configuration;
```

Add an *appsettings.json* file to the root of the project using the appropriate item template:



Migrate Configuration Settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the *appsettings.json* file. Open *appsettings.json*, and note that it already includes the following:

```
{
 "Data": {
  "DefaultConnection": {
   "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"
  }
 }
}
```

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

Migrating Authentication and Identity

By Steve Smith

In the previous article we migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate the registration, login, and user management features.

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in Startup.Auth.cs and IdentityConfig.cs, located in the App_Start folder. In ASP.NET Core MVC, these features are configured in *Startup.cs*.

Install the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` and `Microsoft.AspNetCore.Authentication.Cookies` NuGet packages.

Then, open Startup.cs and update the `ConfigureServices()` method to use Entity Framework and Identity services:

```
public void ConfigureServices(IServiceCollection services)
{
  // Add EF services to the services container.
  services.AddEntityFramework(Configuration)
    .AddSqlServer()
    .AddDbContext<ApplicationDbContext>();

  // Add Identity services to the services container.
  services.AddIdentity<ApplicationUser, IdentityRole>(Configuration)
    .AddEntityFrameworkStores<ApplicationDbContext>();

  services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven't yet migrated from the ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new *Models* folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC versions of these classes in `/Models/IdentityModels.cs`, but we will use one file per class in the migrated project since that's more clear.

ApplicationUser.cs:

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

namespace NewMvc6Project.Models
{
  public class ApplicationUser : IdentityUser
  {
  }
}
```

ApplicationDbContext.cs:

```
using Microsoft.AspNetCore.Identity.EntityFramework;
using Microsoft.Data.Entity;

namespace NewMvc6Project.Models
{
  public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
  {
    public ApplicationDbContext()
    {
      Database.EnsureCreated();
    }

    protected override void OnConfiguring(DbContextOptions options)
    {
      options.UseSqlServer();
    }
  }
}
```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the ApplicationDbContext. When migrating a real application, you will also need to migrate all of the custom properties and methods of your application's user and DbContext classes, as well as any other Model classes your application utilizes (for example, if your DbContext has a DbSet, you will of course need to migrate the Album class).

With these files in place, the Startup.cs file can be made to compile by updating its using statements:

```
using Microsoft.Framework.ConfigurationModel;
using Microsoft.AspNetCore.Hosting;
using NewMvc6Project.Models;
using Microsoft.AspNetCore.Identity;
```

Our application is now ready to support authentication and identity services - it just needs to have these features exposed to users.

Migrate Registration and Login Logic

With identity services configured for the application and data access configured using Entity Framework and SQL Server, we are now ready to add support for registration and login to the application. Recall that earlier in the migration process we commented out a reference to _LoginPartial in _Layout.cshtml. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Update _Layout.cshtml; uncomment the @Html.Partial line:

```
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
    @*@Html.Partial("_LoginPartial")*@
  </div>
</div>
```

Now, add a new MVC View Page called _LoginPartial to the Views/Shared folder:

Update _LoginPartial.cshtml with the following code (replace all of its contents):

```
@inject SignInManager<User> SignInManager
@inject UserManager<User> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="LogOff" 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 off</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>
}
```

At this point, you should be able to refresh the site in your browser.

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of an ASP.NET Identity to ASP.NET Core.

Migrating from ASP.NET Web API

By Steve Smith and Scott Addie

Web APIs are HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Core MVC includes support for building Web APIs providing a single, consistent way of building web applications. In this article, we demonstrate the steps required to migrate a Web API implementation from ASP.NET Web API to ASP.NET Core MVC.

View or download sample code

Review ASP.NET Web API Project

This article uses the sample project, *ProductsApp*, created in the article Getting Started with ASP.NET Web API as its starting point. In that project, a simple ASP.NET Web API project is configured as follows.

In *Global.asax.cs*, a call is made to `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` is defined in *App_Start*, and has just one static `Register` method:

```
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 }
            );
        }
    }
}
```

This class configures attribute routing, although it's not actually being used in the project. It also configures the routing table which is used by ASP.NET Web API. In this case, ASP.NET Web API will expect URLs to match the format */api/{controller}/{id}*, with *{id}* being optional.

The *ProductsApp* project includes just one simple controller, which inherits from `ApiController` and exposes two methods:

```
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);
        }
    }
}
```

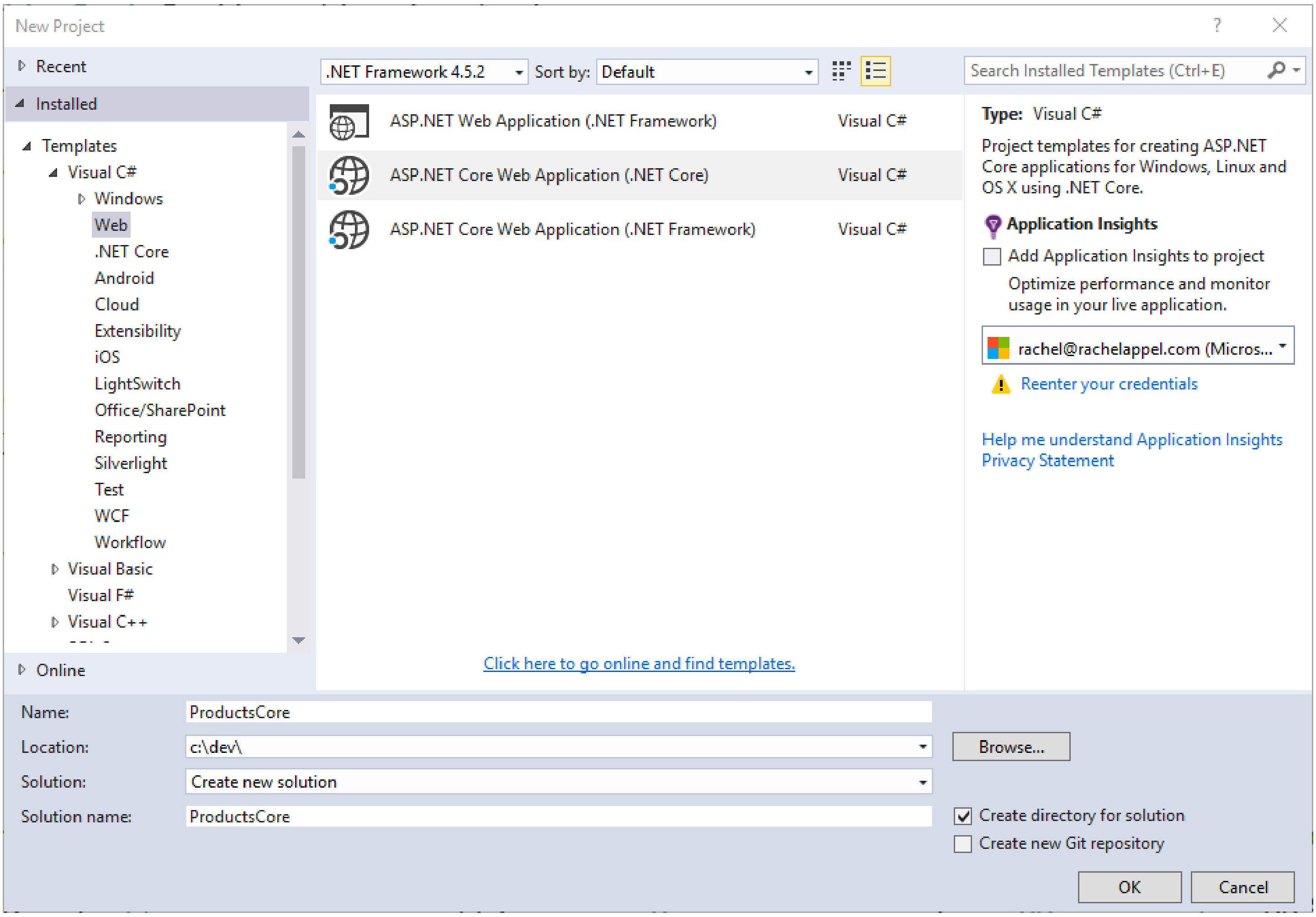
Finally, the model, *Product*, used by the *ProductsApp*, is a simple class:

```

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; }
    }
}
```

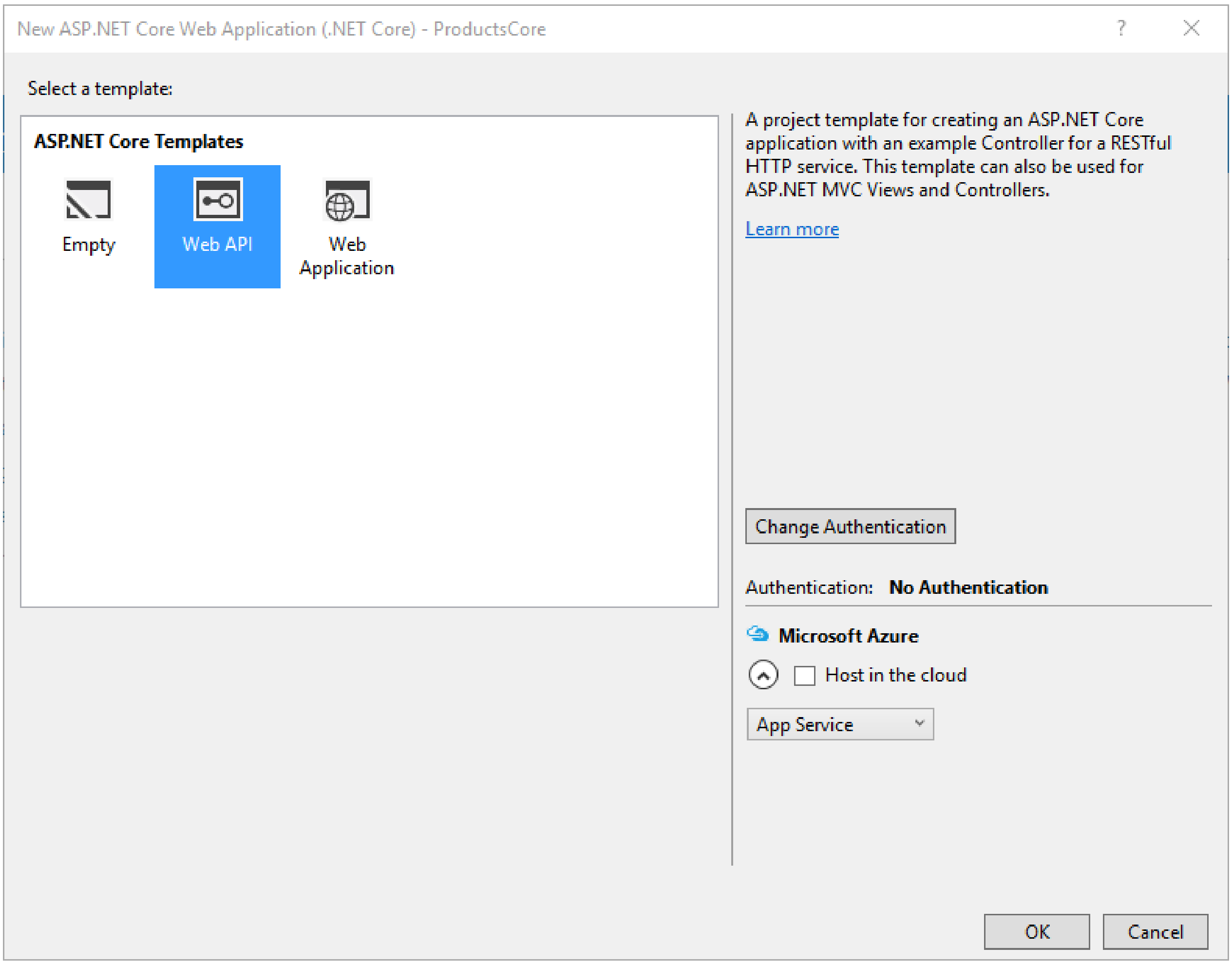
Now that we have a simple project from which to start, we can demonstrate how to migrate this Web API project to ASP.NET Core MVC.

Create the Destination Project

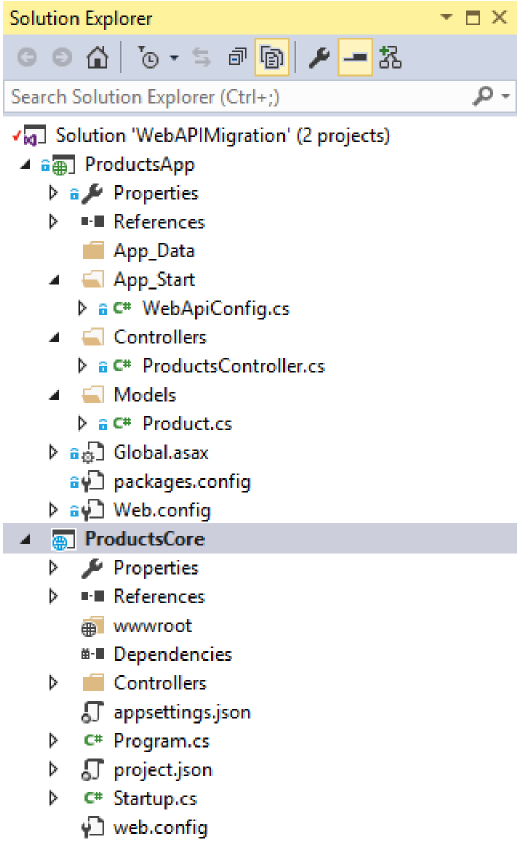
Using Visual Studio, create a new, empty solution, and name it *WebAPIMigration*. Add the existing *ProductsApp* project to it, then, add a new ASP.NET Core Web Application Project to the solution. Name the new project *ProductsCore*.



Next, choose the Web API project template. We will migrate the *ProductsApp* contents to this new project.



Delete the `Project_Readme.html` file from the new project. Your solution should now look like this:



Migrate Configuration

ASP.NET Core no longer uses *Global.asax*, *web.config*, or *App_Start* folders. Instead, all startup tasks are done in *Startup.cs* in the root of the project (see Application Startup). In ASP.NET Core MVC, attribute-based routing is now included by default when `UseMvc()` is called; and, this is the recommended approach for configuring Web API routes (and is how the Web API starter project handles routing).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
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(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();
        }

        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();
        }

        // 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();
        }
    }
}
```

Assuming you want to use attribute routing in your project going forward, no additional configuration is needed. Simply apply the attributes as needed to your controllers and actions, as is done in the sample `ValuesController` class that is included in the Web API starter project:

```
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)
        {
        }
    }
}
```

Note the presence of *[controller]* on line 8. Attribute-based routing now supports certain tokens, such as *[controller]* and *[action]*. These tokens are replaced at runtime with the name of the controller or action, respectively, to which the attribute has been applied. This serves to reduce the number of magic strings in the project, and it ensures the routes will be kept synchronized with their corresponding controllers and actions when automatic rename refactorings are applied.

To migrate the Products API controller, we must first copy *ProductsController* to the new project. Then simply include the route attribute on the controller:

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

You also need to add the `[HttpGet]` attribute to the two methods, since they both should be called via HTTP Get. Include the expectation of an "id" parameter in the attribute for `GetProduct()`:

```
// /api/products
[HttpGet]
...

// /api/products/1
[HttpGet("{id}")]
```

At this point, routing is configured correctly; however, we can't yet test it. Additional changes must be made before *ProductsController* will compile.

Migrate Models and Controllers

The last step in the migration process for this simple Web API project is to copy over the Controllers and any Models they use. In this case, simply copy *Controllers/ProductsController.cs* from the original project to the new one. Then, copy the entire Models folder from the original project to the new one. Adjust the namespaces to match the new project name (*ProductsCore*). At this point, you can build the application, and you will find a number of compilation errors. These should generally fall into the following categories:

ApiController does not exist

System.Web.Http namespace does not exist

IHttpActionResult does not exist

Fortunately, these are all very easy to correct:

Change *ApiController* to *Controller* (you may need to add *using Microsoft.AspNetCore.Mvc*)

Delete any using statement referring to *System.Web.Http*

Change any method returning *IHttpActionResult* to return a *IActionResult*

Once these changes have been made and unused using statements removed, the migrated *ProductsController* class looks like this:

```
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);
        }
    }
}
```

You should now be able to run the migrated project and browse to */api/products*; and, you should see the full list of 3 products. Browse to */api/products/1* and you should see the first product.

Summary

Migrating a simple ASP.NET Web API project to ASP.NET Core MVC is fairly straightforward, thanks to the built-in support for Web APIs in ASP.NET Core MVC. The main pieces every ASP.NET Web API project will need to migrate are routes, controllers, and models, along with updates to the types used by controllers and actions.

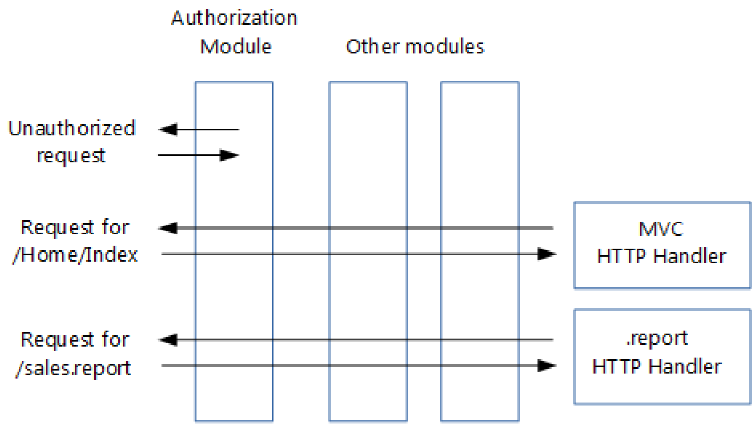
Migrating HTTP handlers and modules to ASP.NET Core middleware

By Matt Perdeck

This article shows how to migrate existing ASP.NET HTTP modules and handlers to ASP.NET Core middleware.

Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



Handlers are:

Classes that implement IHttpHandler

Used to handle requests with a given file name or extension, such as *.report*

Configured in *Web.config*

Modules are:

Classes that implement IHttpModule

Invoked for every request

Able to short-circuit (stop further processing of a request)

Able to add to the HTTP response, or create their own

Configured in *Web.config*

The order in which modules process incoming requests is determined by:

The application life cycle, which is a series events fired by ASP.NET: BeginRequest, AuthenticateRequest, etc. Each module can create a handler for one or more events.

For the same event, the order in which they are configured in *Web.config*.

In addition to modules, you can add handlers for the life cycle events to your *Global.asax.cs* file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

Modules, handlers, *Global.asax.cs*, *Web.config* (except for IIS configuration) and the application life cycle are gone

The roles of both modules and handlers have been taken over by middleware

Middleware are configured using code rather than in *Web.config*

Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

Invoked in principle for every request

Able to short-circuit a request, by not passing the request to the next middleware

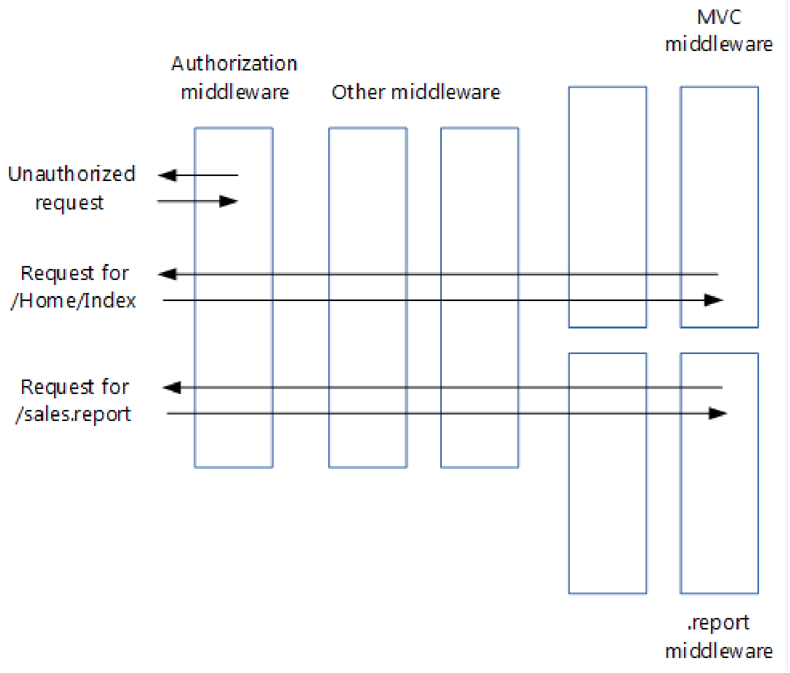
Able to create their own HTTP response

Middleware and modules are processed in a different order:

Order of middleware is based on the order in which they are inserted into the request pipeline, while order of modules is mainly based on application life cycle events

Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses

See Creating a middleware pipeline with IApplicationBuilder



Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

```
// 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.
        }
    }
}
```

As shown in the Middleware page, an ASP.NET Core middleware is a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

```
// 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>();
        }
    }
}
```

The above middleware template was taken from the section on writing middleware.

The *MyMiddlewareExtensions* helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor.

Your module might terminate a request, for example if the user is not authorized:

```
// 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;
    }
}
```

A middleware handles this by not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this does not fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

```
// 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.
}
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. Later on, you'll see how to migrate to the new ASP.NET Core HttpContext.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *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>
```

Convert this by adding your new middleware to the request pipeline in your `Startup` class:

```csharp
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?}");
    });
}
```

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest`, `EndRequest`, etc.) and its order in your list of modules in *Web.config*.

As previously stated, there is no application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware components that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

```
// 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";
        }
    }
}
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

```
// 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>();
        }
    }
}
```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there is no call to `_next.Invoke(context)`. That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

```xml
<?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>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach is that it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

```
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` takes these parameters:

A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.

A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new configuration system gives you these options to solve this:

Directly inject the options into the middleware, as shown in the next section.

Use the options pattern:

Create a class to hold your middleware options, for example:

```
public class MyMiddlewareOptions
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}
```

Store the option values

The configuration system allows you to store option values anywhere you want. However, most sites use *appsettings.json*, so we'll take that approach:

```
{
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

MyMiddlewareOptionsSection here is a section name. It doesn't have to be the same as the name of your options class.

Associate the option values with the options class

The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with a `MyMiddlewareOptions` object that has the actual options.

Update your `Startup` class:

If you're using *appsettings.json*, add it to the configuration builder in the `Startup` constructor:

```
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();
}
```

Configure the options service:

```
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();
}
```

Associate your options with your options class:

```
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();
}
```

Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```
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
    }
}
```

The UseMiddleware extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This is not limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through the dependency injection framework. There is no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

Add a second key to *appsettings.json*

To add a second set of options to the *appsettings.json* file, use a new key to uniquely identify it:

```
{
  "MyMiddlewareOptionsSection2": {
    "Param1": "Param1Value2",
    "Param2": "Param2Value2"
  },
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

Retrieve options values and pass them to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

```
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?}");
    });
}
```

Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

```
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));
    }
}
```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor.

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

```
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of System.Web.HttpContext to the new `Microsoft.AspNetCore.Http.HttpContext`.

HttpContext

HttpContext.Items translates to:

```
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

```
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 and **HttpContext.Request.RawUrl** translate to:

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

```
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 translates to:

```
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers translates to:

```
// 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 translates to:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType;   // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type;    // ex. application
string contentSubType = mediaHeaderValue?.SubType;  // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

```
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"];
}
```

⚠ Warning

Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

⚠ Warning

Use this code only in a handler type middleware, at the end of a pipeline.

You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.

This does not apply to reading a form as shown earlier, because that is done from a buffer.

HttpContext.Response

HttpContext.Response.Status and **HttpContext.Response.StatusDescription** translate to:

```
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and **HttpContext.Response.ContentType** translate to:

```
// 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 on its own also translates to:

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```
string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed here.

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It is this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetHeaders` callback method would look like this:

```
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

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

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetCookies` callback method would look like the following:

```
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);
}
```

Additional Resources

HTTP Handlers and HTTP Modules Overview

Configuration

Application Startup

Middleware

ASP.NET Core 1.1 更新

ASP.NET Core 1.1 更新包括以下新特性：

地址重写中间件
响应缓存中间件
视图组件作为 Tag Helpers
中间件作为 MVC 过滤器
Cookie-based TempData provider
Azure App Service logging provider
Azure Key Vault configuration provider
Azure and Redis Storage Data Protection Key Repositories
WebListener Server for Windows
WebSockets support

ASP.NET Core 选择 1.0 还是 1.1 版本

ASP.NET Core 1.1 相比 1.0 拥有更多新功能。一般来说， 我们推荐您使用最新的版本。

Additional Information

ASP.NET Core 1.1.0 发布说明
如果你想了解 ASP.NET Core 开发团队的进度和计划， 随时访问 ASP.NET Community Standup.