# ASP.NET Core MVC 程序配置演练

By 郑大鹏©

如果在 VS2015 中创建一个 ASP.NET Core MVC 项目，最好选择“Web 应用程序”项目模板，并更改身份验证为“个人用户账户”。这种方式下，项目模板已经做好了各种需要的配置，引用了所有需要用到的服务器端组件和客户端脚本包。我们只需要在上面添加自己的项目需要的模型、视图及控制器就可以了，省了许多麻烦。这正是项目模板所要达到的效果。

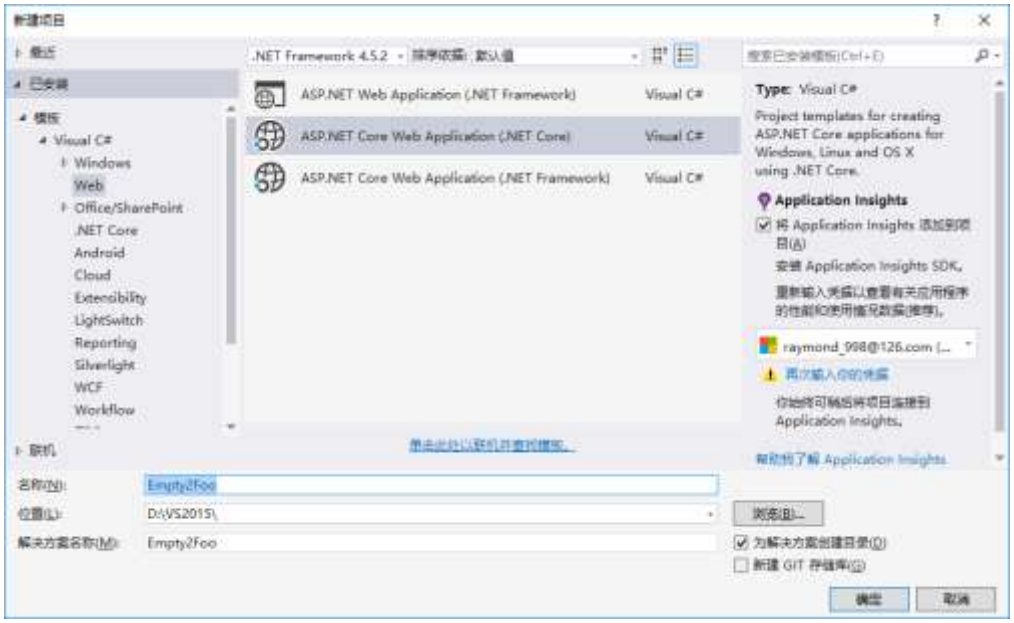
如果要明白这个框架的工作原理，还是要自己动手从空项目模板开始配置一次才比较清晰。本演练用空项目模板创建一个 ASP.NET Core MVC 项目，通过手工添加各项配置，最后让需要使用的各种功能都能运作，并在操作中说明各项配置的含义，帮助初学者了解此编程框架的工作原理。

**1. 创建项目**

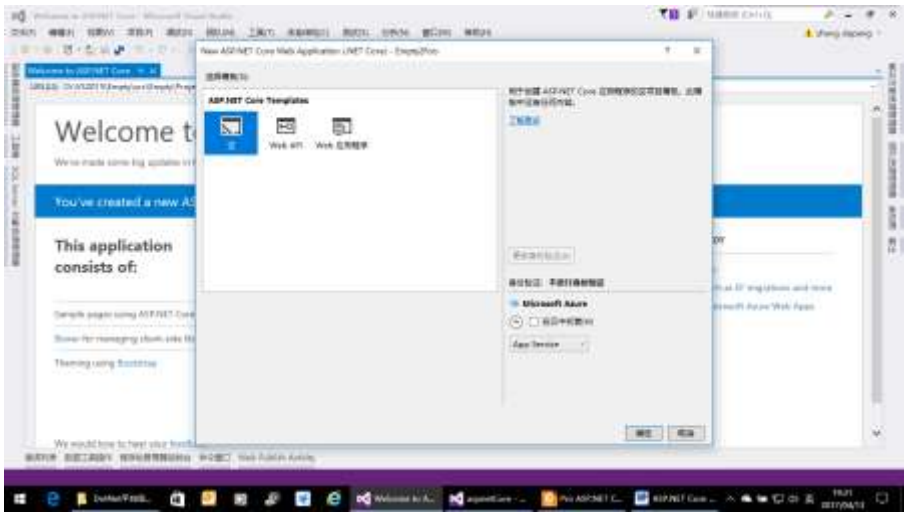
文件>新建>项目…>左边框：已安装—模板—Visual C#—web；

右边：ASP.NET Core Web Application(.NET Core)

下边：名称: Empty2Foo； 位置: D:\VS2015\ 选中“为解决方案创建目录”，不选中“新建 Git 存储库”（参考下图：）

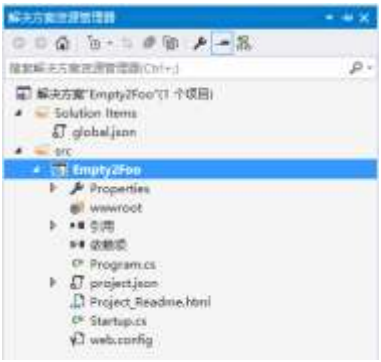



点击“确认”按钮，进入下一页，在选择模板处选“空”，右边保持缺省（身份验证为“不进行身份验证”）。再点击“确认”按钮，这样就创建了一个空的项目。




**2. 添加 Nuget 包**

解释：ASP.NET Core 中，服务器上运行的程序包由 NuGet 管理。NuGet 的包管理目前暂时由 project.json 文件配置(以后版本要改变为 xml 文件配置)。此文件位于项目 Empty2Foo 根目录下（参考下图）。单击 project.json，在 VS 的文档编辑窗口做如下修改。




在 dependencies 对象下添加：

```
"dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.0.1",
      "type": "platform"
    },
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",

    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.1",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    },
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0"
  },
```

在 tools 对象下添加：

```
"tools": {
"Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
"Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
},
```

以上修改所加包如下表说明：

**project.json 文件修改说明**

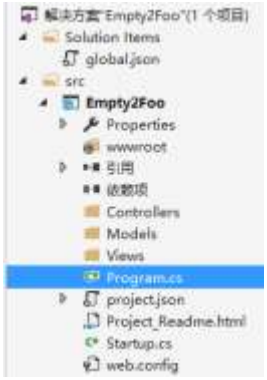
| 包名称 | 说明 |
| --- | --- |
| Microsoft.AspNetCore.Mvc | 此包包含 ASP.NET Core MVC 且提供控制器及 Razor 视图等基本功能 |
| Microsoft.AspNetCore.StaticFiles | 此包提供对 wwwroot 文件夹下 images, JavaScript 和 CSS 等静态文件的服务支持 |
| Microsoft.AspNetCore.Razor.Tools | 此包提供对 Razor 视图的工具支持，包括视图上对内建 tag 帮手的智能感知 |

在 tools 节添加的 Microsoft.AspNetCore.Razor.Tools 配置该工具在 VS 中的正常使用。

添加后，要保存 project.json。VS 会自动开始下载和安装涉及的包。需要一点时间才能安装完毕。

### 3. 添加项目文件夹

右键单击项目名 Empty2Foo，添加 Controllers、Models 和 Views 文件夹。如下图：




### 4. 修改 Startup 类

单击 Startup.cs 文件，在 VS 的文件编辑窗口修改该文件。在 ConfigureServices 方法中，添加一行：

```
services.AddMvc();
```

在 Configure 方法中，先删除

```
app.Run(async (context) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
```

然后添加以下三行：

```
app.UseStatusCodePages();
app.UseStaticFiles();
app.UseMvcWithDefaultRoute();
```

最后结果如下图：

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

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
0 个引用
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();
}
```

ConfigureServices 方法用来启动可在整个应用中通过依赖注入特性使用的共享对象。在该方法中调用的 AddMvc 方法是一个扩展方法，用来启动在 MVC 应用中使用的共享对象。Configure 方法用来启动接收和处理 HTTP 请求的各种特性(功能)。以上在此方法中调用的每一个方法都是扩展方法，用来启动一种对 HTTP 请求的处理程序，具体说明如下：

loggerFactory.AddConsole()：

UseDeveloperExceptionPage()：此扩展方法显示出现在应用的异常的详细信息，这在开发过程中很有用。但发布时不用此功能。

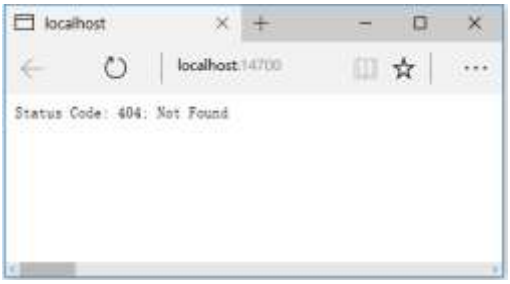
UseStatusCodePages()：此扩展方法添加简单的消息到某些 HTTP 响应中，要不然这类响应没有主体(body)，例如 404-Not Found 响应就是这样。

UseStaticFiles()：此扩展方法启动对 wwwroot 目录下静态内容的服务支持。

UseMvcWithDefaultRoute()：此扩展方法启动 ASP.NET Core MVC 服务，使用缺省的路由配置。

## 5. 试运行程序

此时，单击 VS 菜单上：调试>开始执行(不调试)，程序会运行，并显示如下信息：




## 6. 添加模型、控制器、视图

在 Models 文件夹，添加一个模型类(右键单击 Models 文件夹，添加>新建项，左边选 code,右边选“类”，单击右下角“添加”按钮)文件 Person.cs，代码如下：

```
public class Person
{
    public String Name { get; set; }
    public int Age { get; set; }
}
```

在 Contollers 文件夹，添加一个 HomeController.cs 控制器文件(右键单击 Contollers 文件夹，添加>新建项，左边选 ASP.NET，右边选 MVC 控制器类，单击右下角“添加”按钮)，代码如下：

```
public class HomeController : Controller {
    // GET: /<controller>/
    public IActionResult Index() {
```

```
            ViewBag.Title=”人员信息输入”;
            return View();
        }
        [HttpPost]
        public IActionResult Index(Person p) {
            return View("PersonInfo", p);
        }
    }
```

注意：在名称空间引用部分，要记得加上using Empty2Foo.Models;

在Views文件夹，添加Shared和Home两个文件夹。

右键单击Shared文件夹，添加>新建项，左边选ASP.NET，右边选MVC视图布局页，单击右下角“添加”按钮。文件名为_Layout.cshtml。使用文件的缺省内容即可。

右键单击Views文件夹，添加>新建项，左边选ASP.NET，右边选MVC视图起始页，单击右下角“添加”按钮。文件名为_ViewStart.cshtml。使用文件的缺省内容即可。

右键单击Views文件夹，添加>新建项，左边选ASP.NET，右边选MVC视图导入页，单击右下角“添加”按钮。文件名为_ViewImports.cshtml。在文件上加一行：

```
@using Empty2Foo.Models
```

注意此行后面没有分号(;)。

右键单击Home文件夹，添加>新建项，左边选ASP.NET，右边选MVC视图页，单击右下角“添加”按钮。文件名为Index.cshtml。在文件中删除现有内容，输入以下内容：

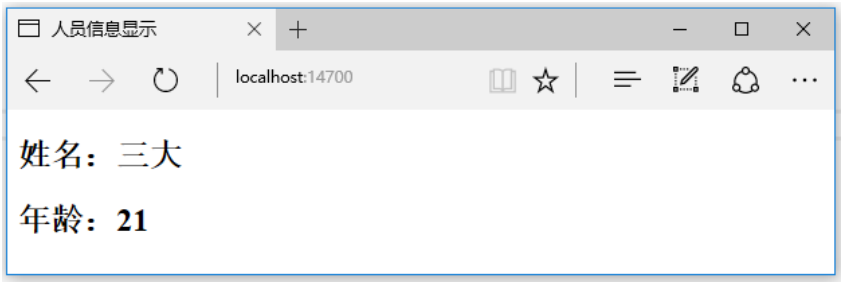
```
<form method="post">
    <label for="Name">姓名：</label>
    <input type="text" name="Name" />
    <label for="Age">年龄：</label>
    <input type="text" name="Age" />
    <input type="submit" />
</form>
```

右键单击Home文件夹，添加>新建项，左边选ASP.NET，右边选MVC视图页，将文件名改为PersonInfo.cshtml，单击右下角“添加”按钮。在文件中删除现有内容，输入以下内容：

```
@model Person
@{ ViewBag.Title = "人员信息显示";}
<h2>姓名：@Model.Name</h2>
<h2>年龄：@Model.Age</h2>
```

至此，单击VS菜单上：调试>开始执行(不调试)，程序会运行，并显示如下信息：


单击页面上“提交查询内容”按钮，会显示如下内容：




理解要点：以上演示了模型自动绑定。即页面表单上的输入字段如果与模型类Person的属性同名，提交后会自动传给控制器参数。模型数据也可以传给视图，通过View()的参数，在视图上用@model Person标注。

另外，ViewBag可以传递模型视图之外的零星数据，如页面显示时的Title。ViewBag可以在控制器方法中直接赋值，也可以在视图中用Razor脚本赋值。

以上演练还说明了模型、视图和控制器之间的关系以及在项目中添加的方式。

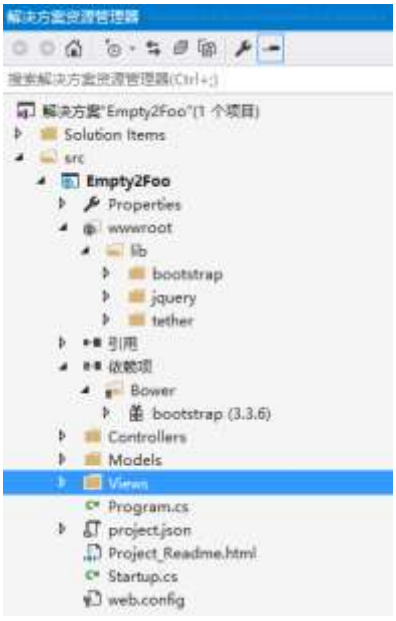
## 7. 添加 bootstrap 布局支持

为了使用bootstrap样式框架，首先要加入有关的客户端脚本包。客户端脚本包使用Bower管理(类似NuGet)。

右键单击项目名Empty2Foo>添加>新建项...，左边面板选Client-Side，右边面板选 Bower配置文件，单击右下角添加按钮。文件名为bower.json不用修改。在文件中加上：

```
{
    "name": "asp.net",
    "private": true,
    "dependencies": {
      "bootstrap": "3.3.6"
    }
}
```

保存文件后，VS会自动下载和安装bootstrap包和它依赖的jQuery包。这些包会出现在wwwroot下面的lib文件夹下。并在依赖项下出现一个Bower文件夹。如果网速慢，保存bower.json文件后，会在依赖项文件夹名称后显示“(正在还原)”的提示。安装完毕文件夹名恢复正常，并看到以上所述内容。这种现象在修改project.json文件后也出现过。现在，解决方案资源管理器内容如下：




注意，上面没有出现bower.json文件。此文件是隐藏的。如果要再次编辑修改，右键单击Bower项，选

择打开bower.json文件即可。

现在，在视图文件中还要做一些修改，就可以使用bootstrap的样式功能了。

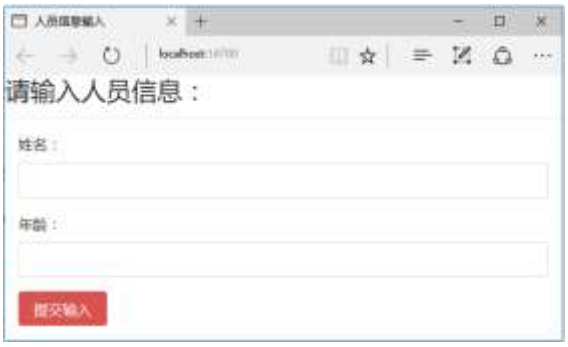
将_Layout.cshtml文件内容改为：

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"/>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
    <title>@ViewBag.Title</title>
</head>
<body>
     @RenderBody()
     <script src="/lib/jquery/dist/jquery.js"></script>
     <script src="/lib/bootstrap/dist/js/bootstrap.js"></script>
</body>
</html>
```

将index.cshtml文件内容重写为：

```
<form method="post"    class="form-horizontal">
    <h3>请输入人员信息：</h3>
    <hr />
    <div class="form-group">
        <label for="Name" class="col-md-2 control-label">姓名：</label>
        <div class="col-md-10">
            <input type="text" name="Name" class="form-control" />
        </div>
    </div>
    <div class="form-group">
        <label for="Age" class="col-md-2 control-label">年龄：</label>
        <div class="col-md-10">
            <input type="text" name="Age" class="form-control" />
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-danger">提交输入</button>
        </div>
    </div>
</form>
```

现在，点击VS主菜单上 调试>开始执行(不调试)，则index页显示效果如下,说明bootstrap已经有效。




## 8. 添加数据验证

服务器端的数据验证是在模型数据绑定时自动进行的。只需要在视图模型类中添加对属性数据的验证标注，服务器端在进行模型数据绑定时就会自动验证。如果验证通过，设置 ModelState 对象的 IsValid 属性为 true，否则设置为 false。在控制器方法中可以据此作出不同的处理。为了说明以上原理，先修改 Models 中的 Person.cs 文件内容如下：

```
public class Person
{
    [Required(ErrorMessage="姓名不能为空！")]
    public String Name { get; set; }
    [Range(15,30,ErrorMessage ="年龄必须在15到30岁。" )]
    public int Age { get; set; }
}
```

注意，要在此文件上部加上：

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

然后，修改 HomeController.cs 文件中 Index(Person p)方法代码如下：

```
        [HttpPost]
        public IActionResult Index(Person p)
        {
            if (ModelState.IsValid)
                return View("ValidateOk");
            else
                return View("ValidateFailed");
        }
```

在 Views 文件夹下的 Home 子文件夹中，添加 ValidateOk.cshtml 文件，在文件中写一句话：

```
Validation is ok!
```

在同一文件夹添加一个 ValidateFailed.cshtml 文件，在文件中写：

```
Validation is failed.
```

然后运行程序(点击调试下面的开始执行)，则当没有输入 Name 或者 Age 的值超出 15-30 时，点提交会显示 Validation is failed.；如果输入合符要求，则显示 Validation is ok! 。

以上仅仅使用最基本的服务器端验证，未能将错误信息在输入页面清楚地显示出来。可以按照以下修改得到更多服务器端验证功能。

首先要使用 tag helper。这是 ASP.NET Core MVC 框架新增的特性，以往的版本没有此功能（因此大家无法从英文版参考书之外处获得信息）。所谓 tag helper 其实是处理页面标记的类。这些类提供 cshtml 文件上特定属性的替换服务。当用户请求一个功能时，路由会被解析，指向一个控制器及控制器方法。在控制器方法中，会渲染一个 cshtml 文件视图。渲染视图时，需要将含有 Razor 脚

本的 cshtml 页面转换成 html 页面。这时，除了处理 Razor 脚本，还可以使用 tag helper 类对 cshtml 页面上的特殊属性(tag)进行替换。每一 tag helper 类对应一定的特殊属性。开发者可以根据自己的需要编写这种 tag helper 类。但与数据验证信息显示有关的 tag helper 类系统已经提供。为了使用系统提供的这些 tag helper，需要在_ViewImports.cshtml 文件中添加一行：

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

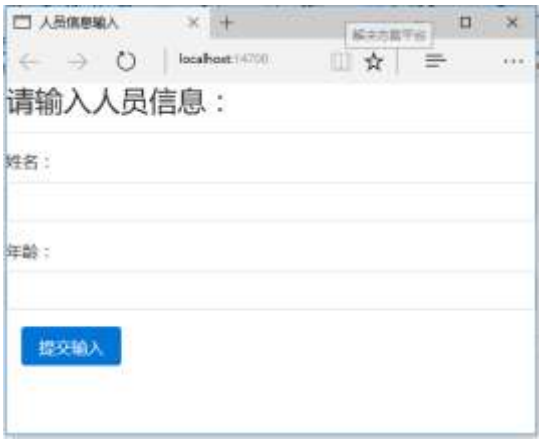
然后，在我们的视图中，可以在 html 元素的属性中添加一些特殊的属性。修改 Index.cshtml 文件内容如下：

```
@model Person
<form method="post" class="form-horizontal">
    <h3>请输入人员信息：</h3>
    <hr />
    <div class="form-group">
        <label asp-for="Name">姓名：</label>
        <div><span asp-validation-for="Name" class="text-danger"></span></div>
        <input asp-for="Name" class="form-control" />
    </div>
    <div class="form-group">
        <label asp-for="Age">年龄：</label>
        <div><span asp-validation-for="Age" class="text-danger"></span></div>
        <input asp-for="Age" class="form-control" />
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-primary">提交输入</button>
        </div>
    </div>
</form>
```

以上代码中，以 asp-开头的元素属性，就是系统内部已经提供的 tag helper 需要处理的属性标记。如 asp-for="Age"。这里，asp-for 特殊属性所指定的值"Age"必须是当前视图模型的属性，故要在该文件的上面，标记@model Person。上述 就是在输入 Name 出错时，显示相应验证错误信息的标记。由于输入出错时要在原输入页显示错误信息，故对 HomeController.cs 中的 Index(Person p)方法做如下修改：

```
        [HttpPost]
        public IActionResult Index(Person p)
        {
            if (ModelState.IsValid)
                return View("PersonInfo",p);
            else
                return View();
        }
```

现在运行程序，可以看到以下界面：




如果不输入任何信息，直接点击“提交输入”，则出现以下显示：


现在，当服务器端验证不通过时，可以显示有关验证信息了。在上面的信息中，由于没有输入年龄，而年龄的类型为 int。服务器端在数据验证时，无法将空字符串转换为整数，就显示了内部的缺省信息提示。内部的缺省信息提示有很多种，可以通过代码改变。将 Startup.cs 文件中 ConfigureServices 方法代码改为以下：

```
public void ConfigureServices(IServiceCollection services)
{
    //services.AddMvc();
    services.AddMvc().AddMvcOptions(opts =>
    {
        opts.ModelBindingMessageProvider.ValueMustNotBeNullAccessor =
        value => "请输入一个值：";
    });
}
```

再次运行程序，当没有输入任何值时点击提交会得到：


这时，如果输入姓名，但年龄范围不在 15 到 30 之间，会有如下显示：


关于服务器端验证，还有许多功能这里从略。

如果要在用户输入不合要求时，在传给服务器之前就立即反馈信息，就要使用客户端验证。通常客户端验证和服务器端验证是同时使用的。Asp.NET Core MVC 框架将这两种验证都自动化了，且客户端验证的依据也是视图模型中的标注，只是客户端验证需要依赖两个 js 库，jquery-validatio.js 和 jquery-validation-unobtrusive.js。前者是客户端 Web 开发通用的 jQuery 验证库，后者是微软为自己的 ASP.NET Core MVC 框架增加的。

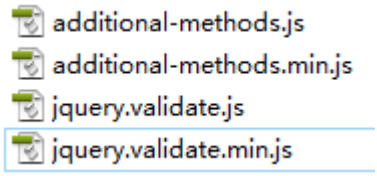
为了使用这两个 js 库(都依赖 jQuery 库)，需要在项目中加以引用。本来也应可按以下步骤修改 bower.json 文件中的 dependencies 节，让 bower 自动下载和安装需要的包，英文参考书也是这样指引的。但实际安装过程中，遇到了很多问题，最终未能成功（以下【】围住灰底部分是按照指引做的过程，但最后无法运行）。

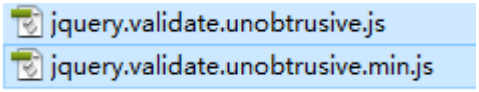
【右键单击解决方案资源浏览器中“依赖项”下面的 Bower 文件夹，选择“打开 bower.json”项。在 VS 编辑框中对该文件修改如下：

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.6",
    "jquery": "3.2.1",
    "jquery-validation": "1.16.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

上述修改保存后，VS自动下载和安装新的包。但jquery-validation包安装后，在其文件夹下面缺少dist文件夹

及需要使用的jquery-validate.js文件。而此文件夹和文件需要对安装的jquery-validation包进行构建(build，类似编译发布)才能得到，这需要下载和安装一个Visual Studio的扩展插件Grunt Launcher extension，地址是：
https://visualstudiogallery.msdn.microsoft.com/dcbc5325-79ef-4b72-960e-0a51ee33a0ff
点击上述地址下载扩展插件并安装，安装成功后，需要关闭当前已打开的VS，然后重新起动和进入Empty2Foo项目。在解决方案管理器中，展开wwwroot/lib/jquery-validation文件夹，右键单击其中的package.json文件，上下文菜单中会出现NPM install packages，点击开始构建。但是，中途会提示缺少一个名称为phantomjs-2.1.1-windows的包，系统自动开始下载(由于网速极慢，等很久都难完成，可以在VS的输出窗口将下载地址粘贴到迅雷上下载，几分钟可以完成)。如果用迅雷下载，要将下载的文件拷贝到NPM所默认的文件夹(等同为NPM包完成了下载)： %user%/AppData/Local/Temp/phantomjs/。重新开始NPM install packages操作，这次可以完成（也会报错，可不理）。完成后可以看到jquery-validation文件夹下有了dist文件夹，且文件夹中有jquery-validate.js文件。】
由于jquery库只是一个js脚本文件，从别的项目中拷贝相应文件到当前项目也是可以的。为此，在wwwroot/lib文件夹下创建jquery-validation和jquery-validation-unobtrusive文件夹，在jquery-validation下创建一个dist文件夹，采用添加现有项方式从其他项目文件中添加以下文件到此文件夹：

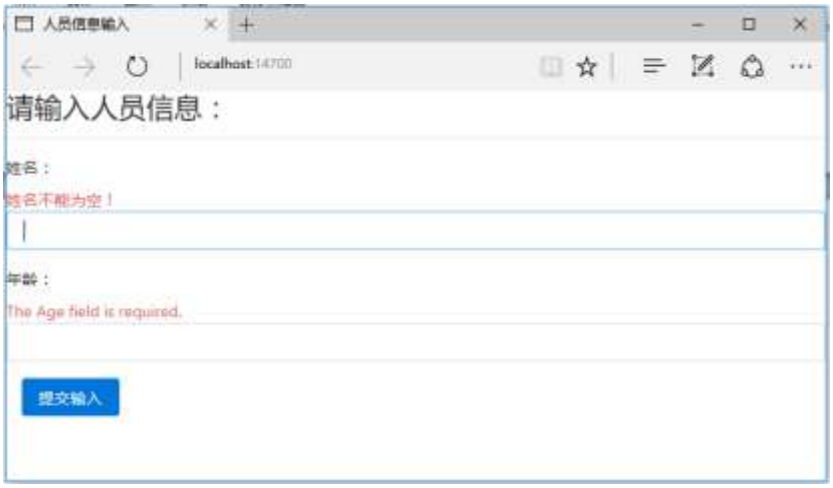



用同样方法添加 

到jquery-validation-unobtrusive文件夹。
修改_Layout.cshtml文件如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"/>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
    <title>@ViewBag.Title</title>
</head>
<body>
     @RenderBody()
     <script src="/lib/jquery/dist/jquery.js"></script>
     <script src="/lib/bootstrap/dist/js/bootstrap.js"></script>
     <script src="/lib/jquery-validation/dist/jquery.validate.js"></script>
     <script src="/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
</body>
</html>
```

重新运行项目，在没有输入任何资料时，显示页面如下：




现在的错误提示，是客户端脚本在请求发往服务器之前产生的，这可以从未输入年龄所显示的缺省提示变化看出。现在客户端验证已经生效了。还可以将控制器HomeController代码改为：

```
public class HomeController : Controller
{
    // GET: /<controller>/
    public IActionResult Index()
    {
        ViewBag.Title = "人员信息输入";
        return View();
    }
    [HttpPost]
    public IActionResult Index(Person p)
    {
        if (ModelState.IsValid)
            return View("ValidateOk");
        else
            return View("ValidateFailed");
    }
}
```

来证明现在起作用的是客户端验证。因为如果是服务器端验证不通过，将显示ValidateFailed.cshtml页面，里面只有一行“Validation is failed.”。但实际并非如此。现在在页面输入符合要求的内容，显示的是ValidateOk.cshtml页面，里面只有一行“Validation is ok!”。

## 9. 添加 EF Core 功能

所谓EF即Entity Framework，是.NET平台的对象映射框架，可以将关系数据库中的数据存储映射为C#中的对象，极大地简化了数据应用的编程。经过多年发展，此框架已经成熟。.NET Core平台对应的EF称为EF Core。为了在项目中使用数据库，必须安装EF Core组件。打开project.json文件。在dependencies部分和tools部分分别添加与EF Core有关的包如下（黑体为新增加部分）：

```
"dependencies": {
  "Microsoft.NETCore.App": {
    "version": "1.0.1",
    "type": "platform"
```

```
    },
    "Microsoft.AspNetCore.Diagnostics": "1.0.0",

    "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",
    "Microsoft.AspNetCore.Server.Kestrel": "1.0.1",
    "Microsoft.Extensions.Logging.Console": "1.0.0",
    "Microsoft.AspNetCore.Razor.Tools": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    },
    "Microsoft.AspNetCore.StaticFiles": "1.0.0",
    "Microsoft.AspNetCore.Mvc": "1.0.0",
    "Microsoft.EntityFrameworkCore.SqlServer": "1.0.1",
    "Microsoft.EntityFrameworkCore.SqlServer.Design": {
      "version": "1.0.1",
      "type": "build"
    },
    "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
  },

  "tools": {
    "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
    "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final",
    "Microsoft.EntityFrameworkCore.Tools": {
      "version": "1.0.0-preview2-final",
      "imports": [ "portable-net45+win8+dnxcore50", "portable-net45+win8" ]
    }
  },
```

保存修改后的project.json文件，则NuGet开始自动下载和安装EF Core需要的包。这些包是使用数据库和根据现有数据库生成模型文件必须的，是下一步工作的准备。

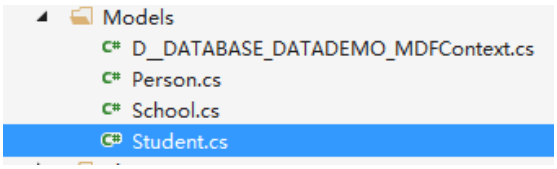
## 10. 使用数据库

下载 DataDemo 数据库文件包，解压到你的电脑的 D:\Database 文件夹。右键单击“服务器资源管理器”中的“数据连接”，点击“添加连接…”，更改数据源为“Sql Server 数据库文件”，点击浏览按钮，定位到 D:\Database 文件夹的数据库文件，点击确定，则 DataDemo.mdf 数据库连接被添加。

点击 工具>NuGet 程序包管理器>程序包管理器控制台，在 PM>提示符号后面，输入(粘贴)以下命令并按回车键：

```
Scaffold-DbContext "Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=D:\Database\DataDemo.mdf;Integrated
Security=True;Connect Timeout=30" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

注意，以上命令中，双引号围住的部分是刚添加的数据库连接的连接字符串，要从数据库连接的属性窗口拷贝上去，以免手工输入出错。如果命令顺利执行，会在下一行显示 PM> 提示符号。这时，可以在 Models 文件夹看到新增了以下数据模型：




以上新增的文件，即根据现有数据库 DataDemo.dbf 自动生成的模型文件，节省了大量编写模型代码时间。但还需要对个别文件进行修改。

首先，打开 D__DATABASE_DATADEMO_MDFContext.cs 文件，删除或注释掉该文件中的 OnConfiguring 方法(在文件的上面)；然后，在代码中添加以下构造方法：

```
public D__DATABASE_DATADEMO_MDFContext(DbContextOptions<D__DATABASE_DATADEMO_MDFContext>
options) : base(options) { }
```

注意，以上 D__DATABASE_DATADEMO_MDFContext 为自动生成的类名，如果你的电脑上类名不同，则要以实际情况为准。

打开 Startup.cs 文件，在上部添加以下名称空间引用：

```
using Empty2Foo.Models;
using Microsoft.EntityFrameworkCore;
```

在 ConfigureServices(...)方法添加以下代码注册数据库上下文服务:

```
var connection =     //链接字符串也可以从配置文件appsettings.json中读取
                @"Data
            Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=D:\Database\DataDemo.mdf;Integrated
            Security=True;Connect Timeout=30";
services.AddDbContext<D__DATABASE_DATADEMO_MDFContext>(options =>
options.UseSqlServer(connection));
```

以上操作，让控制器类可以通过 ASP.NET Core MVC 自带的依赖注入机制，自动获得数据库上下文对象，这样就可以通过 EF Core 读写数据库了。使用 EF Core 读写数据库不需要直接使用 ADO.NET，而且完全面向对象，还可以使用高效率的编程语法 Linq。

为了使用数据库中 School 表，可以先对模型类 School.cs 进行如下修改：

```
public partial class School
{
    public School()
    {
        Student = new HashSet<Student>();
    }
    public int ObjId { get; set; }
    [Required(AllowEmptyStrings =false,ErrorMessage ="名称不能为空！")]
    public string Name { get; set; }
    [RegularExpression(@"http://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?", ErrorMessage ="请输入正确的网址！")]
    public string Site { get; set; }
    public virtual ICollection<Student> Student { get; set; }
}
```

即：添加数据验证要求。注意，要在文件上部加上

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

在 Controller 文件夹下，添加 SchoolController 控制器，代码如下：

```
public class SchoolController : Controller
```

```
{
    private D__DATABASE_DATADEMO_MDFContext _dbContext;
    public SchoolController(D__DATABASE_DATADEMO_MDFContext dbContext)
    {
        _dbContext = dbContext;
    }
    // GET: /<controller>/
    public IActionResult Index()
    {
        School s = new School();
        return View(s);
    }
    [HttpPost]
    public IActionResult Index(School s)
    {
        _dbContext.School.Add(s);
        _dbContext.SaveChanges();
        var schools = _dbContext.School.AsQueryable<School>();                    ;
        return View("SchoolList",schools);
    }
}
```

此文件名为 SchoolController.cs。在上方要加上：using Empty2Foo.Models;。注意以上代码中的黑体部分，是为了通过依赖注入自动生成本控制器需要用的数据库上下文类。

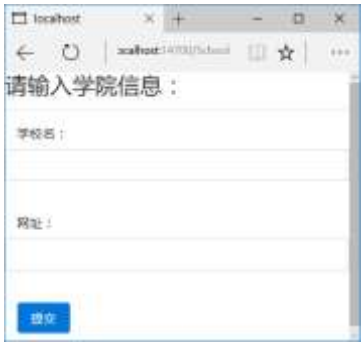
在 Views 文件夹下，添加 School 文件夹，然后在该文件夹下添加 Index.cshtml 如下：

```
@model School
<form method="post" class="form-horizontal">
    <h3>请输入学院信息：</h3>
    <hr />
    <div class="form-group">
        <label asp-for="Name" class="col-md-2">学校名：</label>
        <input asp-for="Name" class="col-md-4 form-control" />
        <span asp-validation-for="Name" class="col-md-4 text-danger"></span>
    </div>
    <div class="form-group">
        <label asp-for="Site" class="col-md-2">网址：</label>
        <input asp-for="Site" class="col-md-4 form-control" />
        <span asp-validation-for="Site" class="col-md-4 text-danger"></span>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="提交" class="btn btn-primary" />
        </div>
    </div>
</form>
```

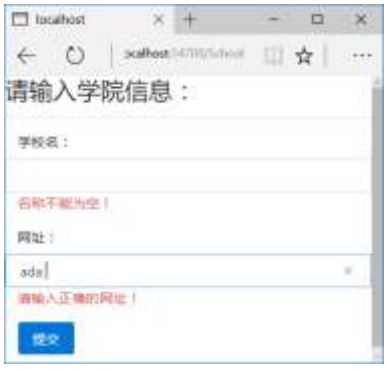
在同一文件夹添加 SchoolList.cshtml 视图，代码如下：

```
@model IQueryable<School>
<h3>学院列表：</h3>
@foreach(School s    in Model)
{
    <p>名称：@s.Name ; 网址：@s.Site</p>
}
```

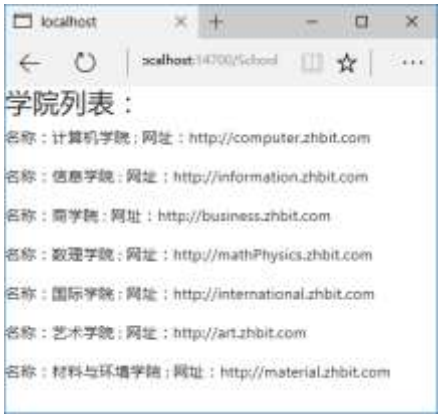
到这里，可以实验一下效果了。先点击 VS 主菜单上的调试>开始执行(不调试)，等首页显示出来后，在地址栏后面加上/School，然后回车，则显示页面如下：




不输入学校名，输入格式不正确的网址，点击提交可以再次看到客户端验证生效的页面如下：




再输入合乎要求的内容，点击提交，会显示以下页面：




## 11. 使用应用程序配置文件 appSettings.json

在以上代码中，数据库的链接字符串写在了 Startup.cs 文件中。这不仅难看，也不便于修改。放在配置文件中可以在修改配置项时不必重新编译程序。

右键单击数据库连接 DataDemo.mdf，点击“属性”，在属性窗中拷贝链接字符串。在解决方案资源管

理器中，右键单击 Empty2Foo 项目，添加>新建项，在已安装模板下选“asp.net”,在右边选“ASP.NET 配置文件”，不用修改文件名，点确认，进入文件编辑框，将链接字符串粘贴在 appsettings.json 文件的 "DefaultConnection": 之后（注意双引号，并在连接字符串尾部保留原来的 MultipleActiveResultSets=true）。最后为：

```
"ConnectionStrings": {
    "DefaultConnection": "Data
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=D:\\Database\\DataDemo.mdf;Integrated
Security=True;Connect Timeout=30;MultipleActiveResultSets=true"
      }
```

以上操作添加了应用程序的配置文件。目前只含有连接字符串。为了在代码中读取此配置文件的内容，需要在 project.json 文件 dependencies 部分添加新的包引用：

```
"Microsoft.Extensions.Configuration": "1.0.0",
"Microsoft.Extensions.Configuration.Json": "1.0.0"
```

保存 project.json 文件，待包安装完成。然后，打开 Startup.cs 文件，在上面加上以下名称空间引用：

```
using Microsoft.Extensions.Configuration;
```

然后，在 Startup 类中添加以下构造方法和属性：

```
public Startup(IHostingEnvironment env)
{
    Configuration = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    .AddJsonFile("appsettings.json")
    .Build();
}
public IConfigurationRoot Configuration { get; set; }
```

最后，在 ConfigureServices 方法中注释 connection 变量的定义和赋值，将该方法修改为：

```
public void ConfigureServices(IServiceCollection services)
{
    //services.AddMvc();
    services.AddMvc().AddMvcOptions(opts =>
    {
        opts.ModelBindingMessageProvider.ValueMustNotBeNullAccessor =
        value => "请输入一个值：";
    });
    //var connection =     //链接字符串也可以从配置文件appsettings.json中读取
    //      @"Data Source= (LocalDB)\MSSQLLocalDB;AttachDbFilename=D:\Database\DataDemo.mdf;
Integrated Security=True;Connect Timeout=30";
    services.AddDbContext<D__DATABASE_DATADEMO_MDFContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
    }
```

注意用黑体部分取代原来的 connection 参数即可。

现在重新运行网站，并访问/School 控制器，如果与原来一样可以读写数据库，则连接字符串读写无误。

## 12. 添加 Identity 身份认证

Identity 是 ASP.NET Core MVC 框架自带的个人用户身份认证及授权管理的 API。经过一代一代的发展，

终于达到了灵活性与功能性俱佳的水平。为了使用此框架提供的功能，需要在 project.json 中 dependencies 部分添加以下包说明：

"Microsoft.AspNetCore.Identity.EntityFrameworkCore": "1.0.0"

由于 Identity 包中已经提供了复杂的用户管理和权限管理功能，一般的应用都不再需要自己扩展就能很好地满足要求，尽管该框架提供了丰富的扩展功能。以下演练仅仅使用基本功能。

首先，在 Models 文件夹，添加一个模型文件，定义应用程序需要的用户模型。此用户模型需要继承自 Identity 框架的 IdentityUser 类。代码如下：

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
namespace Empty2Foo.Models
{
    public class AppUser : IdentityUser
    {
        // no additional members are required
        // for basic Identity installation
    }
}
```

由于 IdentityUser 类中已经包含了需要的各种字段，这里可以不添加任何属性。当然，如果需要你可以在这个类添加你需要的属性。IdentityUser 类已经具有以下属性：

| 属性名 | 说明 |
| --- | --- |
| Id | 此属性为用户的唯一 ID |
| UserName | 此属性返回用户的用户名 |
| Claims | 此属性返回用户的“身份凭据”(claims)集合。 |
| Email | 用户的 e-mail 地址 |
| Logins | 此属性返回用户的登录集合，用于第三方身份认证(如用 facebook 账号登录) |
| PasswordHash | 此属性返回用户密码的哈希 |
| Roles | 此属性返回用户的角色集合 |
| PhoneNumber | 此属性返回用户的电话号码 |
| SecurityStamp | 此属性返回一个值，回随着用户识别信息如密码的改变而改变。Stamp 即邮戳，随时间变。 |

下一步是创建一个基于 EF Core 的 Identity 框架需要用的数据库上下文类，同样可放在 Models 文件夹下。代码如下：

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
namespace Empty2Foo.Models
{
    public class AppIdentityDbContext : IdentityDbContext<AppUser>
    {
        public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options) : base(options)
        {
        }
    }
}
```

这里，基类 IdentityDbContext<TUser>也是 Identity 框架提供的，必须继承它。虽然可以根据需要扩展，但这里这样定义已经足够。此类已经定义了一个用于 Identity 管理需要的数据库。后面会用 EF Core 命令创建这个配套的数据库。此前需要做的是指定这个数据库的连接字符串，这样可以将数据库放在我们希望的服

务器上或目录上。由于本演练前面已经有了一个 DataDemo 数据库，这里创建 Identity 需要的数据库时，最好让这两个数据库合二为一，便于管理。因此下面只要将 DataDemo 的连接字符串给 Identity 用就可以了。这个连接字符串已经保存在 appsettings.json 配置文件中，且配置文件读取前已设置就绪。

打开 Startup.cs 文件，在上面添加一行名称空间引用：

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

在 ConfigureServices 方法中添加以下两行代码：

```
services.AddDbContext<AppIdentityDbContext>(options=>options.UseSqlServer(
                    Configuration.GetConnectionString( "DefaultConnection")));
services.AddIdentity<AppUser, IdentityRole>().AddEntityFrameworkStores<AppIdentityDbContext>();
```

在 Configure 方法中，在 app.UseMvcWithDefaultRoute();上面加上一行：app.UseIdentity();

即：

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole();
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseIdentity();
    app.UseMvcWithDefaultRoute();
}
```

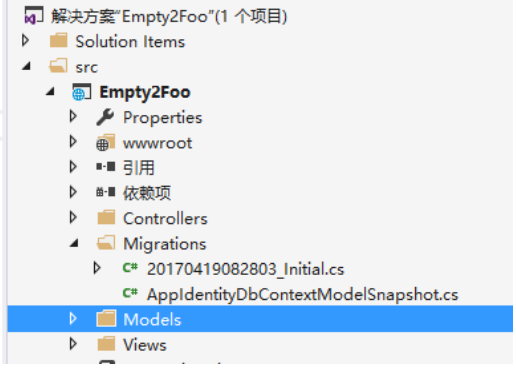
最后一步，只需要点击：工具>NuGet 包管理器>程序包管理器控制台，打开控制台后在 PM>提示符后面分别输入以下三行命令（输入后等一会才执行完）：

```
Import-Module C:\Users\[你登录 Windows 的用户名]\.nuget\packages\Microsoft.EntityFrameworkCore.Tools
\1.0.0-preview2-final\tools\EntityFrameworkCore.psd1【回车】
Add-Migration Initial -context AppIdentityDbContext【回车】
Update-Database -context AppIdentityDbContext【回车】
```

执行完第一条命令后，可能会有类似以下的提示：

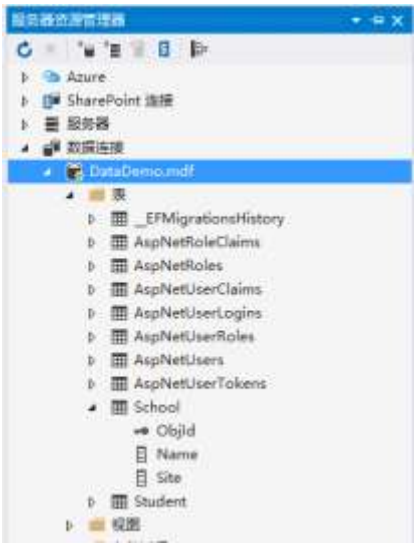
模块“EntityFrameworkCore”中的某些导入命令的名称包含未批准的动词，这些动词可能导致这些命令名不易被发现。若要查找具有未批准的动词的命令，请使用 Verbose 参数再次运行 Import-Module 命令。有关批准的动词列表，请键入 Get-Verb。

不必理会。但如果有红色字体的提示就要检察原因了。没有问题后继续输入第 2 条命令。执行第 2 条命令后，在解决方案资源管理器中，可以看到项目文件夹下多了一个 Migrations 文件夹如下：



这些自动生成的文件夹和文件不能被手工修改。

执行完第三条命令后，从服务器资源管理器打开数据库连接，查看 DataDemo.mdf 数据库，可以看到添加了 Identity 用到的表，如下图所示：




这时，使用 Identity 框架的准备工作已经完成，可以编写代码使用其强大的功能了。

在 Models 文件夹下，为用户注册和登录分别定义两个视图模型，可以放在两个文件，也可以放在一个文件。

代码如下:

```
using System.ComponentModel.DataAnnotations;
namespace Empty2Foo.Models
{
    public class RegisterViewModel
    {
        [Required]
        [Display(Name = "姓名")]
        public string Name { get; set; }
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }
        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "密码")]
        public string Password { get; set; }
        [DataType(DataType.Password)]
        [Display(Name = "确认密码")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }
    public class LoginViewModel
    {
```

```
        [Required]
        public string UserName { get; set; }
        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
        [Display(Name = "记住我?")]
        public bool RememberMe { get; set; }
    }
}
```

在 Controllers 文件夹下面，添加一个 AccountController.cs 控制器定义文件，代码如下：

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Empty2Foo.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Authorization;

namespace Empty2Foo.Controllers
{
    public class AccountController : Controller
    {
        private readonly UserManager<AppUser> _userManager;
        private readonly SignInManager<AppUser> _signInManager;
        public AccountController(UserManager<AppUser> userManager, SignInManager<AppUser>
signInManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
        }
        // GET: /<controller>/
        public IActionResult Index()
        {
            return    View(_userManager.Users);
        }
        public ViewResult Register() => View();
        [HttpPost]
        public async Task<IActionResult> Register(RegisterViewModel model)
        {
            if (ModelState.IsValid)
            {
                AppUser user = new AppUser
                {
                    UserName = model.Name,
                    Email = model.Email
                };
```

```
                IdentityResult result = await _userManager.CreateAsync(user, model.Password);
                if (result.Succeeded)
                {
                    return RedirectToAction("Index");
                }
                else
                {
                    foreach (IdentityError error in result.Errors)
                    {
                        ModelState.AddModelError("", error.Description);
                    }
                }
            }
            return View(model);
        }

        //
        // GET: /Account/Login
        [HttpGet]
        [AllowAnonymous]
        public IActionResult Login(string returnUrl = null)
        {
            ViewData["ReturnUrl"] = returnUrl;
            return View();
        }

        //
        // POST: /Account/Login
        [HttpPost]
        [AllowAnonymous]
        public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
        {
            ViewData["ReturnUrl"] = returnUrl;
            if (ModelState.IsValid)
            {
                var result = await _signInManager.PasswordSignInAsync(model.UserName, model.Password,
model.RememberMe, lockoutOnFailure: false);
                if (result.Succeeded)
                {
                    return RedirectToAction("Index");
                }
            }
            // If we got this far, something failed, redisplay form
            return View(model);
```

```
        }

        private IActionResult RedirectToLocal(string returnUrl)
        {
            if (Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction(nameof(HomeController.Index), "Home");
            }
        }
    }
}
```

此控制器中，分别定义了 Index、Register、Login 三种 Action。由于含有输入数据表单的 Action 实际包含 HttpGet 和 HttpPost 两种方法，故需要分开定义。

在 Views 文件夹下，添加一个 Account 文件夹，在该文件夹下面增加一下三个视图文件：

1. Index.cshtml

```
@model IEnumerable<AppUser>
<div class="bg-primary panel-body"><h4>User Accounts</h4></div>
<table class="table table-condensed table-bordered">
    <tr><th>ID</th><th>Name</th><th>Email</th></tr>
    @if (Model.Count() == 0)
    {
        <tr><td colspan="3" class="text-center">No User Accounts</td></tr>
    }
    else
    {
        foreach (AppUser user in Model)
        {
            <tr>
                <td>@user.Id</td>
                <td>@user.UserName</td>
                <td>@user.Email</td>
            </tr>
        }
    }
</table>
<div class="row">
    <div class="col-md-3 left"><a class="btn btn-primary" asp-action="Register">注册</a></div>
    <div class="col-md-3 right"><a class="btn btn-primary" asp-action="Login">登录</a></div>
</div>
```

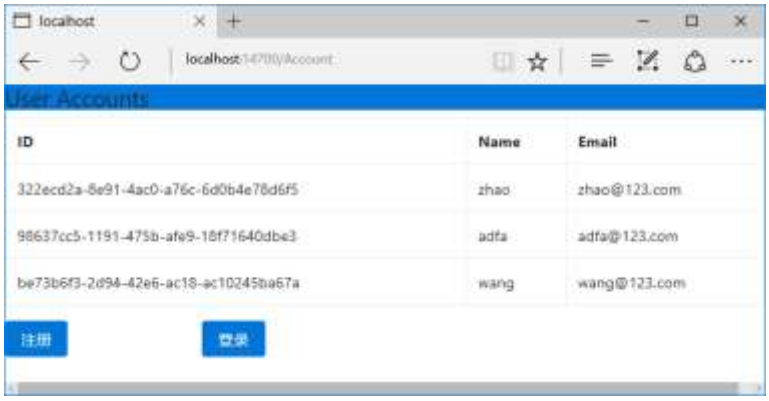
Register.cshtml

```
@model RegisterViewModel
@{
    ViewData["Title"] = "注册用户";
}

<h2>@ViewData["Title"].</h2>

<form asp-controller="Account" asp-action="Register" method="post" class="form-horizontal">
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="All" 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="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>
```

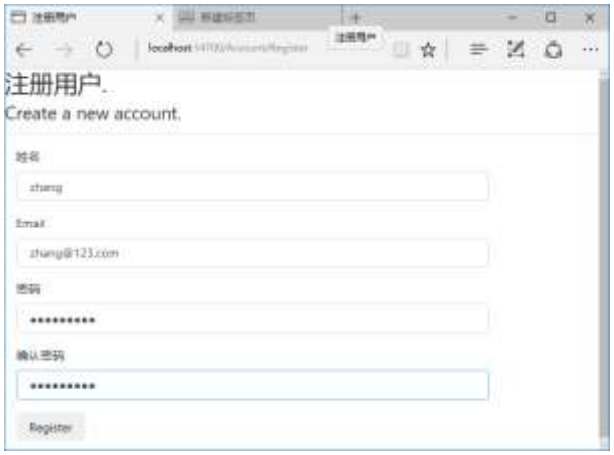
```
    </div>
</form>

Login.cshtml
@model LoginViewModel
@{
    ViewData["Title"] = "登录";
}

<h2>@ViewData["Title"]</h2>
<div class="row">
    <div class="col-md-8">
        <form asp-controller="Account" asp-action="Login" asp-route-returnurl="@ViewData["ReturnUrl"]"
method="post" class="form-horizontal">
            <h4>使用本地帐号登录.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="UserName" class="col-md-2 control-label">用户名：</label>
                <div class="col-md-10">
                    <input asp-for="UserName" class="form-control" />
                    <span asp-validation-for="UserName" class="text-danger"></span>
                </div>
            </div>
            <div class="form-group">
                <label asp-for="Password" class="col-md-2 control-label">密码：</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">
                <div class="col-md-offset-2 col-md-10">
                    <div class="checkbox">
                        <label asp-for="RememberMe">
                            <input asp-for="RememberMe" />
                            @Html.DisplayNameFor(m => m.RememberMe)
                        </label>
                    </div>
                </div>
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <button type="submit" class="btn btn-default">登录</button>
```

```
                </div>
            </div>
        </form>
    </div>
</div>
```

请认真理解以上代码，在理解的基础上，运行网站。运行方法是在 VS 主菜单点击 调试>开始执行(不调试)，然后在出现的页面的地址栏后面(前面的网址不要删除)，加上/Account，回车即可。显示页面如下：




点击注册，进入注册页：




点击 Register 按钮，又回到开始页。再点击登录，出现以下页：


点击登录，成功后回到开始页面。

## 13. 修改路由配置

ASP.NET Core MVC 框架的一个核心功能是路由解析和路由模式设置。路由解析是根据用户请求的 Url，分析所请求的应用程序的资源（内容或服务），对于 MVC 框架就是控制器、操作方法。如果将一个 Url 中

的协议名、服务器域名和查询字符串截去，剩下的部分是以/分隔的段。例如，设有以下 Url：

http://www.somecompany.com/Feature/Add?age=19&name=wang

http:为协议名称，www.somecompany.com 为服务器域名，?号后的 age=19&name=wang 为查询字符串。剩下部分 Feature/Add 表示两个段：Feature 和 Add。

路由解析就是要分析出这两个段的语义，采用的方法是与路由模式进行比较。以下是路由模式的例子：

{controller}/{action}

{controller}/{action}/{id}

Summary/{year}/{month}/{day}

路由模式也是由/分开的一些段。如果段用{}围住，则{}中的部分是变量；如果没有用{}围住，则段为静态段。匹配时，URL 中的段与模式的段的数量一样，且 URL 中与模式静态段对应的部分必须一样，而与变量对应的部分可以是任何内容。上面举例的 Url 只有两个段，只能与以上第一个模式匹配，且得到变量 controller 的值为 Feature, action 的值为 Add。

如果 Url 中的段为

Summary/2017/12/3

则可以与上面第三个模式匹配。year、month 和 day 分别为 2017、12 和 3。

模式中，一个段可以同时有静态值和变化值。如：

X{controller}/{action}

表示第一个段必须由 X 开头，而 controller 的取值不包含 X。例如 Url 段

Xhome/index

与以上模式匹配，且得到 controller 为 home，action 为 index。

在模式中，可以设置变量的缺省值，当 Url 没有给定对应的段时，变量值取缺省值。例如：

{controller=Home}/{action=Index}

也可以指定可选的段，方法是在变量名后用？标注。例如：

{controller=Home}/{action=Index}/{id?}

以上 id 是可选的。只有当前面两个段存在时，才会与 id 匹配。即以上路由模式可以匹配任何 0 段、1 段、2 段和 3 段的 Url。这个路由模式正是系统使用的缺省模式。Url 解析举例：

| Url 段 | controller | action | id |
|---|---|---|---|
| (0 段) | Home | Index | |
| Account | Account | Index | |
| Account/Balance | Account | Balance | |
| Account/Balance/12 | Account | Balance | 12 |

在代码中，使用 Startup.cs 文件中的 Configure 方法设置路由模式。目前 Configure 方法最后一句为：

```
app.UseMvcWithDefaultRoute();
```

该语句表明启动 MVC 框架服务，并使用缺省的路由模式。即{controller=Home}/{action=Index}/{id?}

可以删除以上语句，通过 app.UseMvc()方法设置不同的路由模式，也可以为应用设置多个模式。以下是设置一个路由模式的写法：

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

以下是设置多个路由模式的写法：

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

```
    routes.MapRoute(name: "", template: "Public/{controller=Home}/{action=Index}");
    routes.MapRoute(name: "ShopSchema", template: "Shop/{action}",   defaults: new { controller = "Home" });
});
```

当存在多个模式时，要注意匹配是按照由上到下的先后顺序进行的。多个模式如果摆放位置不当，会造成前面的模式屏蔽后面的模式。如果将以上第一个模式位置与第 2 个模式位置交换，则以 X 开头的两段路径

XAccount/Balance

会解析为 contoller=XAccount, action=Balance。这样，"X{controller}/{action}"永远都无法发挥作用，即被屏蔽了。

以上最后一个模式的含义是，如果 Url 中输入 Shop 为首段，则将 controller 值缺省地设定为 Home。这就相当于代码中已将原来名为 Shop 的控制器改名为 Home 控制器，但用户输入地址和页面中的地址链接可以不用改变以前的习惯和写法。这里可以看出路由模式的威力了。

有了以上了解，你可以自己试着修改 Startup 类，并试验路由解析效果(通过输入不同的 Url 地址)。

【全文完】