

ASP.NET MVC5 教程

译文链接: <http://www.cnblogs.com/DoduNet/p/3234076.html>

原文链接: <http://www.asp.net/mvc/tutorials/mvc-5/introduction/>

QQ 群 : 25380362 整理

目录

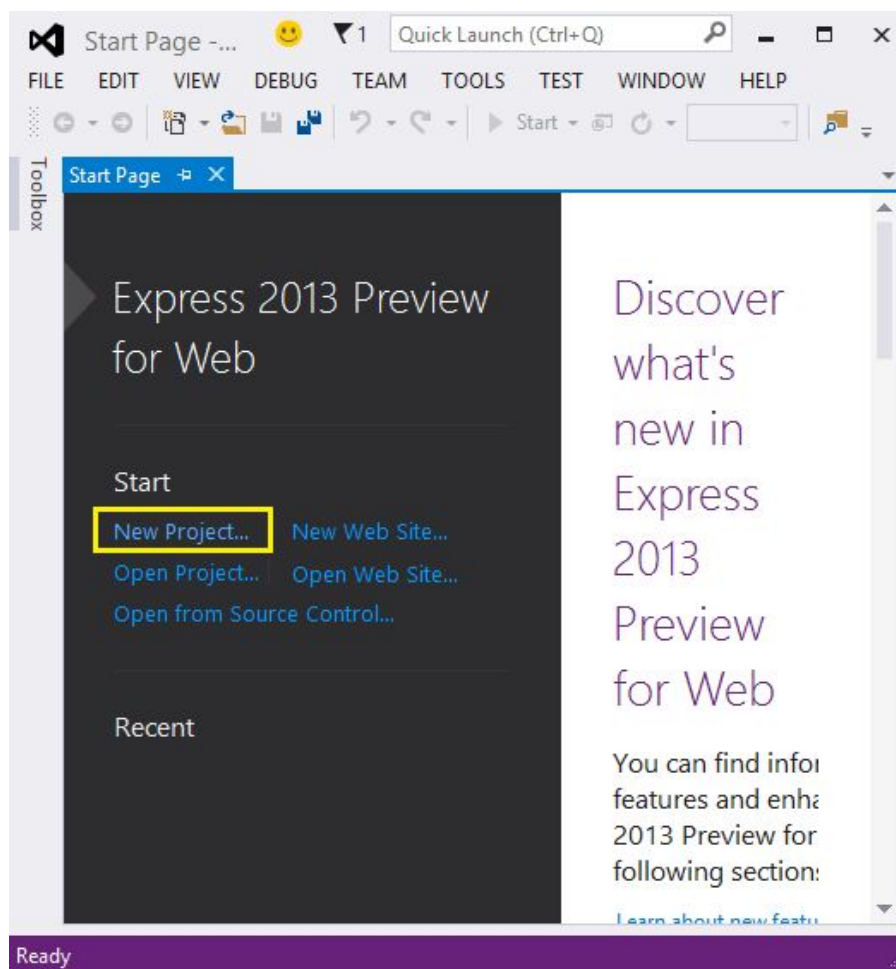
1: 入门.....	3
2: 添加控制器.....	8
3: 添加视图.....	15
4: 添加模型.....	28
5: 使用 SQL Server LocalDB 创建连接字符串.....	32
6: 通过控制器访问模型的数据.....	35
7: Edit 方法和 Edit 视图详解.....	44
8: 搜索查询.....	55
9: 添加新字段.....	62
10: 添加验证.....	74
11: Details 和 Delete 方法详解.....	84

1: 入门

本教程将教你使用 Visual Studio 2013 预览版构建 ASP.NET MVC 5 Web 应用程序 的基础知识。本主题还附带了一个采用 C# 源代码的 Visual Web Developer 项目。[下载 C# 版本](#)。

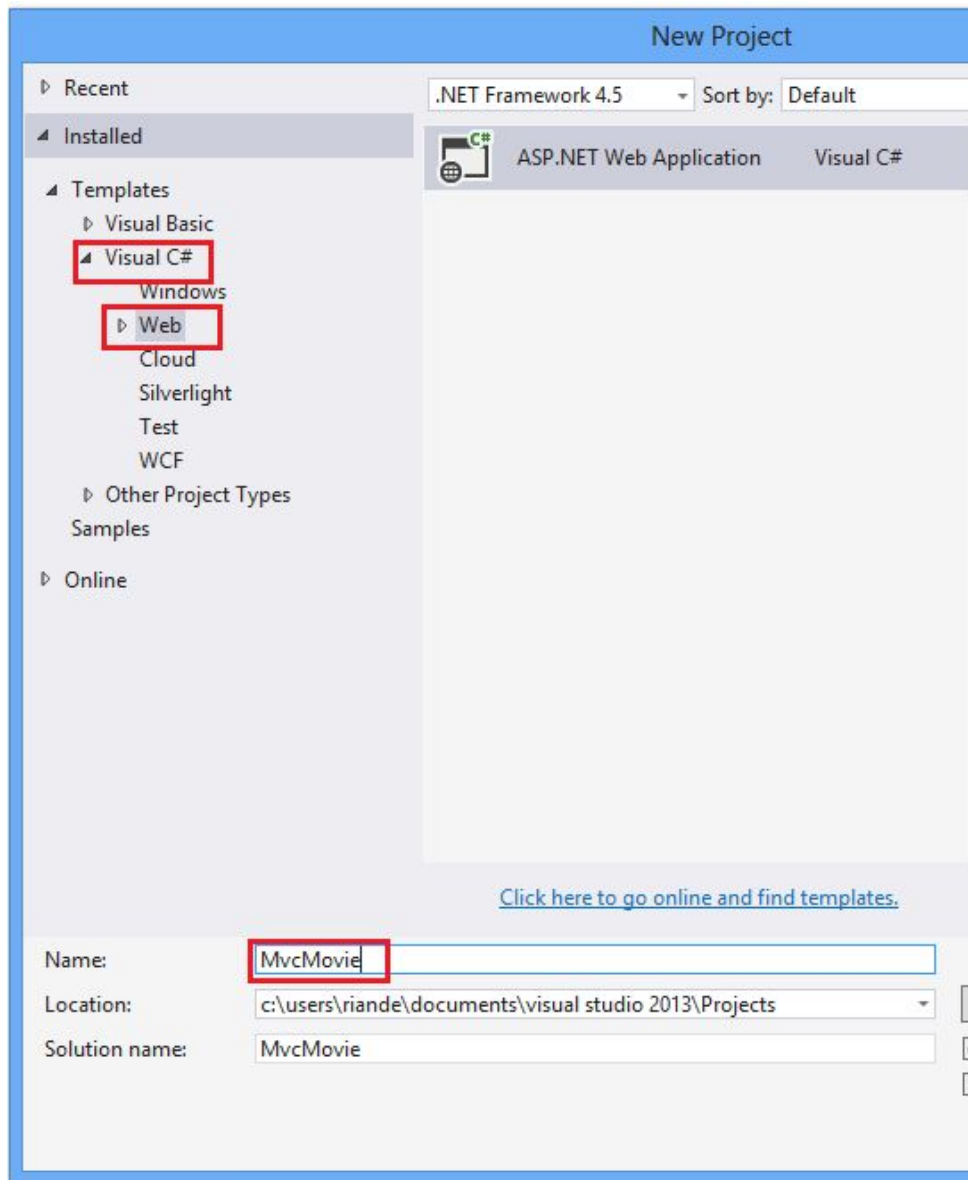
入门

Visual Studio 是一个集成的开发环境。就像您使用 Microsoft Word 写文档，您将使用 IDE 创建的应用程序。Visual Studio 的顶部有一个工具栏，其中显示了您可以使用的不同选项。还有一个菜单，提供了另一种在 IDE 中执行任务的方法。（例如，可以从起始页选择新的项目，或者使用菜单 选择 文件>新建项目。）

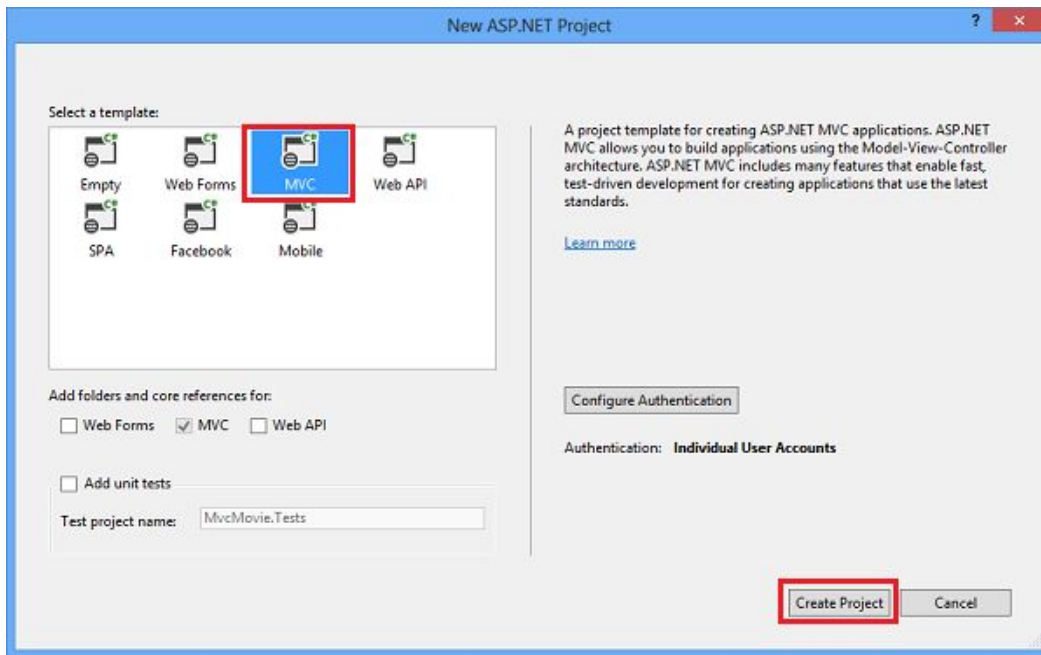


创建您的第一个应用程序

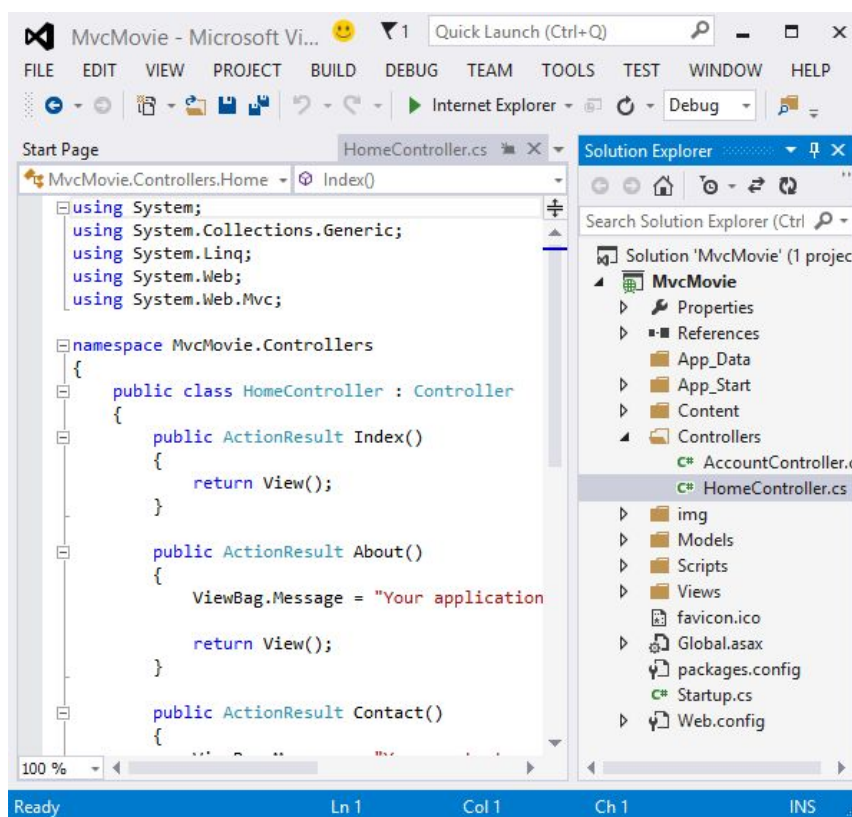
您可以创建使用 Visual Basic 或 Visual C# 作为编程语言的应用程序。单击新建项目，然后在左边选择 Visual C#，然后 WEB，然后选择 ASP.NET Web 应用程序。将您使用的项目名称称为："MvcMovie"，然后单击确定。



在新的 **ASP.NET** 项目对话框中，单击 **MVC** ，然后单击**创建项目**。

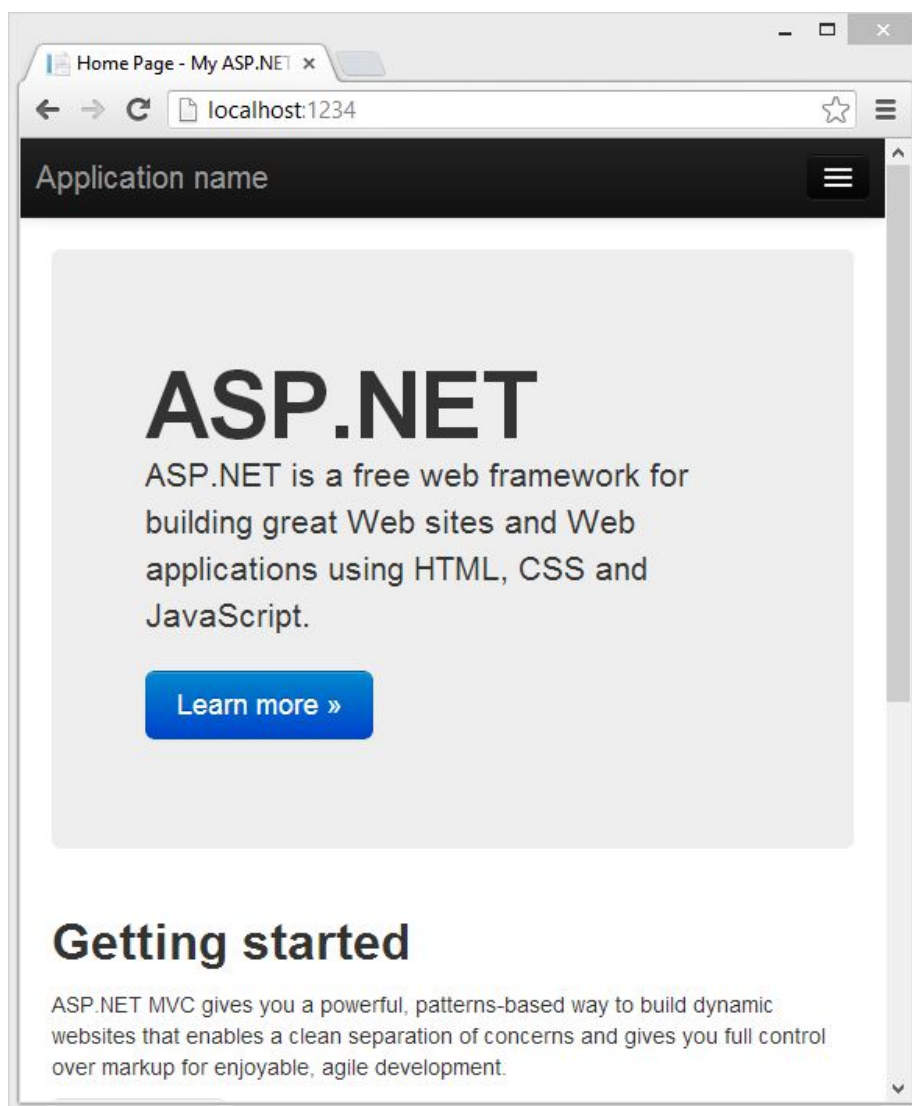


单击**确定**。会得到使用 Visual Studio 默认模板的 ASP.NET MVC 项目，这是一个简单的“Hello World!”项目，是你开发应用程序的一个好的开始。

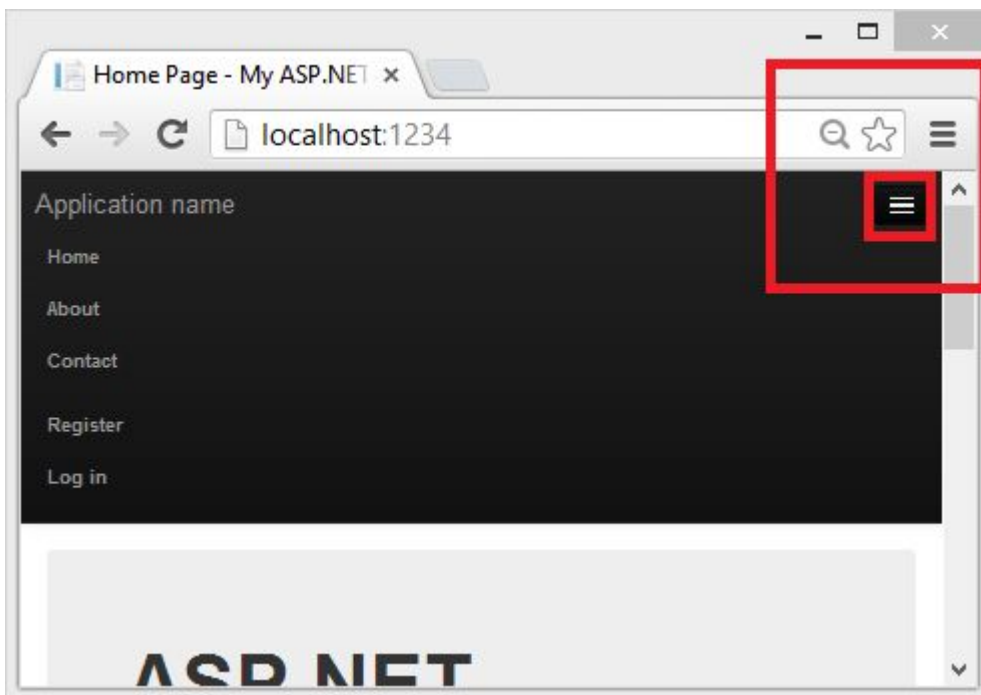
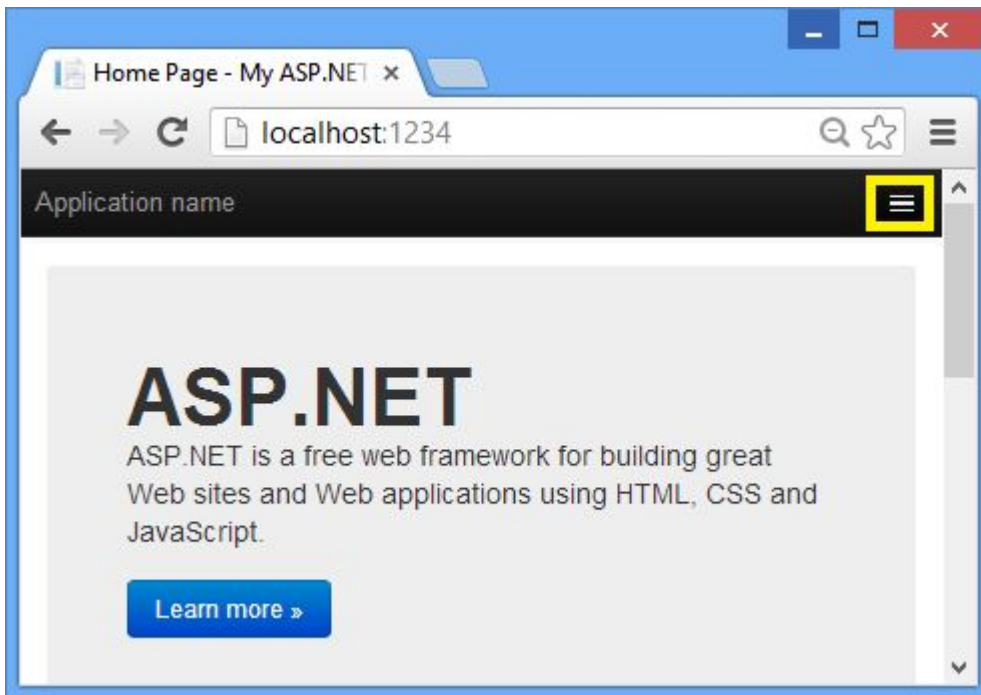


按 F5 启动调试。F5 导致 Visual Studio 启动 IIS Express 并运行您的 web 应用程序。Visual Studio 启动浏览器，然后打开该应用程序的主页。注意浏览器的地址栏显示的是 localhost，而不是类似于 example.com 这样的地址。这是因为 localhost 始终指向您自己的本地计算机，

在这种情况下你只是生成的应用程序运行。当 **Visual Studio** 运行时 **web** 项目中，会使用服务器的一个随机端口用于 此 **web** 应用。在下面的图像中，端口号是 **1234**。当您运行该应用程序时，您就会看到一个不同的端口号。



此默认模板直接为您提供了主页、 联系人和关于页面。上面的图片不能显示首页，以及联系人的链接。根据您的浏览器窗口的大小，您可能需要单击导航图标，看到这些链接。



应用程序还提供注册、登录的示例。

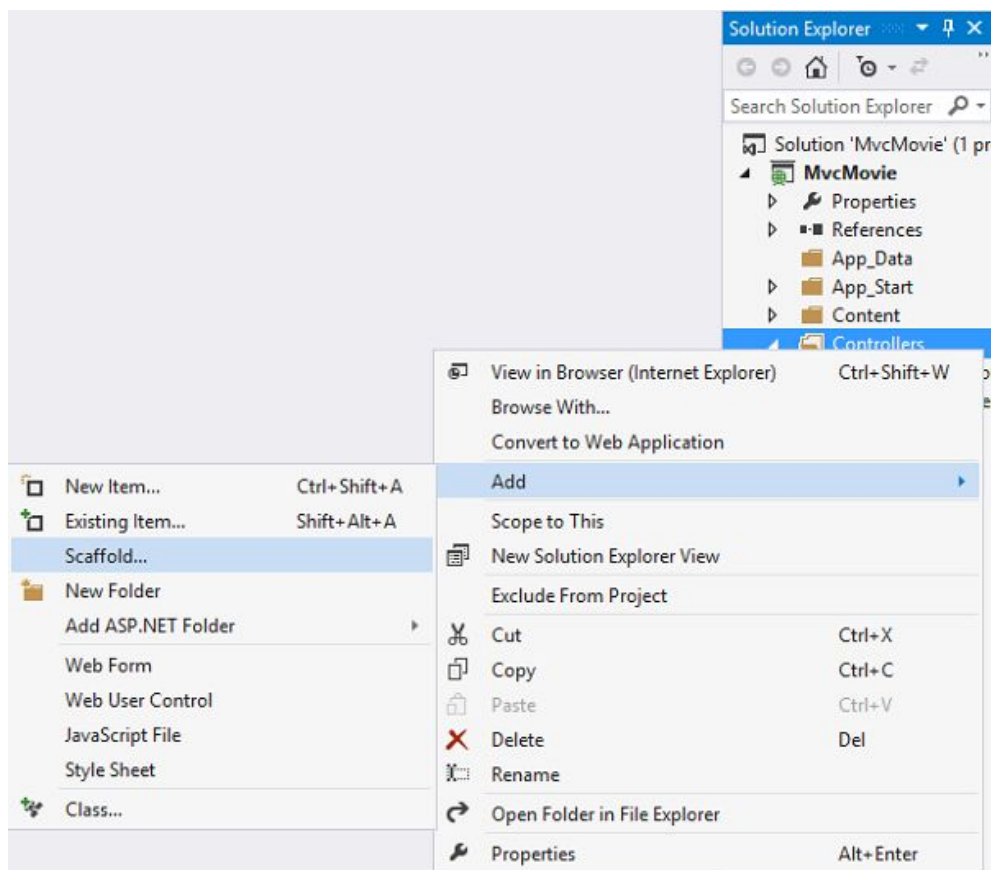
2: 添加控制器

MVC 表示 *模型-视图-控制器*。MVC 是一种用于开发应用程序的模式，具备良好架构，可测试和易于维护。基于 MVC 应用程序中包含：

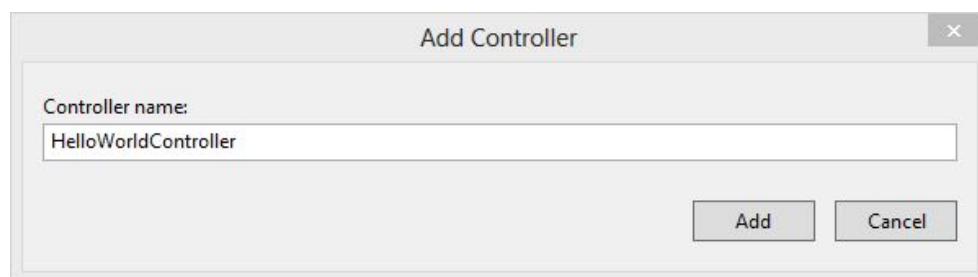
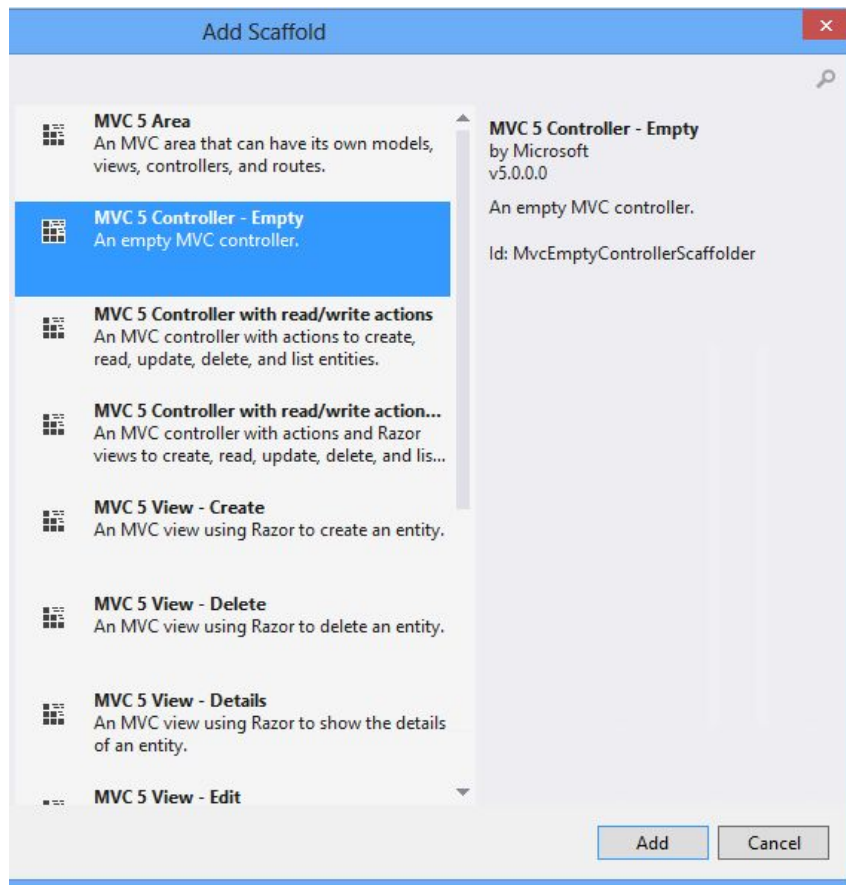
- **Models:** 表示应用程序的数据，并使用验证逻辑强制执行业务规则的数据的类。
- **Views:** 您的应用程序用来动态生成 HTML 响应的模板文件。
- **Controllers:** 处理传入的请求的浏览器，类检索模型的数据，然后指定将响应返回到浏览器中的视图模板。

在这个系列教程中涵盖了所有这些概念和教你怎么使用它们构建应用程序。

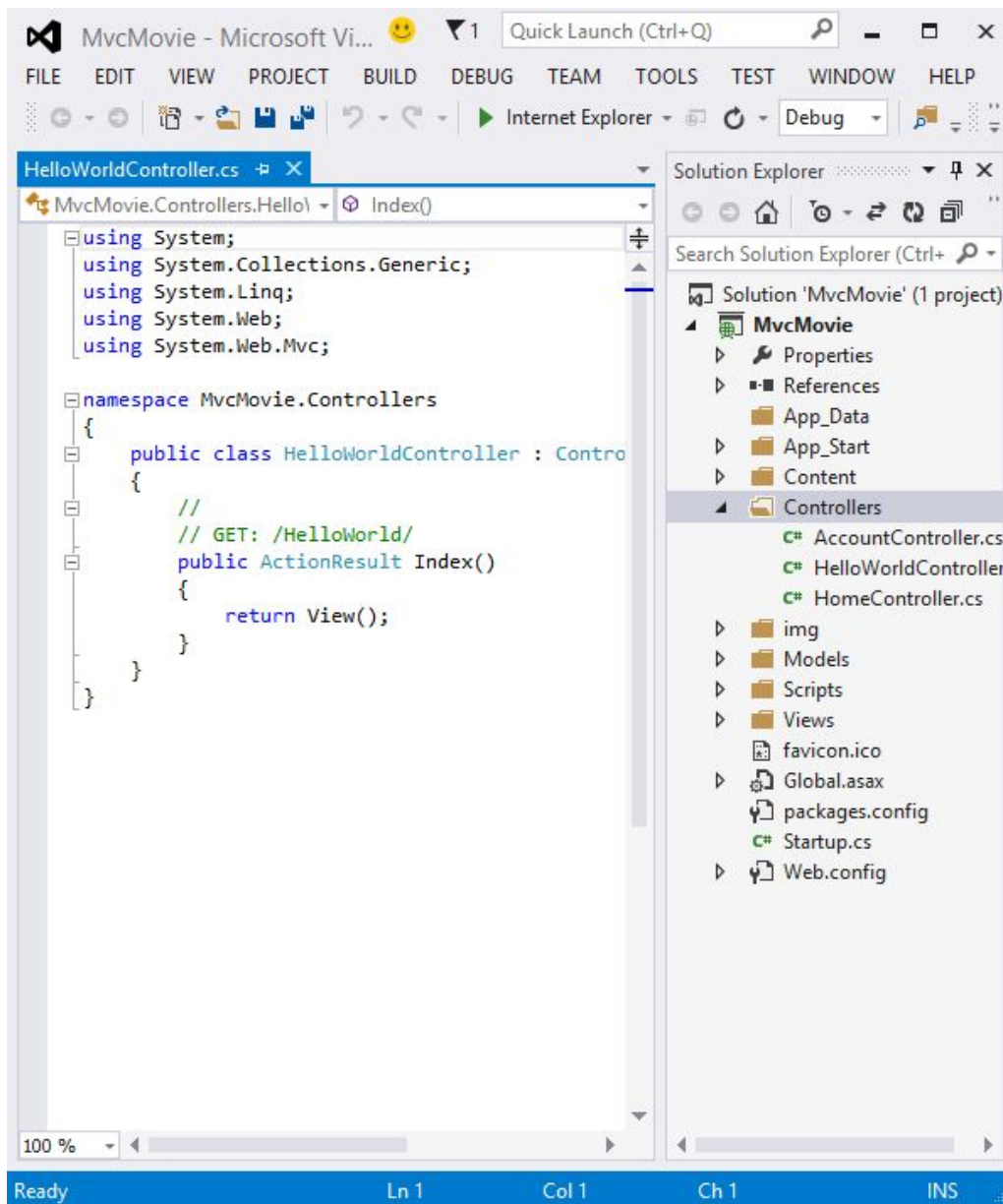
让我们开始创建一个控制器类。在 **解决方案资源管理器**中，右键单击 **Controllers** 文件夹，然后单击 **添加**，然后**支架**。



添加支架对话框中，单击 **MVC 5 控制器-空**，然后单击**添加**。



注意在 解决方案资源管理器 已创建的新文件命名 *HelloWorldController.cs*。在 IDE 中打开该文件。



写入下面的代码：

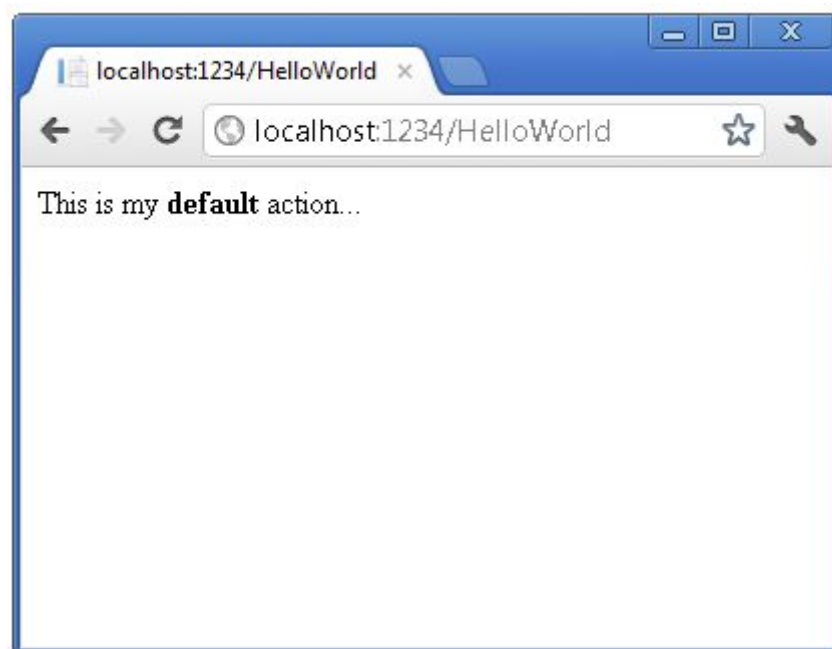
```
using System.Web;
using System.Web.Mvc;
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        // GET: /HelloWorld/
        public string Index()
        {
            return "This is my <b>default</b> action...";
        }
    }
}
```

```
// GET: /HelloWorld/Welcome/
public string Welcome()
{
    return "This is the Welcome action method...";
}
}
```

示例中，控制器的类名为 **HelloWorldController**，继承于：**Controller**，上述第一种方法为 **Index**。让我们从浏览器中调用它。

运行该应用程序（按 **F5** 或 **Ctrl + F5**）。在浏览器中，将"HelloWorld"添加到地址栏上的路径后面。（例如，在图中，它下面的 <http://localhost:1234/HelloWorld>。）

页面在浏览器中的将看到如下图所示的页面。在上述方法中，代码直接返回一个字符串。



ASP.NET MVC 会根据传入的 URL 调用不同的控制器类（以及其中的不同操作方法）。ASP.NET MVC 所使用的默认 URL 路由逻辑使用这样的格式来确定哪些代码来调用：

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

在 *App_Start/RouteConfig.cs* 文件中可以设置路由的格式。

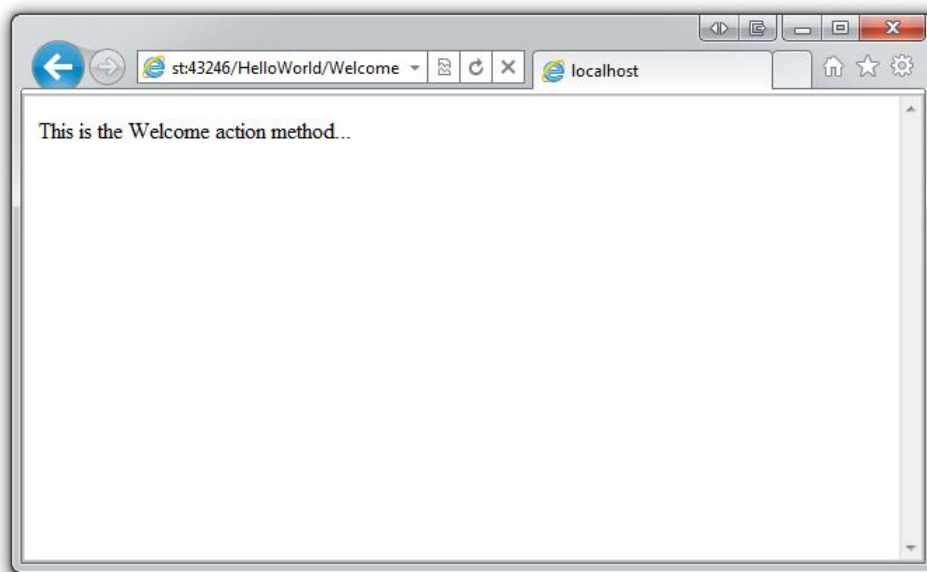
```
public static void RegisterRoutes(RouteCollection routes){
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
```

```
defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
); }
```

当您运行该应用程序并不提供任何 URL 部分时，它将默认的"Home"的控制器和上面代码的默认值部分中指定的"Index"的操作方法。

URL 的第一部分确定要执行的控制器类。所以 **/HelloWorld** 映射到 HelloWorldController 类。第二部分 URL 的确定要执行的类上的操作方法。所以 **/HelloWorld/Index** 会找到 HelloWorldController 类的 Index 的方法执行。请注意，我们只在浏览器中看到 **/HelloWorld**，而并无 Index。这是因为命名 Index 的方法被指定一个称为控制器的默认方法。URL 段（Parameters）的第三部分是用于路由数据。我们将在本教程中稍后看到路由配置的资料。

下面在浏览中打开 <http://localhost:xxxx/HelloWorld/Welcome>。Welcome 方法运行，并返回字符串 "This is my **default** action..."。MVC 的默认映射是 **/[Controller]/[ActionName]/[Parameters]**。对于这个 URL，控制器是 HelloWorld，而 Welcome 是操作方法。您还没有使用 URL 的 [Parameters] 部分。



让我们将此示例稍微修改一下，以便您可以将一些参数信息从 URL 中传递给控制器（例如，**/HelloWorld/Welcome?name=Scott&numtimes=4**）。更改您 Welcome 方法，包括两个参数，如下所示。请注意，该代码使用 C# 的可选参数功能来指示：当没有为 numTimes 参数传递值时，该参数的默认值应为 1。

```
public string Welcome(string name, int numTimes = 1) {  
    return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);  
}
```

运行您的应用程序，然后在浏览中打开 URL：

<http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4>

在 URL 中您可以尝试不同的值 name 和 numtimes。在 [ASP.NET MVC 模型绑定系统](#)

从地址栏中命名的参数将自动映射给您的方法中指定的参数。



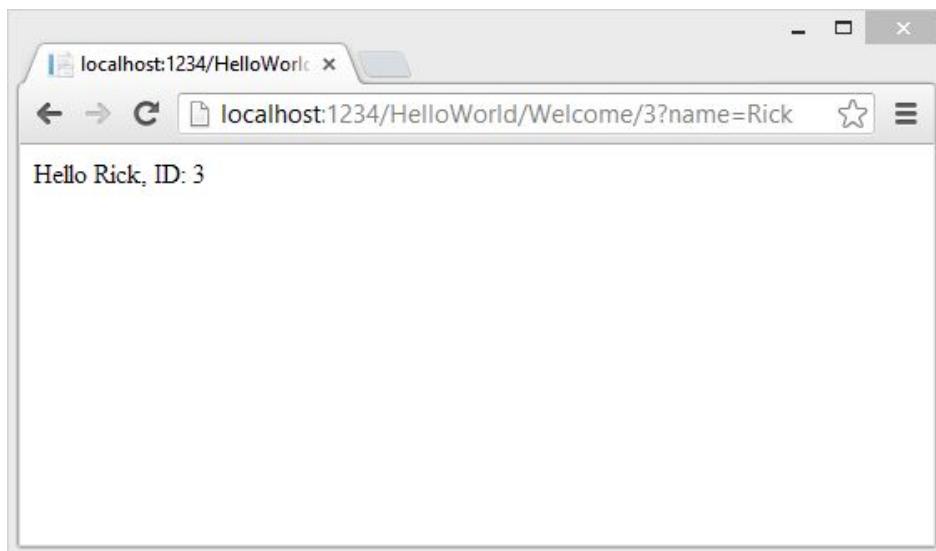
在上面的示例中，不使用 URL 段（Parameters），作为查询字符串传递的 `name` 和 `numTimes` 参数。

`Welcome` 方法替换为以下代码：

```
public string Welcome(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);
}
```

运行应用程序，然后输入以下 URL：

`http://localhost:xxx/HelloWorld/Welcome/3?name=Rick`



这次 URL 的第三个部分匹配 `ID`，因为 `Welcome` 的操作方法有一个匹配的 URL 规范在 `RegisterRoutes` 方法中的参数 (`ID`)。



```
public static void RegisterRoutes(RouteCollection routes){
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

```

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);}

```

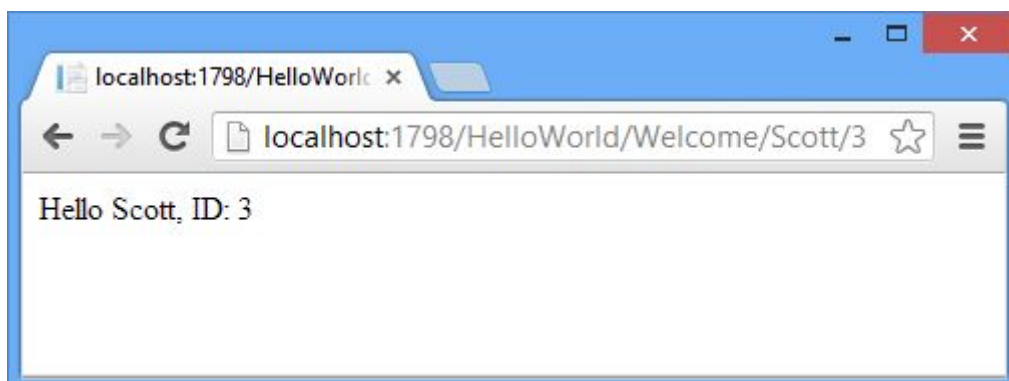
在 ASP.NET MVC 应用程序中，这是默认的路由配置（像刚才那样用 ID）。您也可以添加一个路由，这两个 name 传递给和 numtimes 作为路由数据的 URL 中的参数。在 *App_Start\RouteConfig.cs* 文件中，添加"Hello"的路由配置：

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
        routes.MapRoute(
            name: "Hello",
            url: "{controller}/{action}/{name}/{id}"
        );
    }
}

```

在浏览器中打开 `/localhost:XXX/HelloWorld/Welcome/Scott/3`。



对于许多的 MVC 应用程序都默认路由来正常工作。稍后在本教程中，通过使用模型程序，数据，您将了解到，所以一般不必修改默认路由。

3: 添加视图

在本节内容中，我们将修改 `HelloWorldController` 类，使用视图模板来干净利索的封装生成 HTML 响应客户端的过程。

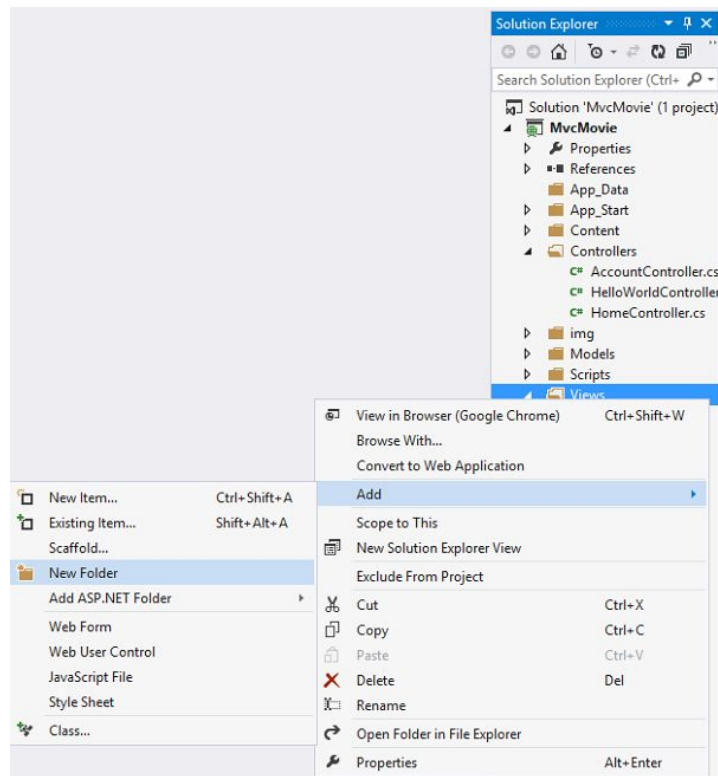
您将创建一个使用 Razor 视图引擎的视图模板文件。`.cshtml` 扩展名的文件都是基于 razor 视图模板文件，Razor 视图引擎将编写视图模板所需的代码降至最低。

目前的 `Index` 方法返回一条消息，是在控制器类中直接写入的字符串。更改 `Index` 方法使其返回一个 `View` 对象，如以下代码所示：

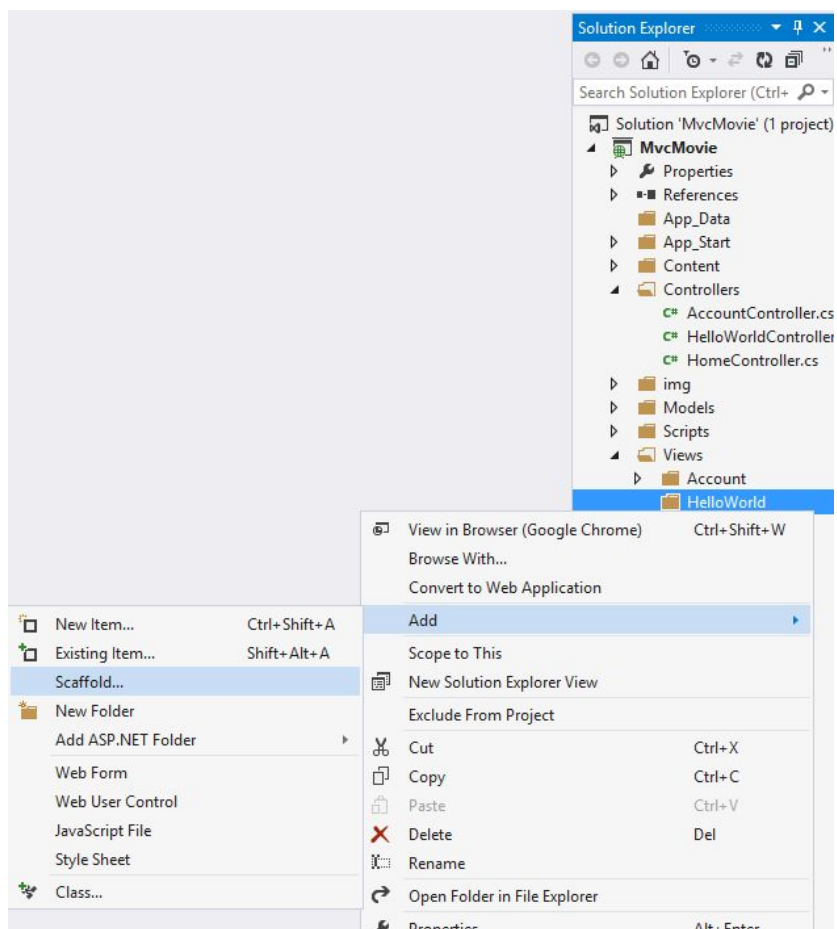
```
public ActionResult Index()
{
    return View();
}
```

上面的 `Index` 方法使用一个视图模板来生成浏览器所需的 HTML 代码。控制器方法（也称为操作方法），如上述，`Index` 方法通常返回的 `ActionResult`（或从 `ActionResult` 派生的类），但他不像是基元类型的字符串。

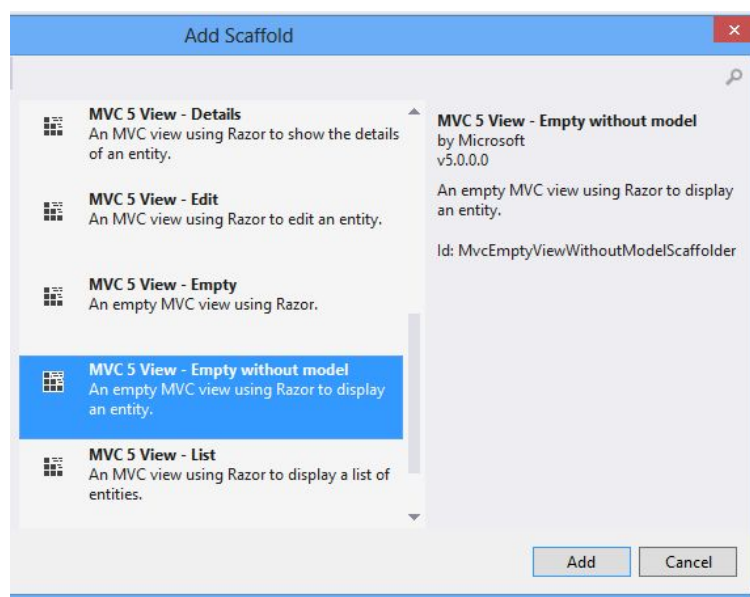
首先，对 `HelloWorld` 控制器创建一个视图文件夹。右键单击 "`View`"，单击添加，然后单击新建文件夹。



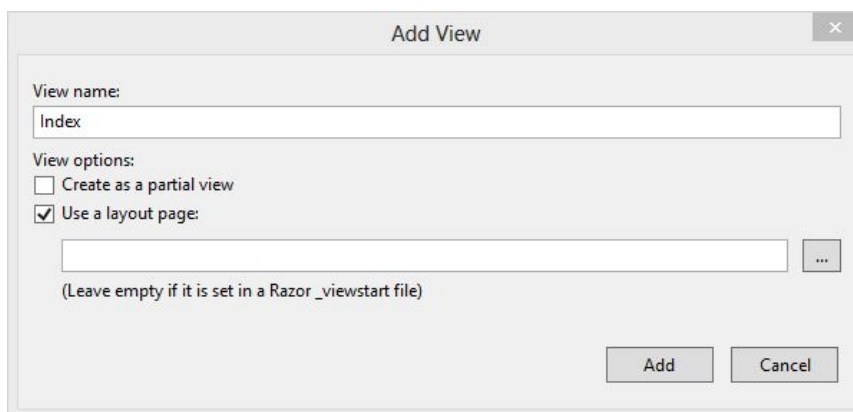
右键单击 *HelloWorld* 文件夹并单击添加，然后单击支架。



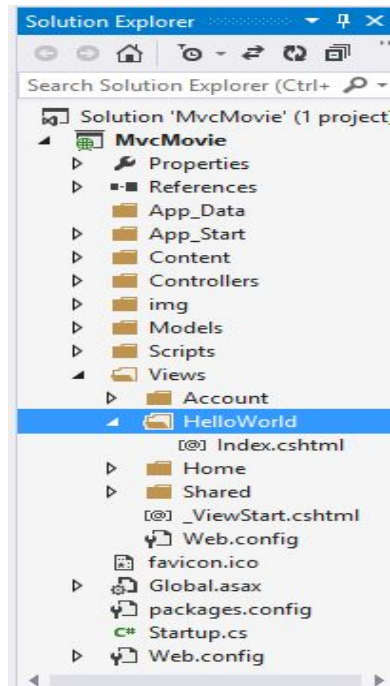
添加支架对话框中，单击**5 视图 MVC-空无模型**。然后单击**添加**。



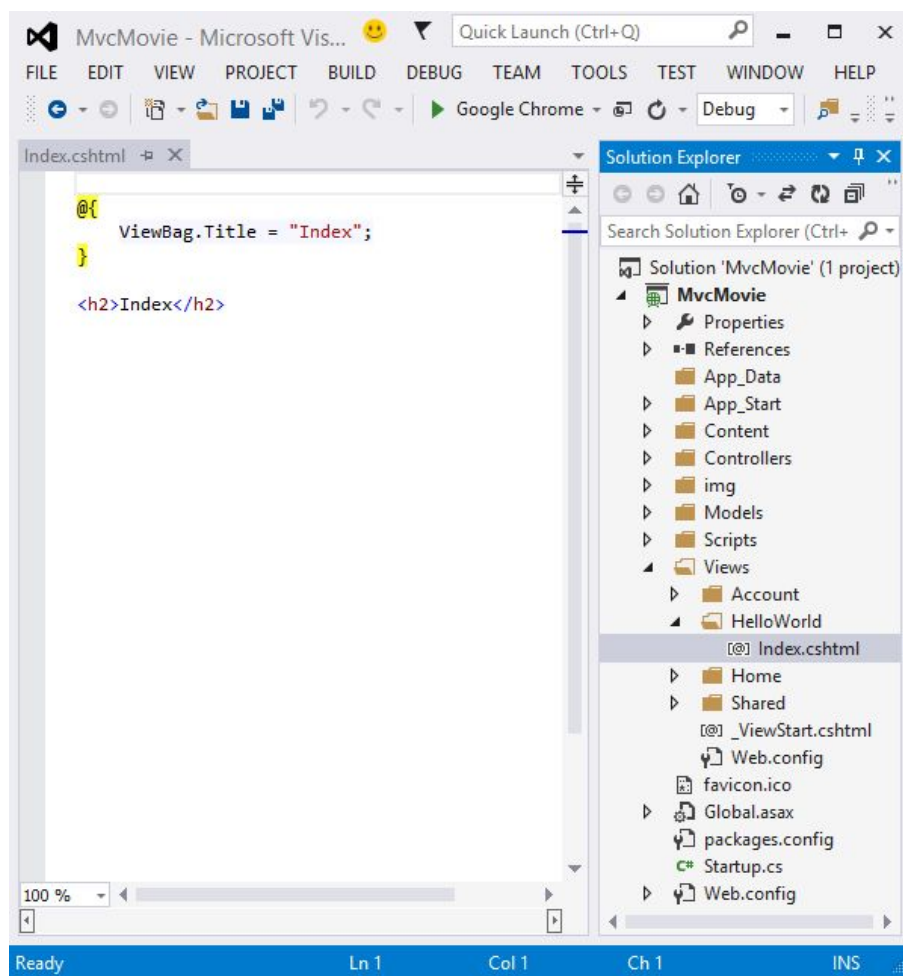
在**添加视图**对话框中，命名视图 **Index** 和保留的其他设置的默认值，然后单击**添加**。



创建的 *MvcMovie\Views\HelloWorld\Index.cshtml* 文件。



下面显示所创建的 *Index.cshtml* 文件:



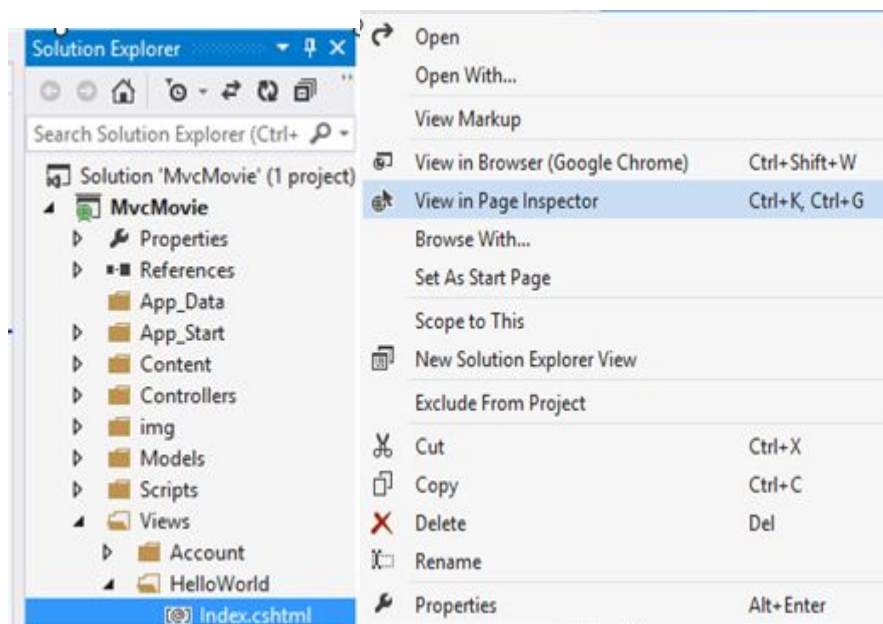
添加下面的 **html** 代码<h2>标记下。

<p>Hello from our View Template!</p>

下面是一个完整的 *MvcMovie|Views|HelloWorld|Index.cshtml* 文件。

```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>Hello from our View Template!</p>
```

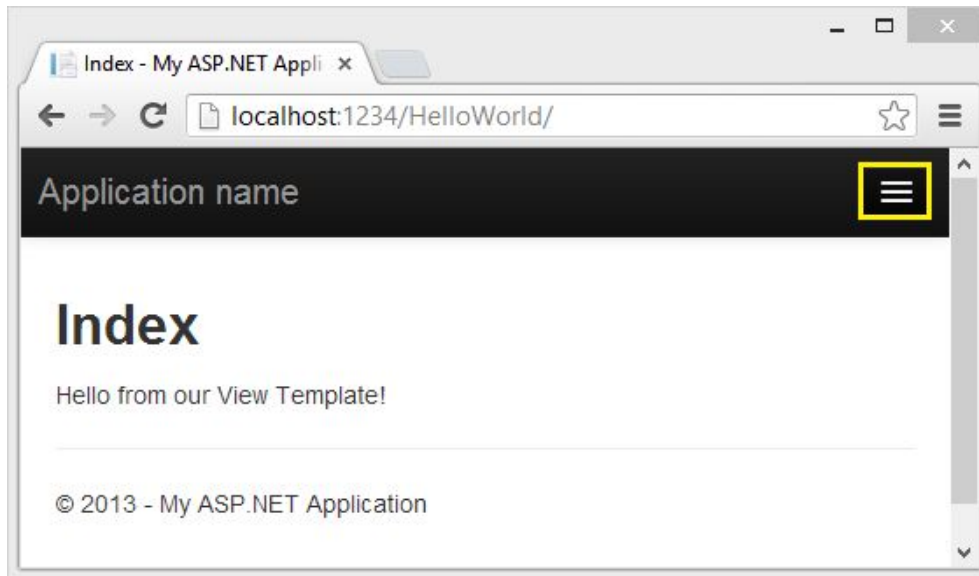
右键单击 *Index.cshtml* 文件，然后选择 View in Page Inspector.



请参见[教程页面检查器](#)的详细信息。

另外，运行程序，浏览到 **HelloWorld** 控制器（<http://localhost:xxxx/HelloWorld>）。控制器中的 **Index** 方法并没有做太多工作，它只是简单的运行了语句 `return View()`，指定 **Index** 方法应该使用一个视图模板文件呈现到浏览器的响应。

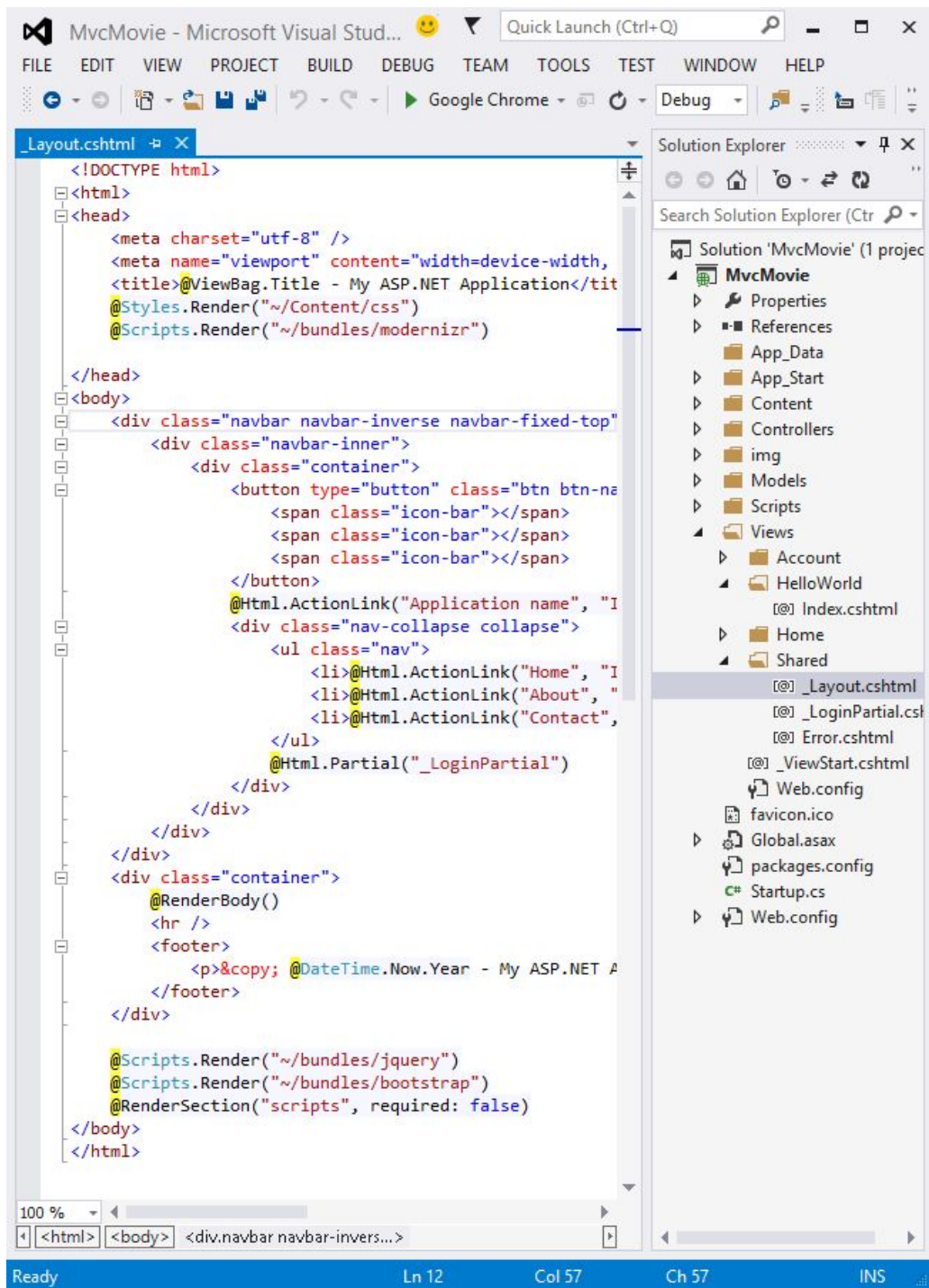
因为你没有明确指定要使用的视图模板文件的名称，**ASP.NET MVC** 默认使用 *|Views|HelloWorld* 文件夹下的 *Index.cshtml* 文件。下面图片中显示的“Hello from our View Template!”硬编码在视图中。



看起来还不错。但是，注意浏览器标题栏显示的“Index - My ASP.NET Ap”和在页面顶部的大个儿的“Application name”链接。根据你窗口大小不同，你可能会看到右上角的三条杠，点击后会看到 Home、About、Contact、Register 和 Log in 链接。

更改视图和布局页面

首先，你希望改变页面顶部的“Application name”链接，这段文字在每个页面都有，是公用的。尽管它出现在程序中的每个页面，但实际上它只写在一个地方。在解决方案资源管理器中找到 */Views/Shared* 文件夹，打开 *_Layout.cshtml* 文件。这个页面叫做布局页，放在所有页面都能用的共享文件夹中。



```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
  <title>@ViewBag.Title - My ASP.NET Application</tit
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <button type="button" class="btn btn-na
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      @Html.ActionLink("Application name", "I
      <div class="nav-collapse collapse">
        <ul class="nav">
          <li>@Html.ActionLink("Home", "I
          <li>@Html.ActionLink("About", "
          <li>@Html.ActionLink("Contact",
        </ul>
        @Html.Partial("_LoginPartial")
      </div>
    </div>
  </div>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>&copy; @DateTime.Now.Year - My ASP.NET A
    </footer>
  </div>
  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/bootstrap")
  @RenderSection("scripts", required: false)
</body>
</html>
```

布局模板允许你在页面的某个地方指定 HTML 容器，然后在网站多个页面中应用。找到 `@RenderBody()` 行，`RenderBody` 是一个占位符，所有你使用了 `_layout.cshtml` 文件的视图页面会被显示在这个地方，“包装”在布局页中。

例如，如果你选择 `About` 链接，视图 `Views|Home|About.cshtml` 将被绘制在 `RenderBody` 方法中。

修改布局模板中的 `ActionLink`，将“Application name”改为“MVC Movie”。




```

<body>
  <div class="navbar navbar-inverse navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <button type="button" class="btn btn-navbar" data-toggle="collapse"
data-target=".nav-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("MVC Movie", "Index", "Home", null, new { @class = "brand" })
        <div class="nav-collapse collapse">
          <ul class="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>
</div>

```

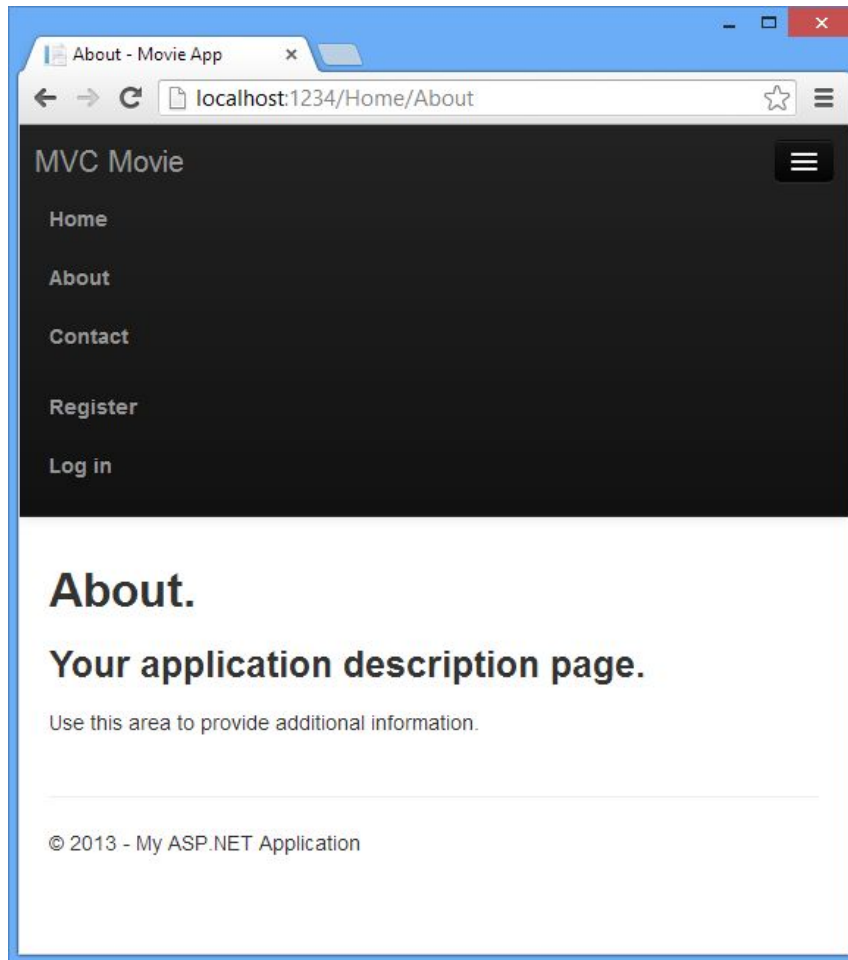
用下面的标签替换 **title** 元素的内容：

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - Movie App</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>

```

运行应用程序，现在已将变成“MVC Movie”了。单击 **About** 链接，你会看到这个页面也显示“MVC Movie”。我们仅仅修改了布局模板，就为网站中的所有页面换上了新的标题。



现在让我们来改变 `Index` 视图中的 `title` 吧。

打开 `MvcMovie|Views|HelloWorld|Index.cshtml`。有两个地方需要我们修改：浏览器标题栏中的文本，然后是次要的标题（`<h2>`元素）。你可以稍微修改一下代码，这样就可以看到哪些代码影响了哪些部分。

```
@{  
    ViewBag.Title = "Movie List";  
}  
<h2>My Movie List</h2>  
<p>Hello from our View Template!</p>
```

为了指明 HTML 显示的标题，上面的代码中设置了 `ViewBag` 对象（在 `Index.cshtml` 视图模板中）的 `Title` 属性。在布局模板中（`Views|Shared|_Layout.cshtml`）的 `<head>` 节点的 `<title>` 标签使用了这个值。

```
<!DOCTYPE html>  
<html>
```

```

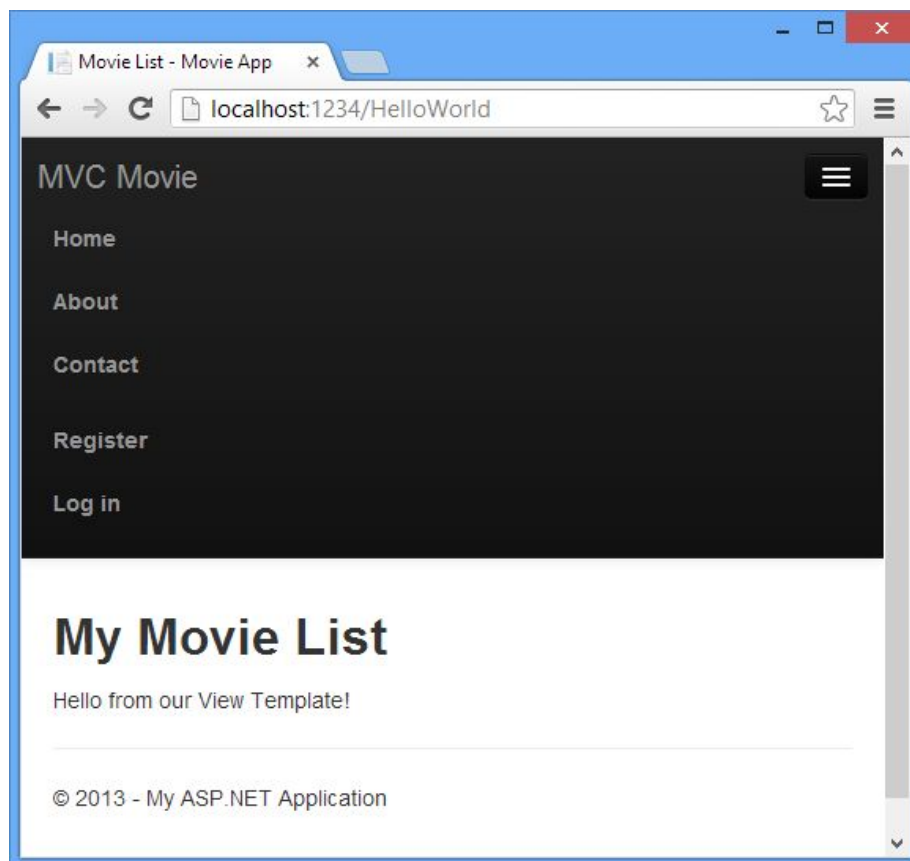
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - Movie App</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>

```

使用此 **ViewBag** 的方法，你可以轻松地其他参数之间传递您的视图模板和布局文件。

运行应用程序并浏览到 <http://localhost:xx/HelloWorld>。注意这个页面发生的变化：浏览器标题、主标题、此标题都发生了改变（如果你没有看到这些变化，可能是浏览器缓存了之前的内容，在浏览器中按 **CTRL+F5** 强制从刷新页面）。

浏览器标题使我们在 *Index.cshtml* 中传递给布局页的参数，布局页又加上了“- *Movie App*”部分。通过这个例子你会看到，布局模板提供了一种简单的修改应用程序中全部页面的方式。



到目前为止，我们用到的少量数据（像上面例子中的“Hello from our View Template!”）都是硬编码的。我们用到了 MVC 中的“V”（View）和“C”（Controller），但还没用到过“M”（Model）。

接下来我们就演练一下如何创建数据库，并通过模型来获取数据。

将数据从控制器传递到视图

在讨论数据库和模型之前，让我们先说说容控制器传递数据给视图。控制器类在响应传入的

请求时被调用，控制器类是你写代码处理浏览器请求、从数据库中检索数据、并最终决定发送什么类型的响应给浏览器。视图模板被控制器用来生成和格式化 HTML 响应给浏览器。

控制器的责任是为视图模板提供必须的数据或对象，用来绘制 HTML 响应浏览器。一个最佳实践是：视图模板从来不参与业务逻辑，或直接与数据库交互。相反的，视图模板仅与控制器提供的数据一起工作。保持这种“关注点分离（separation of concerns）”有助于保持代码的整洁，可测试性和更容易维护。

目前，HelloWorldController 类中的 *Welcome* 方法需要两个参数：*name* 和 *numTimes*，然后直接与将值输出给浏览器。让我们修改控制器，使用视图来替换直接相应 string 字符串。视图模板会生成一个动态响应，这意味着你需要通过控制器传递一些数据用来生成响应。

要做到这些，你需要通过在控制器中将数据（参数）放到 ViewBag 对象中，视图可以访问 ViewBag 对象。

回到 *HelloWorldController.cs* 文件中，修改 *Welcome* 方法，在 ViewBag 对象中添加一个 Message 和 NumTimes 值。ViewBag 是 dynamic 类型的对象，你可以为它添加任何你想要的数据，ViewBag 对象在你添加数据之前，不具有任何属性。

ASP.NET MVC 模型绑定系统从地址参数中自动映射命名的参数（*name* 和 *numTimes*）到方法中。完整的 *HelloWorldController.cs* 文件如下：

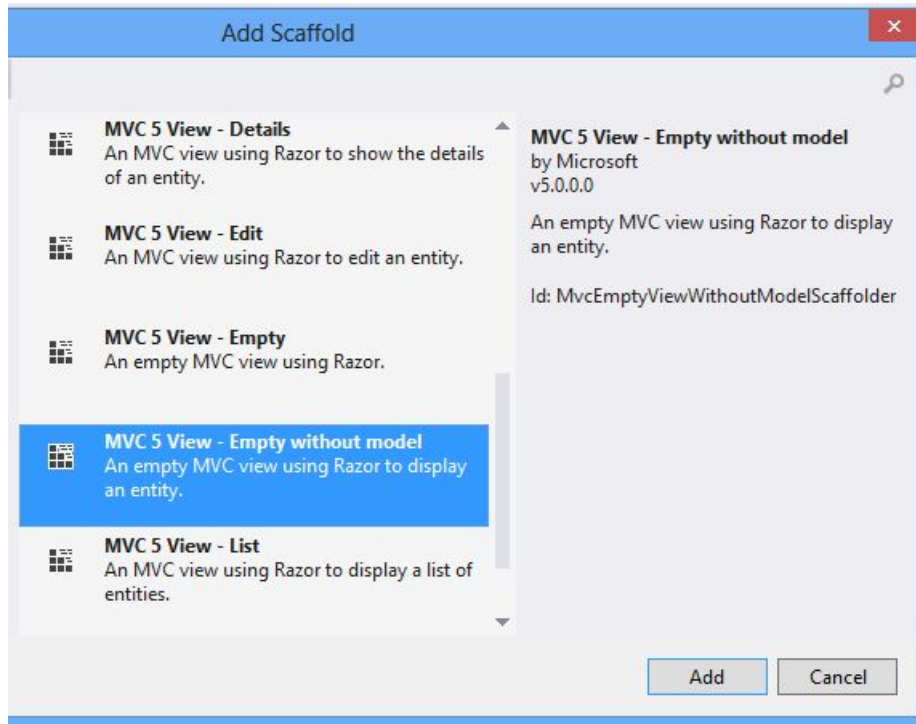
```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;
            return View();
        }
    }
}
```

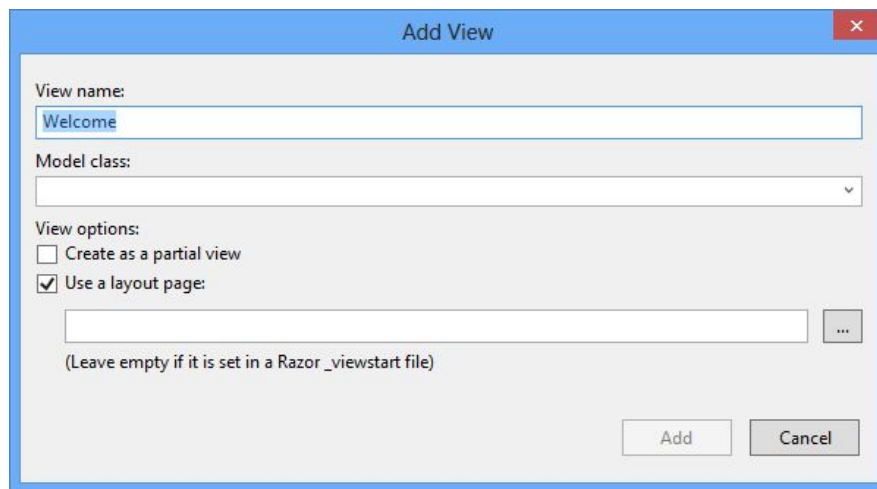
现在 ViewBag 对象已经包含了数据，它会被自动传递给视图。

接下来，你需要一个 *Welcome* 视图模板。在生成菜单中，选择生成解决方案（或使用快捷键 CTRL+SHIFT+B）确定项目已经编译了。

在 *Views\HelloWorld* 文件夹右键，选择“添加”>“支架”：



在 Add View 对话框中，将视图命名为 Welcome



文件 *MvcMovie|Views|HelloWorld|Welcome.cshtml* 已经创建好了。

在文件 *Welcome.cshtml* 的 `<h2>` 元素下添加如下的代码，完整的代码如下：

```
@{
    ViewBag.Title = "Welcome";
}

<h2>Welcome</h2>
<ul>
```

```

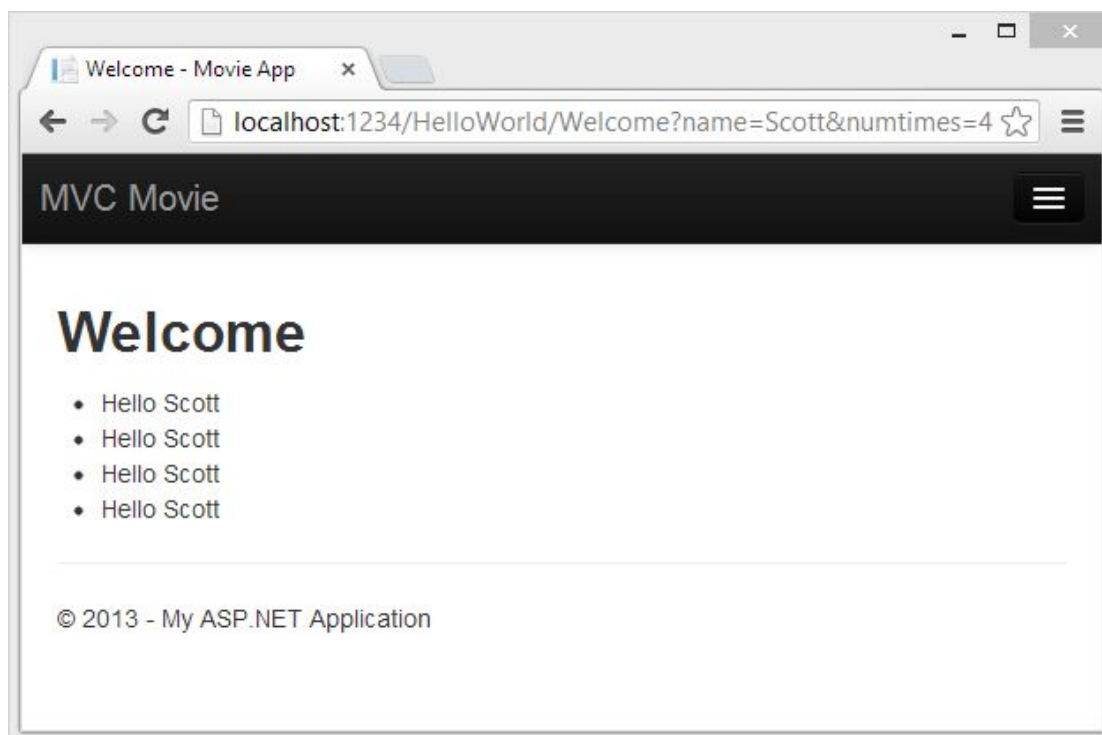
    @for (int i = 0; i < ViewBag.NumTimes; i++)
    {
        <li>@ViewBag.Message</li>
    }
</ul>

```

运行应用程序，在浏览器中查看如下地址：

<http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4>

现在数据从 URL 取出来，通过模型绑定传递给控制器，控制器将数据封装在 ViewBag 对象中，传递给视图，然后视图将数据以 HTML 的方式呈现给用户。



在上面的例子中，我们使用 ViewBag 对象将数据从控制器传递给视图。在家下来的章节中，我们会使用视图模型来传递数据。使用视图模型传递数据比用 ViewBag 要好得多。

这也是模型“M”的一种，但并没有使用数据库。我们接下来要学习的是创建一个数据库，创建一个真正意义的视图模型。

4: 添加模型

在本节中，我们将添加一些管理电影数据库的类，这些类在 ASP.NET MVC 应用程序中扮演“Model”的角色。

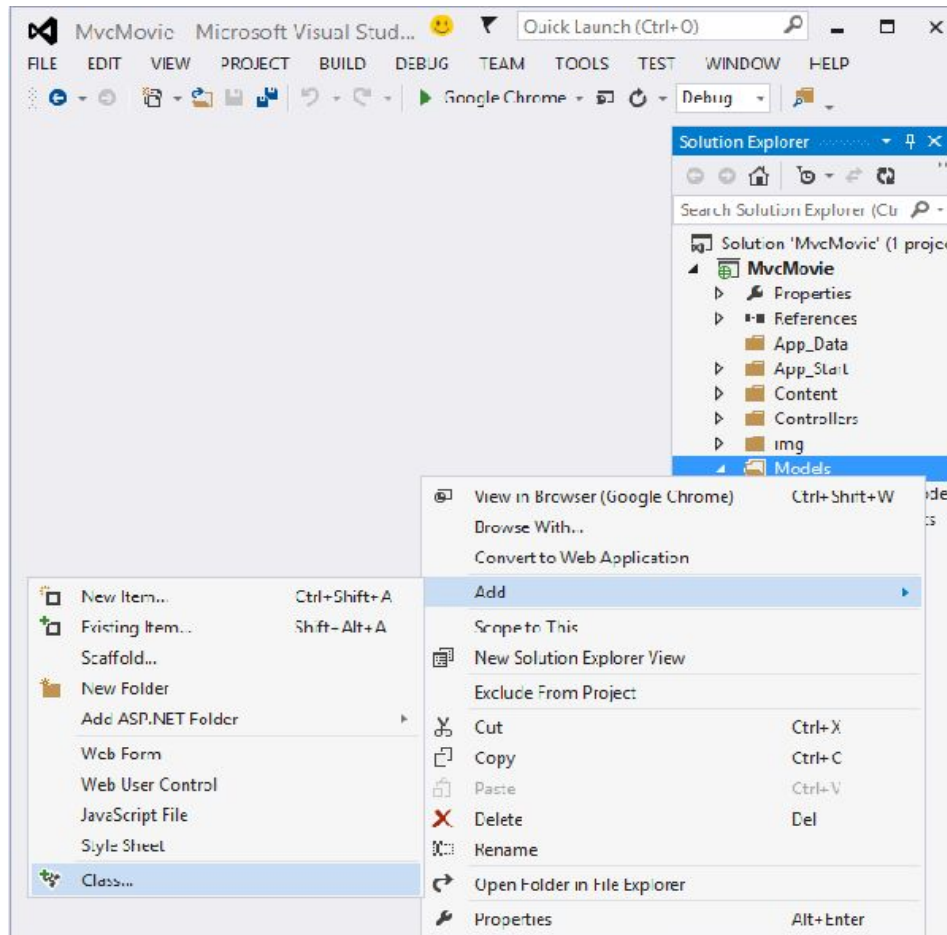
我们将使用 .NET Framework 平台上熟知的 Entity Framework 数据访问技术来定义和使用

这些模型。Entity Framework（简称 EF）提供了一种叫做 *Code First* 的开发范例。

Code First 允许你写一些简单的类来创建模型 这些通常被称为 POCO 类，即“plain old CLR object”）。这些类将会生成数据库，这是一种即简洁又快速的开发流程。

添加 Model 类

在解决方案资源管理器中，右键 *Models* 文件夹，选择 "添加">"类"。



输入类名“Movie”。为 Movie 类添加一下5个属性，完整的 Movie 类代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
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; }
    }
}

```

我们将使用 **Movie** 类代表数据库中的电影。每一个 **Movie** 对象将对应数据表中的一行，**Movie** 类的每一个字段映射数据表中的每一列。

在上面的文件中，添加一个 **MovieDbContext** 类：

```

using System;
using System.Data.Entity;
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; }
    }
    public class MovieDbContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}

```

MovieDbContext 类代表了 Entity Framework 中 **Movie** 类的数据库上下文，用来处理获取、存储和更新数据库中的 **Movie** 类的实例。**MovieDbContext** 类继承自 Entity Framework 中提供的 **DbContext** 类。

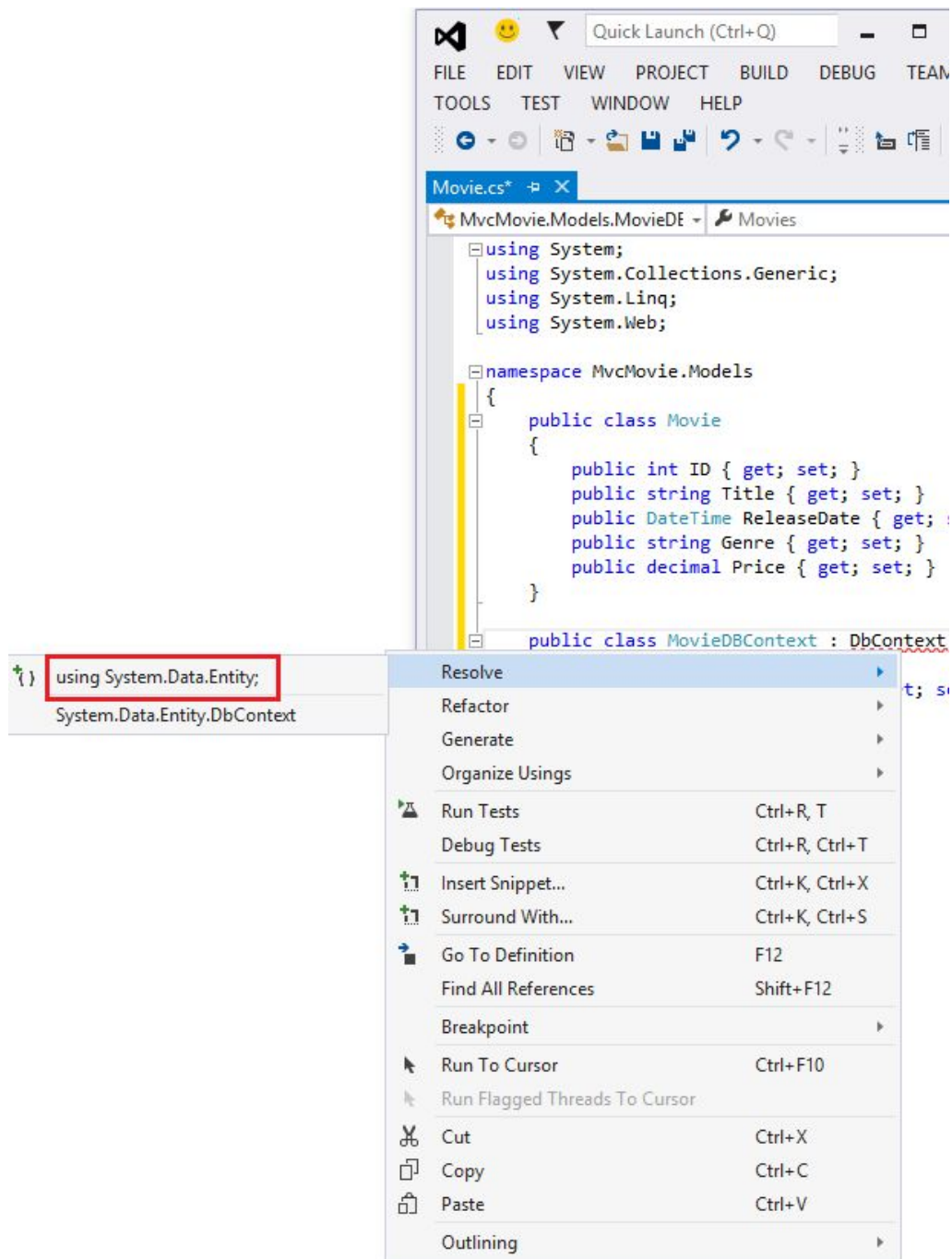
为了能够引用 **DbContext** 和 **DbSet**，你需要添加 **System.Data.Entity** 的引用，代码如下：

```

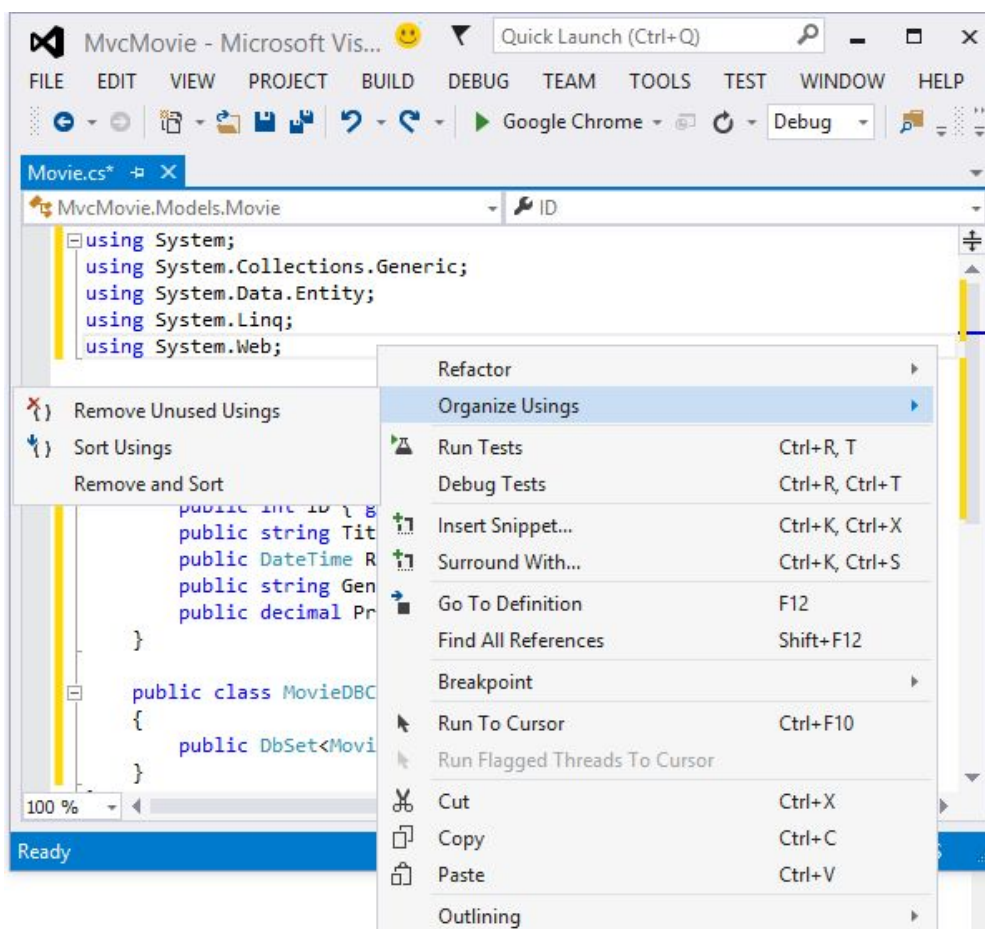
using System.Data.Entity;

```

你可以手动添加 **using** 语句，或者在红色波浪线上面右键，选择“解析”>“using **System.Data.Entity**”。



注意：一些没用到的 `using` 语句已经被移除了。你可以在文件中右键，选择“组织 `using`”>“移除未使用的 `using`”，移除未用到的 `using` 语句。



我们终于添加了一个模型（MVC 中的 M）。在下一节中，我们将讲解使用数据库连接字符串。

5: 使用 SQL Server LocalDB 创建连接字符串

在上一节中，我们创建了 **MovieDbContext** 类来连接数据库、处理 **Movie** 对象和数据库记录的映射。你可能会问我们到底使用了哪个数据库？其实，在我们没有指定数据库的时候，Entity Framework 默认使用 **LocalDB**。

在本节中我们将介绍如何在 **Web.config** 文件中添加一个数据库连接。

SQL Server Express LocalDB

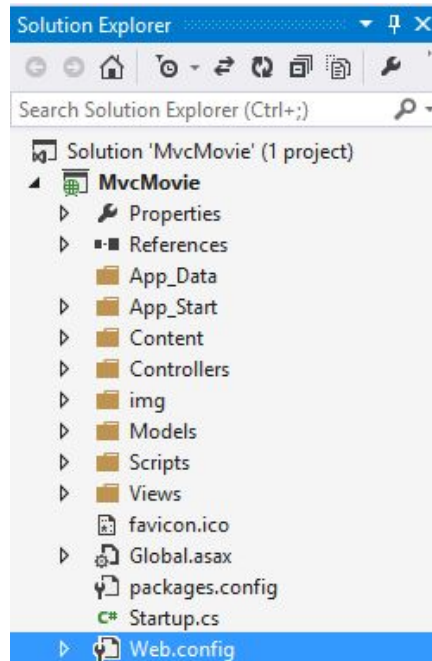
LocalDB 是 SQL Server Express 数据库引擎的轻量级版本，LocalDB 运行在 SQL Server Express 的特殊执行模式下，允许你使用数据库文件（例如 **.mdf** 文件）。通常情况下，LocalDB 数据库文件放在解决方案的 **App_Data** 文件夹下。

SQL Server Express 是不推荐用在 web 应用程序产品中的，其实准确的说，LocalDB 也不能用在 web 应用程序产品中，因为它设计的时候就没考虑和 IIS 一起使用。然而，一个 LocalDB 数据库可以很简单的迁移到 SQL Server 或 SQL Azure。

在 Visual Studio 2013（或 2012）中，Visual Studio 默认安装了 LocalDB。

默认情况下，Entity Framework 会查找和对象上下文类同名的数据库连接（在这个项目中是 MovieDbContext）。

打开应用程序根目录的 Web.config 文件（不是在 Views 目录中的 Web.config）。文件位置如下图：



打开文件后，找到 *connectionString* 节点：

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=301880
-->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile" />
  </configSections>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;
      providerName=System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="PreserveLoginUrl" value="true" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
</system.web>
```

在 `connectionString` 节点下面添加如下连接字符串：

```
<add name="MovieDBContext"
      connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated
Security=True"
      providerName="System.Data.SqlClient"
/>
```

下面是添加了 `MovieDBContext` 连接字符串后完整的 `connectionString` 节点代码：

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-MvcMovie-2013062912
1520.mdf;Initial Catalog=aspnet-MvcMovie-20130629121520;Integrated Security=True"
      providerName="System.Data.SqlClient" />
  <add name="MovieDBContext" connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated
Security=True" providerName="System.Data.SqlClient" />
</connectionStrings>
```

这两个连接字符串很相似，第一个的名字是 *DefaultConnection*，是 `membership` 用来控制谁访问应用程序的数据库。我们添加的连接字符串指定了 *LocalDB* 数据库名字是 *Movie.mdf*，放在 *App_Data* 文件夹。在我们的教程中没有用到 `membership` 数据库，关于 `membership`、`authentication` 和 `security`，我们会在以后的教程中进行介绍。

数据库连接字符串的名字必须和 *DbContext* 类同名。



```
using System;
using System.Data.Entity;
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; }
    }
    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```



事实上，你不需要添加 `MovieDBContext` 连接字符串，如果你没有明确指定连接字符串，

Entity Framework 会在用户目录创建一个 LocalDB 数据库，名字就是 DbContext 类的全名（在本示例中文件名是 MvcMovie.Models.MovieDbContext）。你可以为数据库指定任何你喜欢的名字，只要它有 *MDF* 后缀就行，例如，我们可以用 *MyFilms.mdf* 来命名。

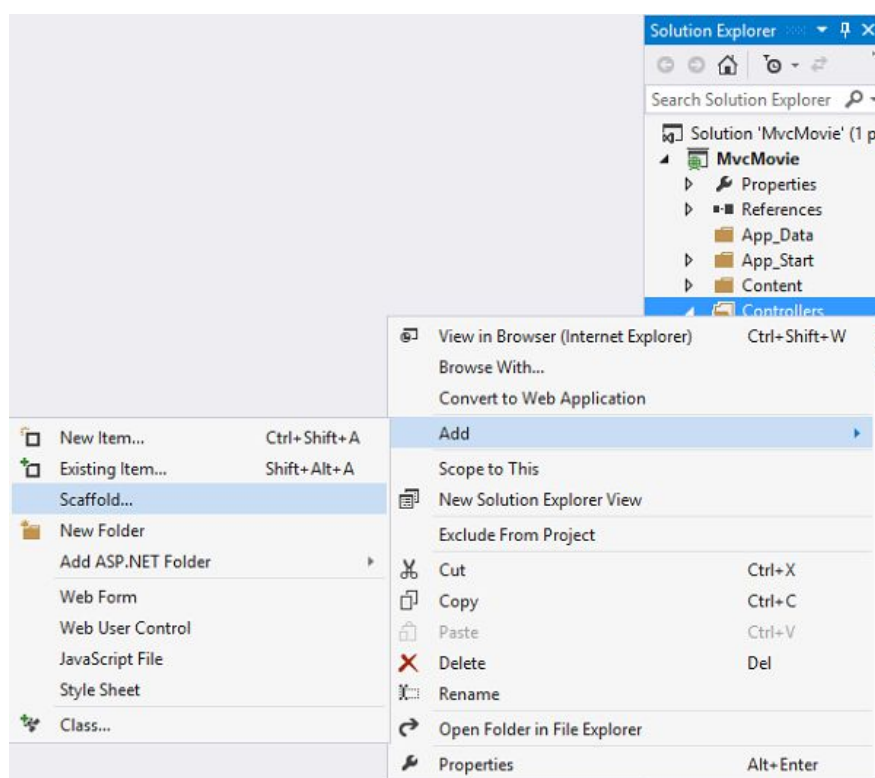
接下来，我们要创建一个新的 **MoviesController** 类，用来显示电影数据和让用户来创建新的电影列表。

6: 通过控制器访问模型的数据

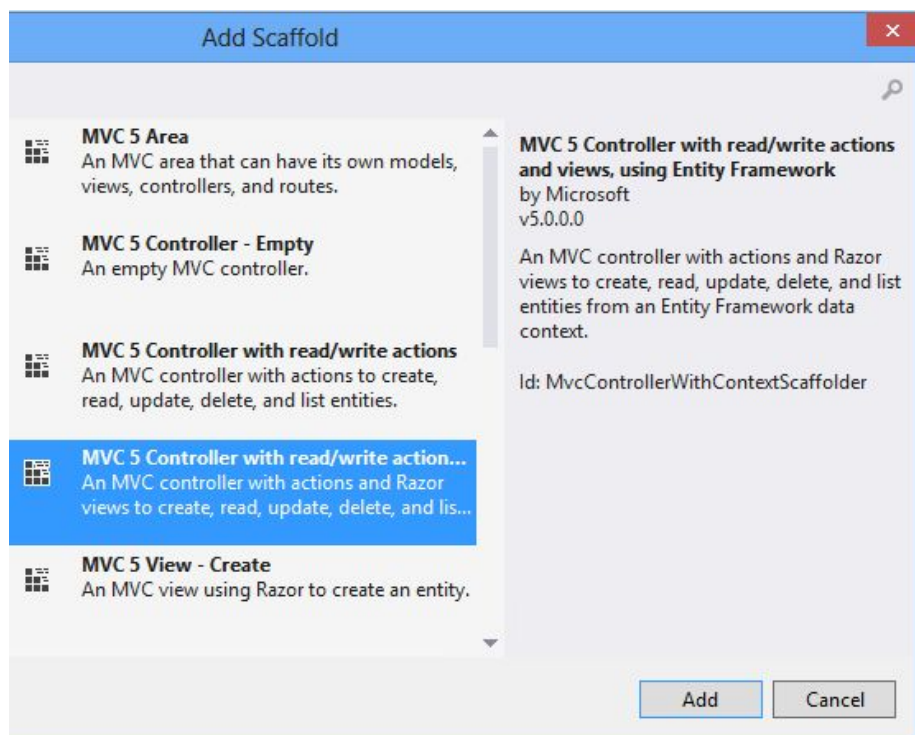
在本节中，你将新建一个 **MoviesController** 类，并编写获取电影数据的代码，使用视图模板将数据展示在浏览器中。

在进行下一步之前，你需要先编译应用程序，否则在添加控制器的时候会出错。

在解决方法资源管理器的 **Controllers** 文件夹右键，选择“添加”>“支架”：

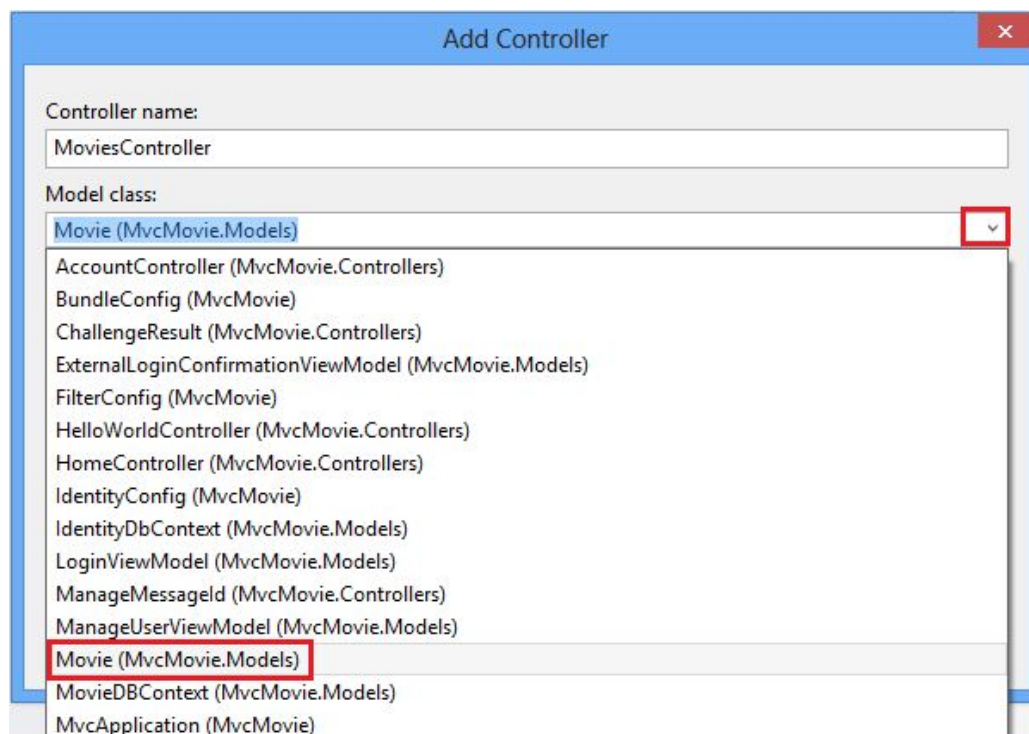


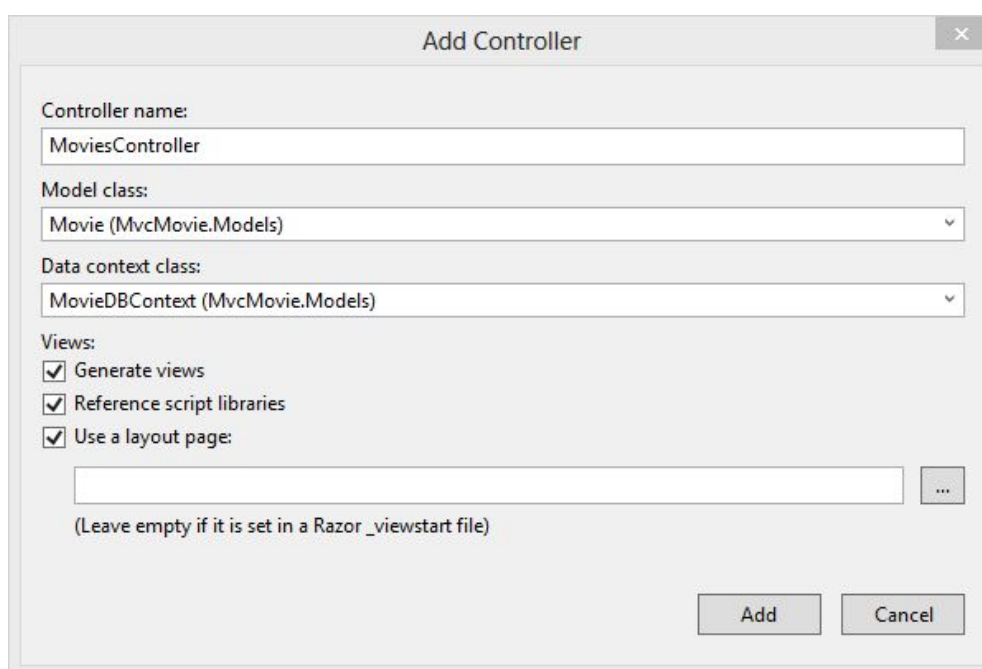
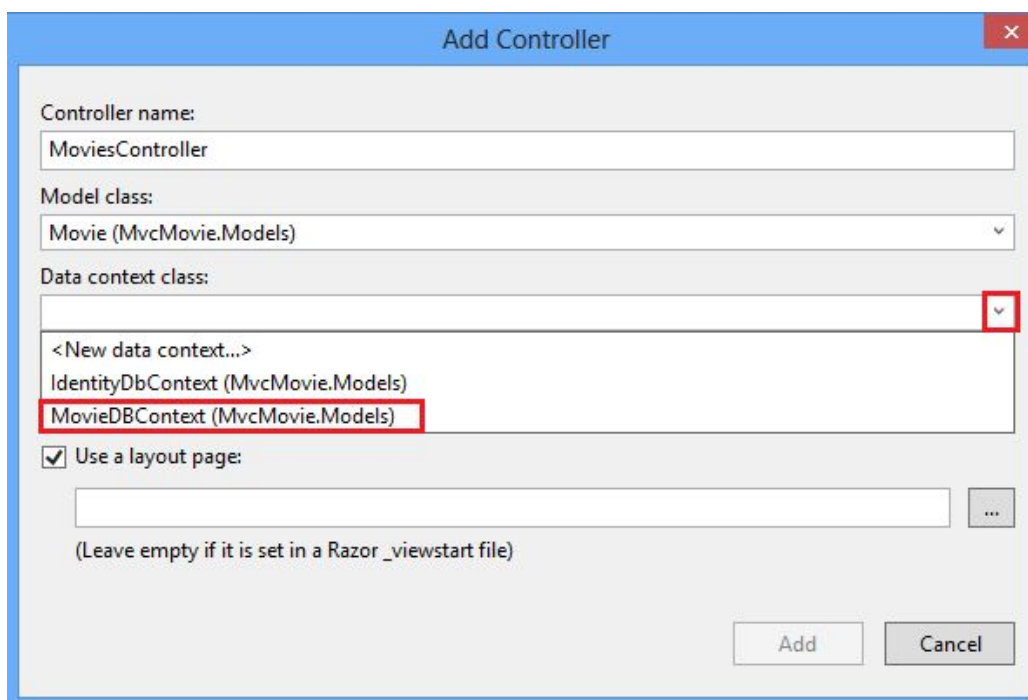
在“添加支架”对话框，选择 *MVC 5 Controller with read/write actions and views, using Entity Framework*，然后单击“添加”按钮。



在 Controller name 输入框中输入 `MoviesController`。在 Model class 选择列表中，选择 `Movie (MvcMovie.Models)`。在 Data context class 选择列表中，选择 `MovieDbContext (MvcMovie.Models)`。

其它几个选项保持默认值，完整的截图如下：





在点击“Add”按钮之后（如果你遇到错误，很有可能是因为你没有编译项目），Visual Studio 会创建如下文件和文件夹：

- 在 *Controllers* 文件夹中创建了 *MoviesController.cs* 文件
- 在 *Views* 文件夹中创建了 *Movies* 文件夹
- 在 *Views\Movies* 文件夹中创建了 *Create.cshtml*, *Delete.cshtml*, *Details.cshtml*, *Edit.cshtml*, 和 *Index.cshtml* 视图文件。

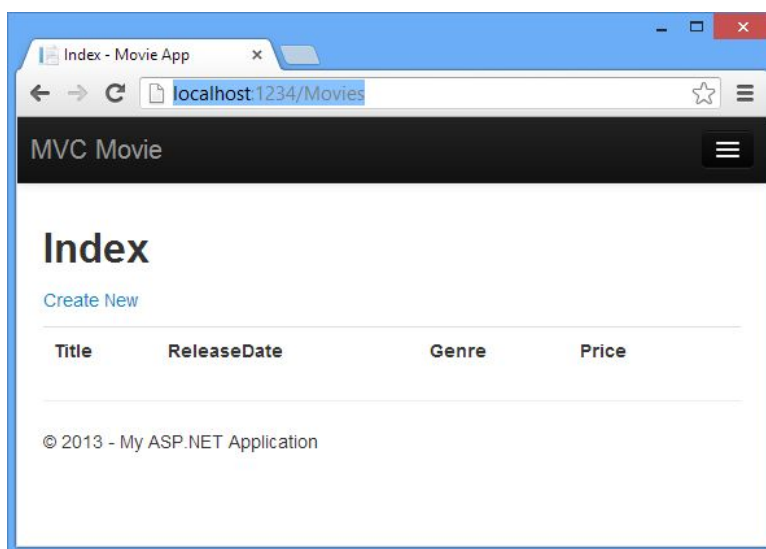
ASP.NET MVC 5 自动创建了 CRUD（create, read, update, delete）操作方法，并为他们创建好了视图。

你现在已经有了一个拥有完整功能的应用程序，你可以使用它来创建、列表显示、编辑和

删除电影了。

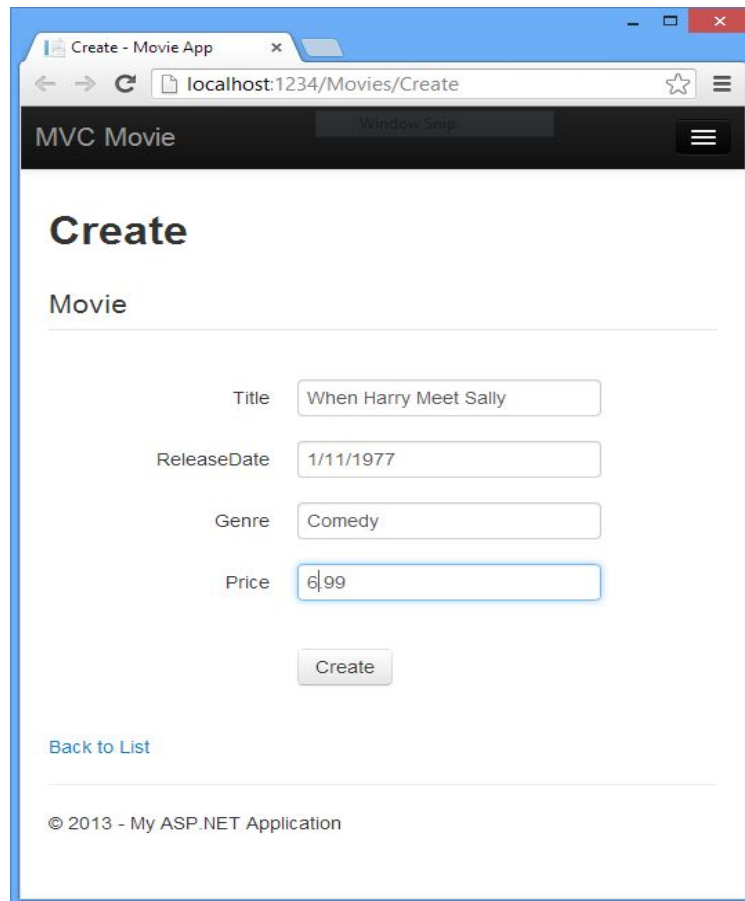
运行应用程序，在浏览器中查看地址 `http://localhost:xxxxx/Movies`。因为程序依赖默认的路由，所以浏览器请求的地址分配给 `MoviesController` 的 `Index` 方法。

换句话说，浏览器请求 `http://localhost:xxxxx/Movies` 地址等同于请求 `http://localhost:xxxxx/Movies/Index` 地址。浏览器中显示一个空的电影列表，因为我们目前还没有添加。



添加电影

选择 “*Create New*” 链接，在打开的页面中输入一些电影信息，然后点击 “*Create*” 按钮：



点击“*Create*”按钮会将数据提交到服务器，服务器将电影信息存入数据库。再次查看 */Movies* 地址，在列表中就能看到我们新添加的电影了。

创建更多的电影记录，然后试试编辑、详细信息和删除功能。

生成的代码详解

打开文件 *Controllers\MoviesController*，检查生成的 *Index* 方法。*MoviesController* 的包含 *Index* 方法的部分代码如下：

```
public class MoviesController : Controller
{
    private MovieDbContext db = new MovieDbContext();
    //
    public ActionResult Index()
    {
        return View(db.Movies.ToList());
    }
}
```

在 *MovieController* 类中，包含了 *MovieDbContext* 类的实例，你可以使用它进行查询、编辑和删除电影。

MoviesController 的 *Index* 方法将返回所有数据库中的电影数据，然后将结果传递给 *Index* 视图。

强类型模型和@model 关键字

在之前的章节中，我们已经介绍了如果使用 *ViewBag* 将数据从控制器传递给视图。*ViewBag* 是一个动态对象，提供了一个便捷的后期绑定的方式将数据传递给视图。

ASP.NET MVC 同样提供了传递强类型数据或对象到视图的功能。这种强类型的方式提供了更好的编译时检查和更丰富的智能感知，Visual Studio 中的脚手架机制在创建 *MoviesController* 类和视图的时候使用了这种方式。

检查 *Controllers\MoviesController.cs* 文件中的 *Details* 方法，下面是 *Details* 方法的部分代码：

```
public ActionResult Details(Int32 id)
{
    Movie movie = db.Movie.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

参数 *id* 通常会作为路由数据传递给控制器，例如：

http://localhost:1234/movies/details/1

将会设置控制器为 *MoviesController*，操作为 *Details*，参数 *id* 为1。你还可以通过查询字符串传递 *id*：

http://localhost:1234/movies/details?id=1

如果找到了电影信息，一个电影的模型就会传递给 *Details* 视图。检查一下 *Views\Movies\Details.cshtml* 文件的内容。

通过文件顶部的 *@model* 语句，你就知道这个视图所期望的什么类型的对象。当你创建了电影控制器，Visual Studio 将会自动在 *Details.cshtml* 文件顶部包含如下语句：

```
@model MvcMovie.Models.Movie
```

@model 指令允许使用强类型 *Model* 对象访问从 *Controller* 传递给 *View* 的电影对象（注意，此时的 *Model* 对象是 *Movie* 类型）。例如，在 *Details.cshtml* 模板中，代码将 *Movie* 的每一个字段通过强类型的 *Model* 对象传递给 *DisplayNameFor* 和 *DisplayFor* HTML 帮助方法。*Create* 和 *Edit* 方法也传递了一个 *Model* 给视图，此处不再多讲。

检查 *Index.cshtml* 模板和 *MoviesController.cs* 文件的 *Index* 方法，程序先创建好一个电影模型的列表对象，然后将创建好的列表对象通过 *View* 方法传递给视图：

```
public ActionResult Index()
{
    return View(db.Movie.ToList());
}
```

Visual Studio 自动在 *Index.cshtml* 文件顶部添加了 `@model` 语句：

```
@model IEnumerable<MvcMovie.Models.Movie>
```

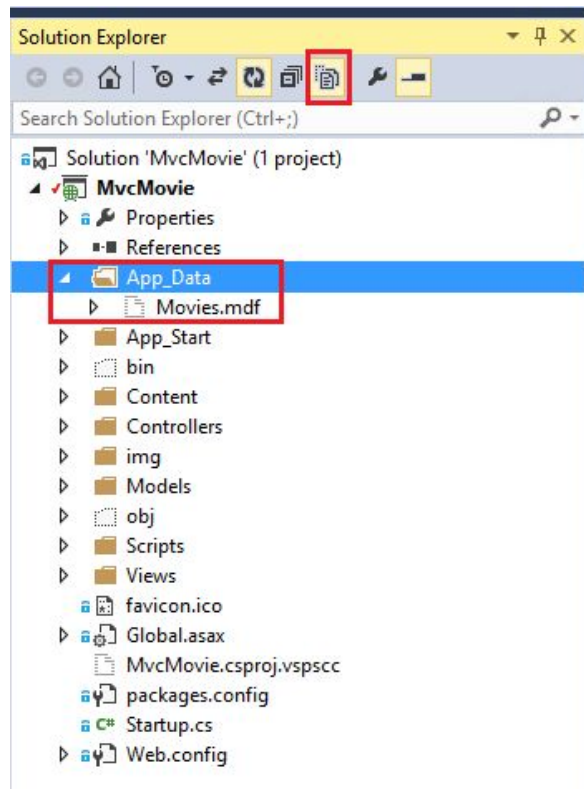
`@model` 指令允许使用强类型 *Model* 对象访问从 *Controller* 传递给 *View* 的电影列表对象（注意，此时的 *Model* 对象是 *IEnumerable<Movie>* 类型）。例如，在 *Index.cshtml* 模板中，代码通过 *foreach* 语句循环强类型 *Model* 中的每个电影对象。

```
@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>                @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
                            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
                            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

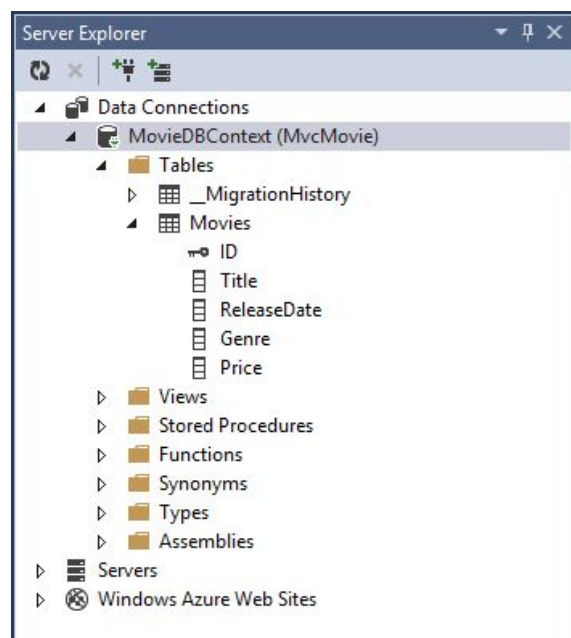
因为 *Model* 对象是强类型 (*IEnumerable<Movie>*)，每个循环中的 *item* 对象都是 *Movie* 类型的。这意味着你的代码具有更好的编译时检查和完整的智能感知支持：

使用 SQL 服务器 LocalDB

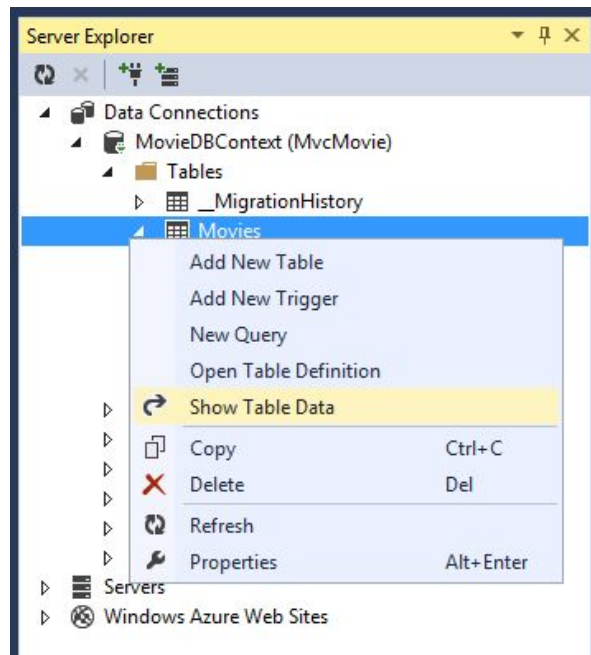
Entity Framework Code First 检查连接字符串使用的数据库是否存在，如果不存在，则会自动创建数据库文件。你可以在 *App_Data* 文件夹下查看数据库是否被创建（如果你没有看到 *Movies.mdf* 文件，点击解决方案资源管理器工具栏上的“显示所有文件”按钮，单击“刷新”按钮，然后展开 *App_Data* 文件夹）。



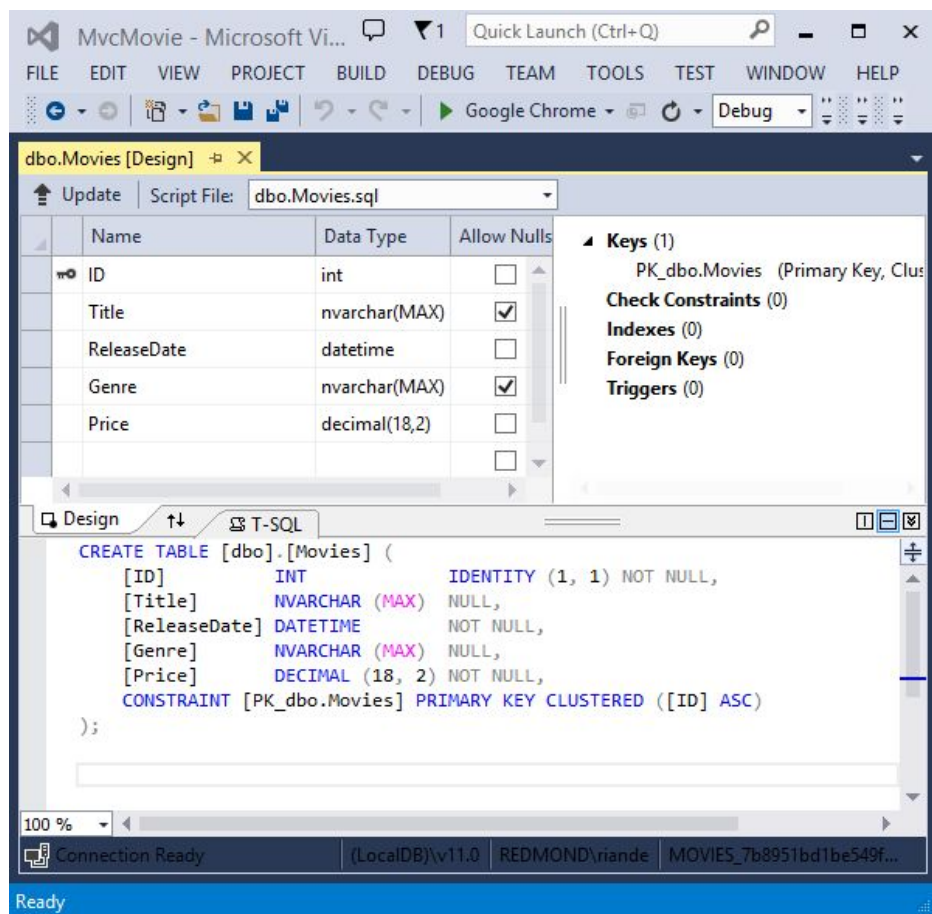
双击 *Movies.mdf* 文件，打开服务器资源管理器，然后展开 *Tables* 目录查看 *Movies* 数据表。



右键 *Movies* 表，选择“显示表数据”查看我们创建的电影数据。

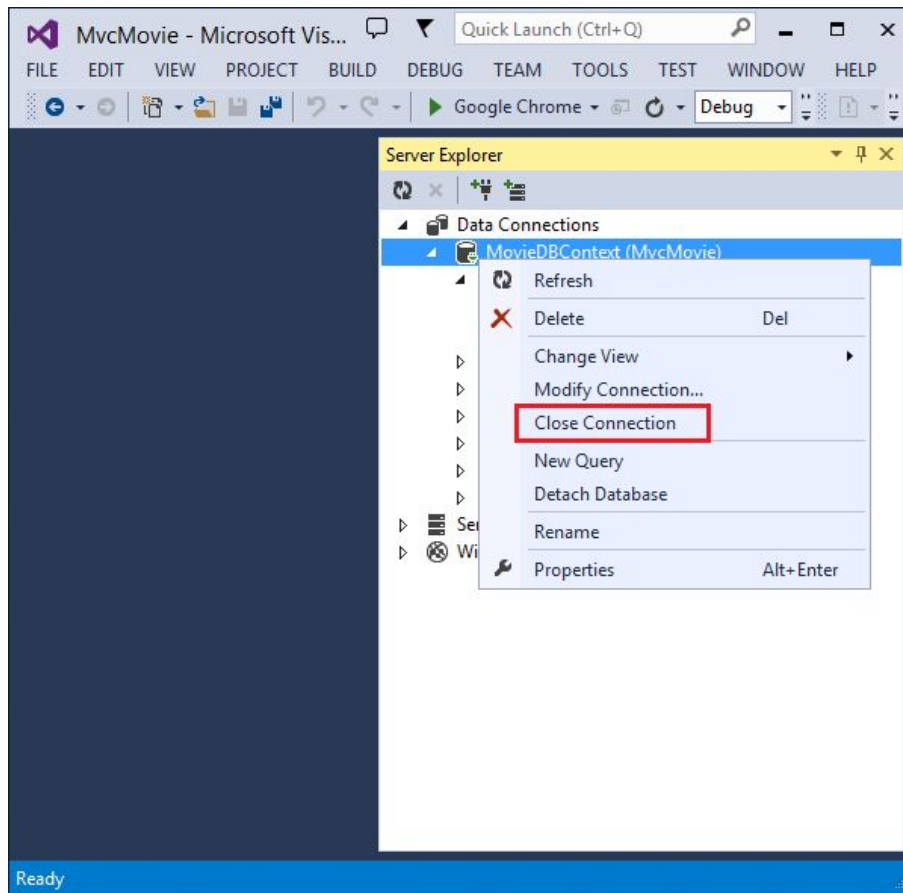


右键 **Movies** 表，选择“打开表定义”，查看 Entity Framework Code First 为我们创建的表结构。



Entity Framework Code First 根据我们 **Movie** 类自动创建了表结构，包括字段名和数据类型。

当我们结束查看或编辑数据库之后，要在服务器资源管理器中关闭连接，右键 **MovieDbContext**，选择“关闭连接”。（如果你不关闭连接，下一次运行程序的时候可能会出现错误）



现在我们已经有了数据库和简单的几个页面来显示数据，在接下来的章节中，我们将研究其它支架生成的代码，并添加一个 **SearchIndex** 方法和 **SearchIndex** 视图，用来查找数据库中的电影。

7: Edit 方法和 Edit 视图详解

在本节中，我们继续研究生成的 **Edit** 方法和视图。但在研究之前，我们先将 **release date** 弄得好看一点。打开 **Models|Movie.cs** 文件，添加下面黄色背景的行：

```
using System;  
using System.ComponentModel.DataAnnotations;  
using System.Data.Entity;
```

```

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; }
    }

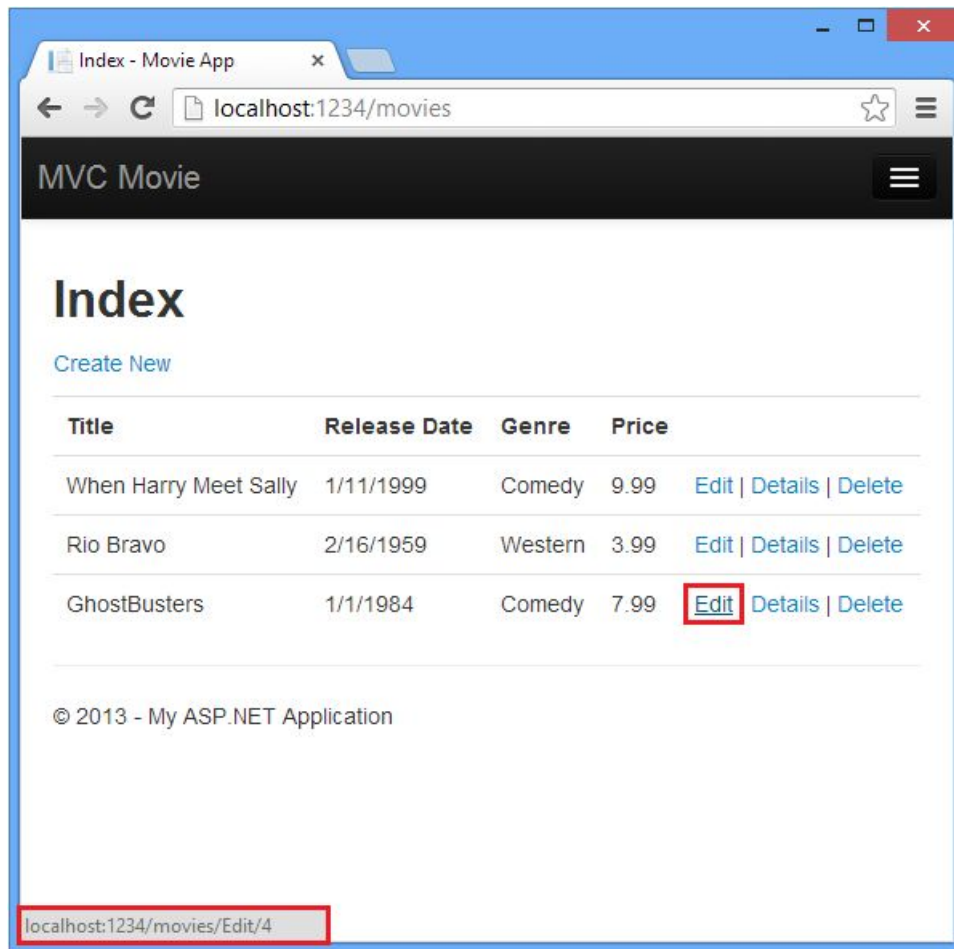
    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}

```



我们将在下一节中介绍 **DataAnnotations**。**Display** 特性指定了显示的字段名（本例中“**Release Date**”替换了“**ReleaseDate**”）。**DataType** 特性指定了数据类型，在本例中它是日期类型，因此存储在该字段的时间信息将不会显示出来。

运行应用程序，在浏览器地址栏中追加 */movies* 来访问 *Movies* 控制器。将鼠标放在 **Edit** 链接上面，查看它链接到的地址：



Edit 链接是通过 `Html.ActionLink` 方法生成的，在 `Views\Movies\Index.cshtml` 视图中，代码如下：

```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
```

```
<td>
@Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) |
@Html. (extension) MvcHtmlString HtmlHelper.ActionLink(string linkText, string actionName, object routeValues)
@Html. Returns an anchor element (a element) that contains the virtual path of the specified action.

Exceptions:
System.ArgumentException
```

`Html` 对象是 `System.Web.Mvc.WebViewPage` 基类中的一个属性，`ActionLink` 方法使动态生成 控制器中方法的 HTML 超链接更为简单。`ActionLink` 方法的第一个参数是要显示的链接文字（例如 `<a>Edit Me`）；第二个参数是要调用的控制器方法（在这里是 `Edit` 方法）；最后一个参数是生成路由数据用的匿名对象（在这里是 `id`，值为1）。

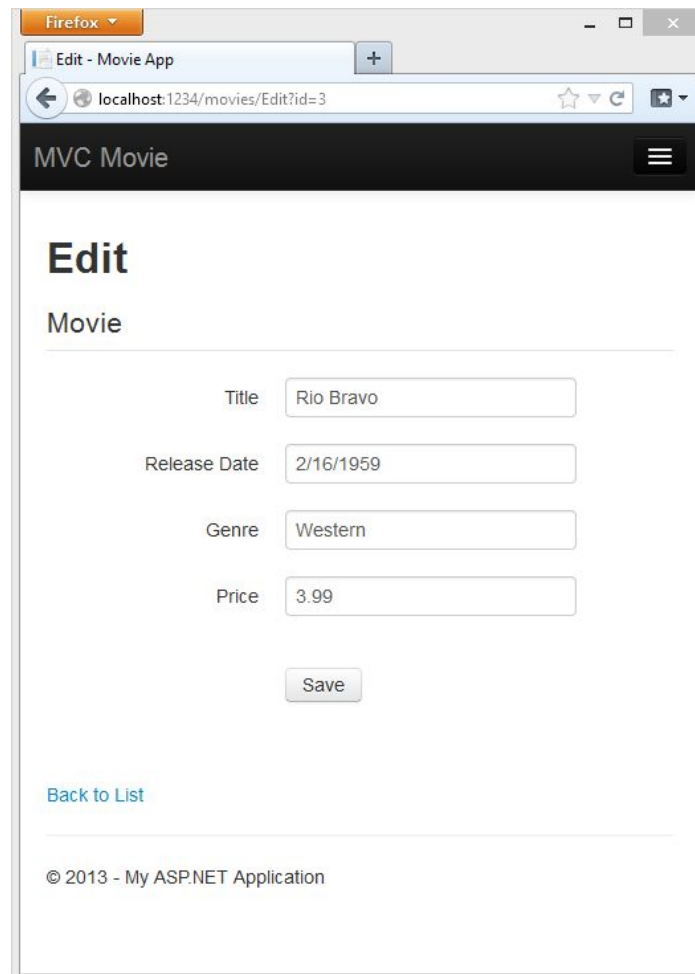
在图1中显示的生成地址是 `http://localhost:2264/Movies/Edit/1`。默认的路由匹配 `{controller}/{action}/{id}` URL 模式。因此，ASP.NET 将请求地址 `http://localhost:2264/Movies/Edit/1` 翻译为 `MoviesController` 的 `Edit` 方法，参数 `id` 为1。查看 `App_Start\RouteConfig.cs` 文件中的代码：

```

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);

```

你还可以使用查询字符串传递参数，例如 *http://localhost:xxxxx/Movies/Edit?ID=1* 也可以将参数 **id** 为1的值传递给 **Movies** 控制器。



打开 **Movies** 控制器，它包含了两个 **Edit** 方法，代码如下：

```

// GET: /Movies/Edit/5
public ActionResult Edit(Int32 id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

```

```

}
//
// POST: /Movies/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

注意第二个 `Edit` 方法，它被 `HttpPost` 特性修饰（代码中黄色背景标注），这个特性指定只有 POST 请求才能调用这个重载的 `Edit` 方法。我们可以为第一个 `Edit` 方法添加 `HttpGet` 特性，但这不是必须的，因为它是默认值（我们将未标记的方法认定为 `HttpGet` 方法）。第二个 `Edit` 方法还有一个 `ValidateAntiForgeryToken` 特性，这个特性用来阻止伪造的请求，它和视图（`Views\Movies\Edit.cshtml`）中的 `@Html.AntiForgeryToken()` 是成对出现的。

```

@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)
    <fieldset class="form-horizontal">
        <legend>Movie</legend>
        @Html.HiddenFor(model => model.ID)
        <div class="control-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Title)
            </div>
            @Html.ValidationMessageFor(model => model.Title, null, new { @class = "help-inline" })
        </div>
    </div>
}

```

`@Html.AntiForgeryToken()` 生成一个表单防伪标记，必须与 `MoviesController` 中的 `Edit` 方法匹配。

HttpGet 标记的 Edit 方法接收一个 ID 的参数, 通过 Entity Framework Find 方法查找电影, 并将找到的结果返回个 Edit 视图。如果调用 Edit 方法的时候没有参数, ID 参数的值将会是默认的0。如果未找到电影信息, 控制器将返回一个 HttpNotFound。当支架系统创建 Edit 视图的时候, 它会检查 Movie 类, 为它的每一个属性创建绘制<label>和<input> 元素的代码。下面的示例展示了 Visual Studio 支架系统创建的 Edit 视图代码:

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)
    <fieldset class="form-horizontal">
        <legend>Movie</legend>
        @Html.HiddenFor(model => model.ID)
        <div class="control-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title, null, new { @class = "help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.ReleaseDate, new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.ReleaseDate)
                @Html.ValidationMessageFor(model => model.ReleaseDate, null, new { @class = "help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.Genre, new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Genre)
                @Html.ValidationMessageFor(model => model.Genre, null, new { @class = "help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.Price, new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Price)
```

```

@Html.ValidationMessageFor(model => model.Price, null, new { @class = "help-inline" })
    </div>
</div>
<div class="form-actions no-color">
    <input type="submit" value="Save" class="btn" />
</div>
</fieldset>}

<div>    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```



在视图文件顶部的语句 `@model MvcMovie.Models.Movie` 指明该视图需要一个 *Movie* 型的模型。

支架生成的代码使用了很多精简 HTML 标记的帮助方法, *Html.LabelFor* 方法用来显示字段名 ("Title", "ReleaseDate", "Genre", 和 "Price"), *Html.EditorFor* 方法生成一个 HTML *<input>* 元素, *Html.ValidationMessageFor* 方法结合属性显示一些验证信息。

运行应用程序, 导航到 */movies* 地址, 点击 *Edit* 链接, 在浏览器中, 查看页面源代码, form 标签的 HTML 代码如下:



```

<form action="/Movies/Edit/1" method="post">
    <input name="__RequestVerificationToken" type="hidden"
value="vM6-yBtYJY5uRgKm3ivFw9gLMjauE9i2-Wi6Y--2Swm4-BlucfrG71zQV3n773tT9i8ytaG
5WSanfBBd54qIhAKAkTJO_Z0UhRaGPDhMJ9M1" />    <fieldset class="form-horizontal">
        <legend>Movie</legend>
        <input data-val="true" data-val-number="The field ID must be a number."
data-val-required="ID 字段是必需的。" id="ID" name="ID" type="hidden" value="1" />
        <div class="control-group">
            <label class="control-label" for="Title">Title</label>
            <div class="controls">
                <input class="text-box single-line" id="Title" name="Title" type="text" value="中国合伙人" />
                <span class="field-validation-valid help-inline" data-valmsg-for="Title"
data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="control-group">
            <label class="control-label" for="ReleaseDate">Release Date</label>
            <div class="controls">
                <input class="text-box single-line" data-val="true" data-val-date="The

```



```

field Release Date must be a date." data-val-required="Release Date 字段是必需的。"
id="ReleaseDate" name="ReleaseDate" type="date" value="2013/6/18" />
    <span class="field-validation-valid help-inline" data-valmsg-for="ReleaseDate"
data-valmsg-replace="true"></span>
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="Genre">Genre</label>
    <div class="controls">
        <input class="text-box single-line" id="Genre" name="Genre" type="text" value="励志"
/>
        <span class="field-validation-valid help-inline" data-valmsg-for="Genre"
data-valmsg-replace="true"></span>
    </div>
</div>
<div class="control-group">
    <label class="control-label" for="Price">Price</label>
    <div class="controls">
        <input class="text-box single-line" data-val="true" data-val-number="The
field Price must be a number." data-val-required="Price 字段是必需的。" id="Price"
name="Price" type="text" value="70.00" />
        <span class="field-validation-valid help-inline" data-valmsg-for="Price"
data-valmsg-replace="true"></span>
    </div>
</div>
<div class="form-actions no-color">
    <input type="submit" value="Save" class="btn" />
</div>
</fieldset>
</form>

```



看一下这个 **form** 标签，它的 **action** 属性是 `/movies/edit/1`，说明 **form** 将提交到这个地址，而它里面包含的 `<input>` 标签则对应每一个 **Movie** 类的字段。当点击“Save”按钮时，**form** 中的数据将提交到服务器。第二行代码中，我使用高亮提示，这行代码是使用 `@Html.AntiForgeryToken()` 生成的隐藏域，用来阻止伪造的请求。

处理 **POST** 请求

下面是接收 **POST** 请求的 **Edit** 方法：



```

[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Movie movie)
{

```

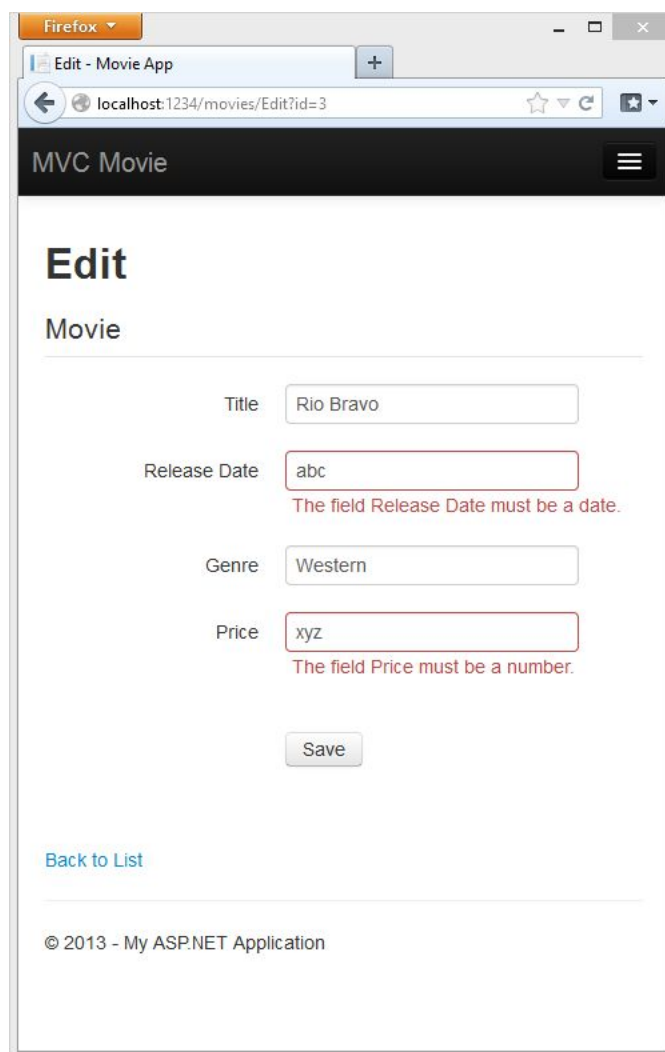
```
if (ModelState.IsValid)
{
    db.Entry(movie).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
}
return View(movie);
}
```



ASP.NET MVC model binder 使用提交的表单数据创建一个 *Movie* 对象，作为 *movie* 参数传递给 *Edit* 方法。*ModelState.IsValid* 验证 *form* 提交的数据是否能够用来修改（编辑或更新）*Movie* 对象，如果验证通过，电影数据就会保存在 *db.Movies* 列表中（*db* 是 *MovieDbContext* 的实例），通过调用 *db.SaveChanges* 方法将修改后的电影数据保存在数据库中。在数据保存之后，*RedirectToAction* 将跳转到 *Index* 方法，用来显示电影列表，此时就能看到更新后的电影信息。

在用户输入过程中，一旦输入了未通过验证的值，客户端就会显示一个错误信息，这些功能是通过 *Javascript* 完成的，如果你禁用了 *Javascript*，客户端验证功能将会失效，但服务器会检查到不合法的数据，然后将包含错误信息的表单重新显示在客户端。在后面的章节中我们将介绍更多验证的细节。

Edit.cshtml 视图中的 *Html.ValidationMessageFor* 方法会显示相应的错误信息。如下图：



所有处理 **Get** 请求的方法都有相似的模式：他们获取一个电影对象（在 **Index** 中是电影列表），然后传递给 **View**。 **Create** 方法传递一个空的电影数据给视图。使用 **HTTP GET** 方法修改数据存在安全风险，也同样违背了 **HTTP** 最佳实践，并且 **REST** 模式中明确指出，**GET** 请求不应该改变应用程序的状态，换句话说，**GET** 操作应该是一个安全的操作，没有副作用，也不会修改你的持久化数据。

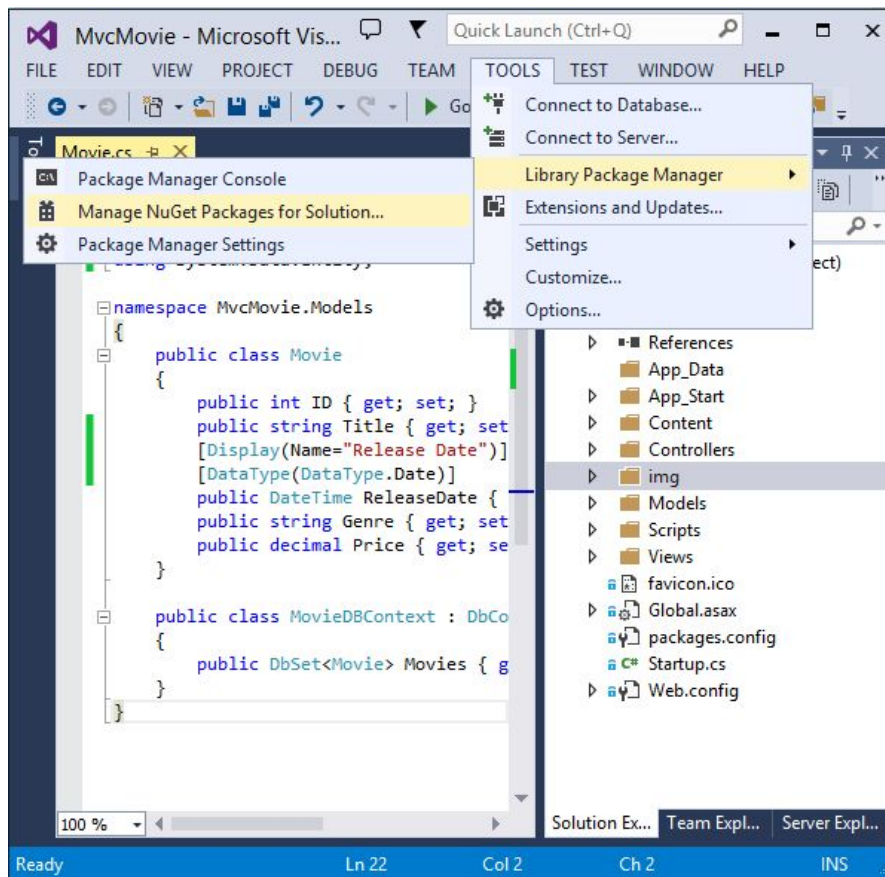
本地化验证

如果你是用英语，这部分可以跳过。作为汉语国家用户，我还是把这部分翻译一下吧。

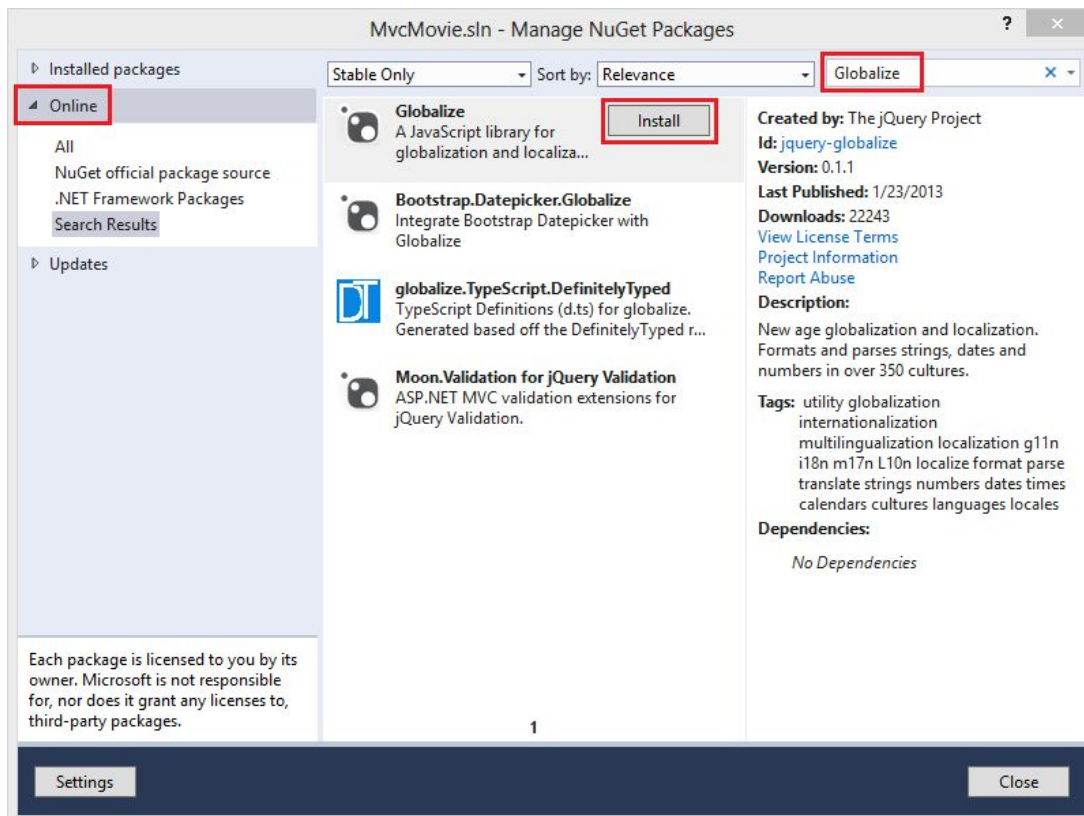
对于非英语地区的用户来说，小数点使用逗号 (",")，还有非美语的日期格式，为了支持 **Jquery** 验证，你必须包含 `globalize.js` 和特定的 `cultures/globalize.cultures.js` 文件，在 **Javascript** 中使用 `Globalize.parseFloat`。你可以通过 **NuGet** 获得 `jQuery non-English validation`（英语地区不要安装 `Globalize`）。

From the **Tools** menu click **Library Package Manager**, and then click **Manage NuGet Packages for Solution**.

在菜单栏中单击“工具”>“库程序包管理器”>“管理解决方案的 **NuGet** 程序包”。



在左侧单击“联机”，在查询框中输入“Globalize”：



点击“安装”按钮，*Scripts\jquery.globalize\globalize.js* 文件将会添加到项目中，文件夹 *Scripts\jquery.globalize\cultures* 中包含了很多 Javascript 文件。

下面的代码显示了将文件 *Views\Movies\Edit.cshtml* 使用汉语区域所作的修改（原文作者使用法语作为演示）：



```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
    <script src="~/Scripts/jquery.globalize/globalize.js"></script>
    <script src="~/Scripts/jquery.globalize/cultures/globalize.culture.fr-FR.js"></script>
    <script>
        $.validator.methods.number = function (value, element) {
            return this.optional(element) ||
                !isNaN(Globalize.parseFloat(value));
        }
        $(document).ready(function () {
            Globalize.culture('fr-FR');
        });
    </script>
    <script>
        jQuery.extend(jQuery.validator.methods, {
            range: function (value, element, param) {
                //Use the Globalization plugin to parse the value
                var val = $.global.parseFloat(value);
                return this.optional(element) || (
                    val >= param[0] && val <= param[1]);
            }
        });
    </script>
    <script>
        $.validator.methods.date = function (value, element) {
            return this.optional(element) ||
                !isDate(Globalize.parseDate(value));
        }
    </script>}
```



为了避免在每个编辑视图都写上这么一大段代码，你可以将他们写在布局页。

在下一节中我们将为程序实现查找功能。



8: 搜索查询

添加一个搜索的方法和搜索的视图

在本节中，我们为 **Index** 方法添加查询功能，使我们能够根据电影的题材或名称进行查找。

修改 **Index** 表单

首先，我们需要更新 **MoviesController** 的 **Index** 方法，代码如下：

```
  
public ActionResult Index(string searchString)  
{  
    var movies = from m in db.Movies  
                  select m;  
  
    if (!String.IsNullOrEmpty(searchString))  
    {  
        movies = movies.Where(s => s.Title.Contains(searchString));  
    }  
  
    return View(movies);  
}  

```

Index 方法的第一行代码创建了一个 **LINQ** 查询，用来选择符合条件的电影：

```
var movies = from m in db.Movies select m;
```

这个查询虽然在这里定义出来，但并没有在数据库中执行。

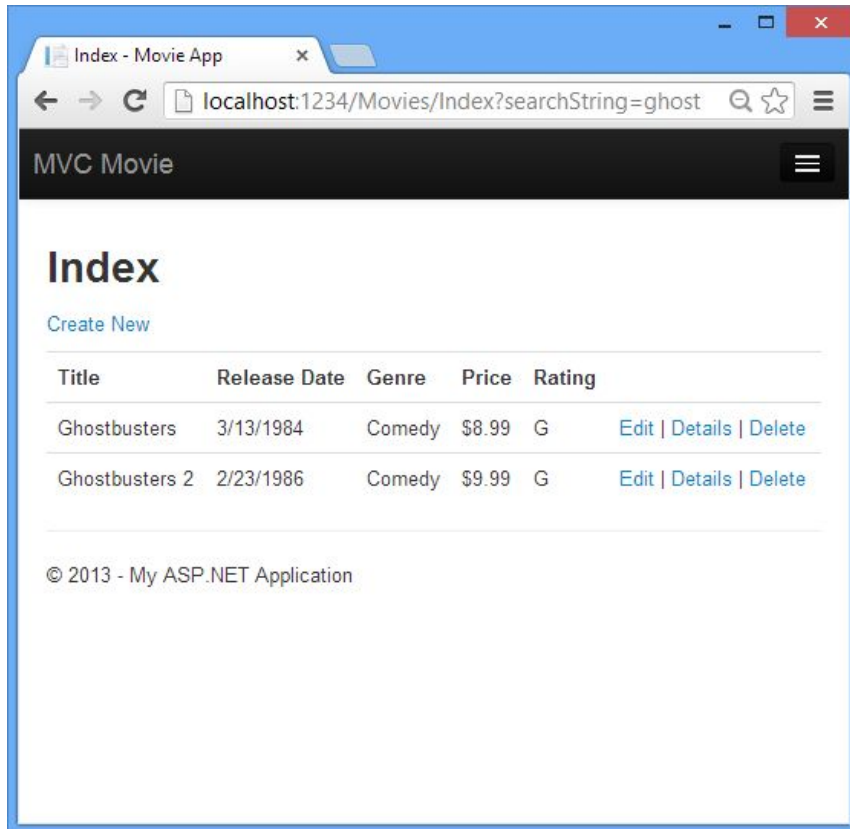
如果 **searchString** 参数包含一个字符串（不是空字符串），**movies** 查询将会添加一个查询字符串的过滤条件，代码如下：

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

代码清单3中的 **s => s.Title** 是一个 **Lambda** 表达式，**Lambda** 表达式被用在基于方法的 **LINQ** 查询中（上面代码中的 **Where** 方法），当做参数来使用。**LINQ** 语句在定义或修改的时候不会执行，相反的，查询会延迟执行，这意味着一个赋值语句直到迭代完成或调用 **ToList** 方法才具备真正的值。在上面的示例中，查询语句在 **Index.cshtml** 视图中执行。

现在，你可以修改 **Index** 视图，让他展示一个表单给用户输入。

运行应用程序，并导航到 **/Movies/Index**，在 **URL** 后面添加一个查询，例如 **?searchString=中国**，被过滤的电影内容如下：



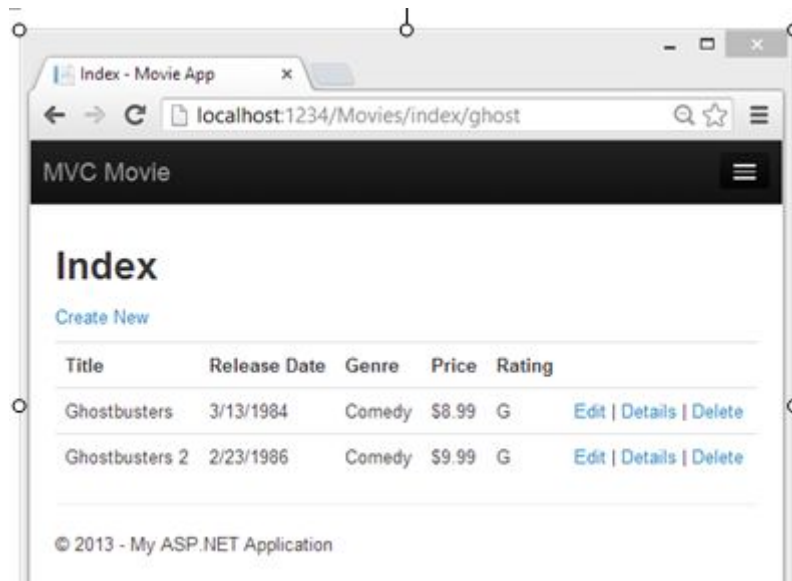
如果你把 `Index` 方法的参数名改为 `id`，那么 `id` 参数将会匹配 `App_Start\RouteConfig.cs` 文件中的默认路由中的 `{id}`。

`{controller}/{action}/{id}`

修改后的 `Index` 方法如下：

```
public ActionResult Index(string id)
{
    string searchString = id;
    var movies = from m in db.Movies select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }
    return View(movies);
}
```

修改以后，我们可以通过路由数据来传递查询字符串：



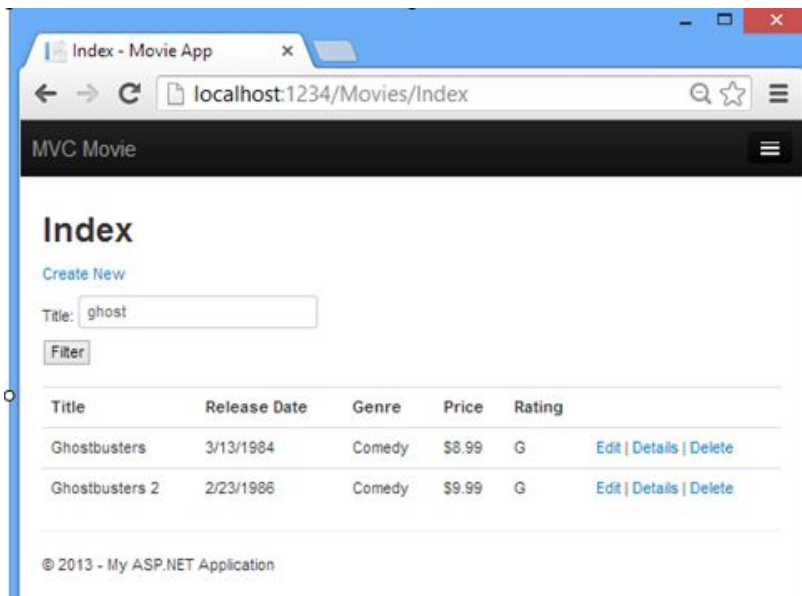
然而，你不能指望用户每次都通过修改 URL 来查找电影，因此你需要在界面上帮助他们过滤数据。如果你刚刚使用代码清单4中的代码测试了路由绑定是如何传递 ID 参数，现在再把它改回来，原始的代码可参考代码清单1。

打开 *Views\Movies\Index.cshtml* 文件，在 `@Html.ActionLink("Create New", "Create")` 后面添加如下代码：

```
<p>    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm())
    {
        <p>        Title: @Html.TextBox("SearchString") <br />
        <input type="submit" value="Filter" />
    }
</p>
```

`Html.BeginForm` 帮助方法创建一个 `<form>` 标签。通过单击“**Filter**”按钮将表单提交给当前的页面。

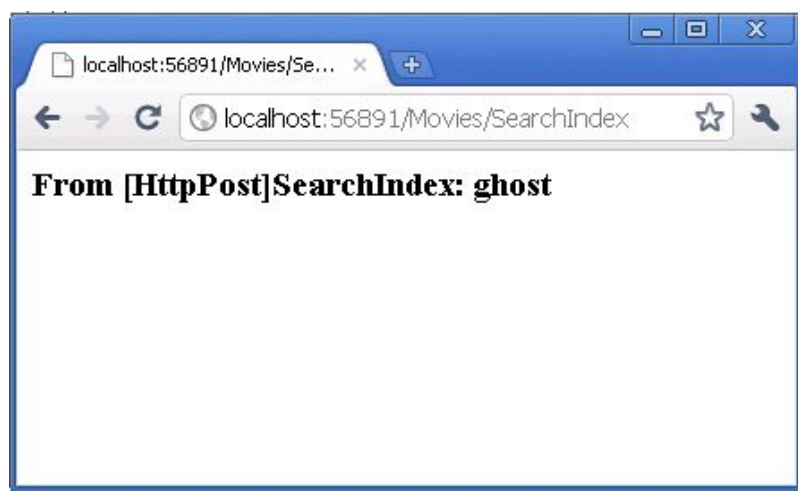
运行应用程序，然后试着查找一个电影：



我们没有为 `Index` 方法定义 `HttpPost` 的重载，因为我们根本没有修改数据，只是做了一个查询。

我们可以为 `Index` 方法添加 `HttpPost` 的重载，这样一来，程序将会调用 `HttpPost` 修饰的 `Index` 方法，相应的代码和截图如下：

```
[HttpPost]
public string Index(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]Index: " + searchString + "</h3>";
}
```



然而，尽管我们添加了 `HttpPost` 版的 `Index` 方法，它在实现的时候仍然存在一些局限性。设想你想将一个比较详细的查询添加书签，或者你想将查询结果以链接形式发送给朋友，注意观察 `HTTP POST` 请求的时候，`URL` 是没有改变的（仍然是 `localhost:xxxxx/Movies/Index`），这个地址本身不包含查询信息。现在，查询信息是作为表单数据发送到服务器的，这意味着你

不能抓取到 URL 中的查询信息，将 URL 作为书签或发送给朋友。

解决方案就是重写 `BeginForm` 语句，使它发送一个 GET 请求，从而调用 `HttpGet` 版本的 `Index` 方法，修改后的代码如下：

```
@using (Html.BeginForm("Index", "Movies", RequestMethod.Get))
```

```
@Html.ActionLink("Create New", "Create")
@using (Html.BeginForm("SearchIndex", "Movies", RequestMethod.Get))
{
    <p>
    <input type="text" value="ghost" />
    <input type="button" value="Filter" />

```

▲ 5 of 13 ▼ (extension) MvcForm HtmlHelper.BeginForm(string actionName, string controllerName, RequestMethod method)

Writes an opening <form> tag to the response. When the user submits the form, the request will be processed by an action method.

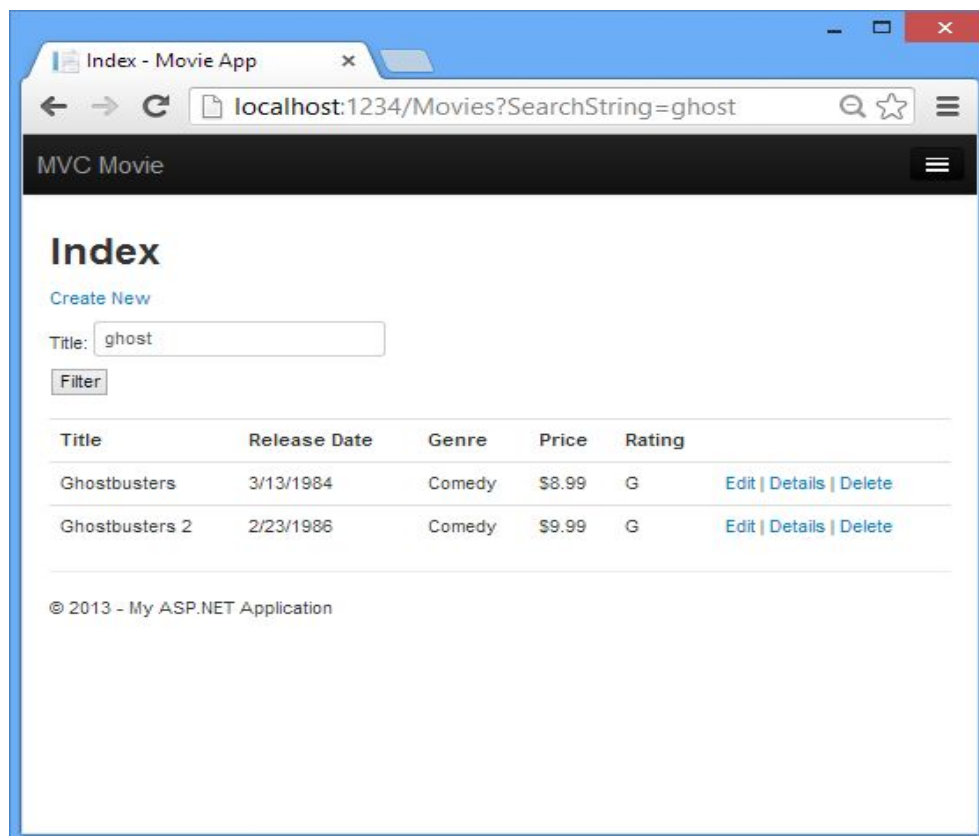
method: The HTTP method for processing the form, either GET or POST.

```

}

```

现在你再点击“Filter”按钮进行查找，查找的参数将包含在 URL 中，然后调用 `HttpGet` 版的 `Index` 方法：



添加题材查询

删掉之前代码中添加的 `HttpPost` 版的 `Index` 方法，以后我们不再用到了。

然后，我们修改 `Index` 方法的代码，使它能够根据题材进行查询。修改后的代码如下：

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

```

```

var GenreQry = from d in db.Movies
                orderby d.Genre
                select d.Genre;
GenreLst.AddRange(GenreQry.Distinct());
ViewBag.movieGenre = new SelectList(GenreLst);
var movies = from m in db.Movies
              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);
}
return View(movies);
}

```



在修改完控制器之后，我们还需要在 **Index** 视图添加一个字段。在我们之前的名字查询之前，使用 **Html.DropDownList** 添加一个下拉框，修改后的代码如下：

```

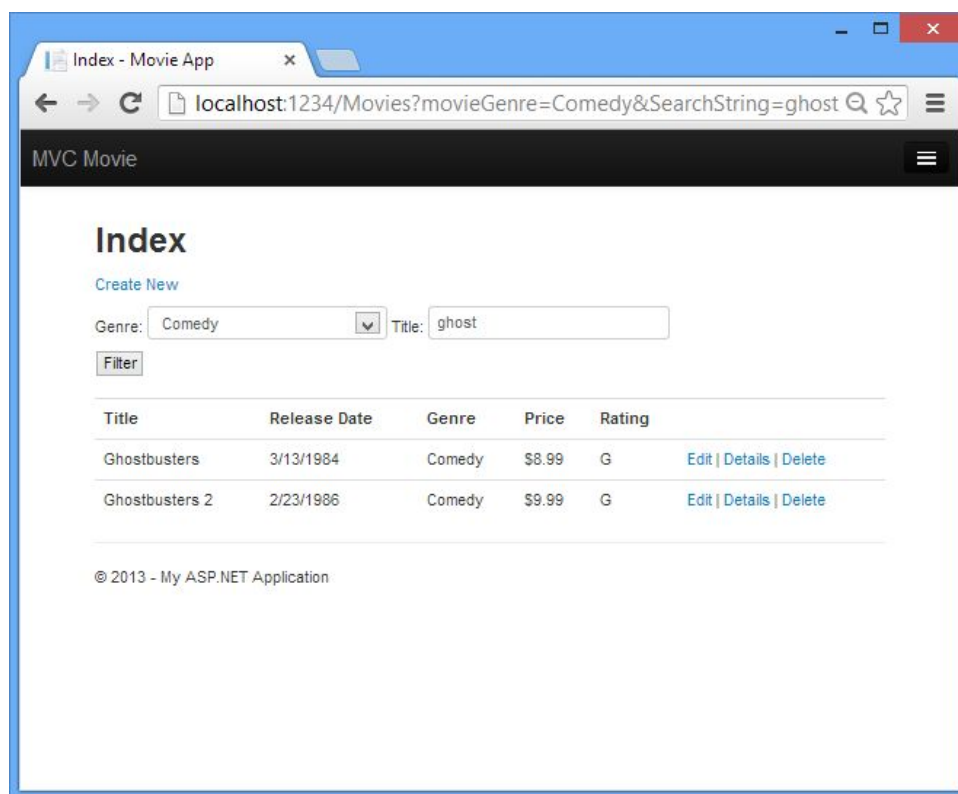
@using (Html.BeginForm("Index", "Movies", FormMethod.Get))
{
    <p>    Genre: @Html.DropDownList("movieGenre", "All")
    Title: @Html.TextBox("SearchString") <br />
    <input type="submit" value="Filter" />
</p>}

```



代码 **@Html.DropDownList("movieGenre", "All")** 生成一个下拉列表，参数 **movieGenre** 指明要查找的 **ViewBag** 中的数据集合的名称，我们在代码8中做了标记；参数 **all** 是预选中的数据，我们可以使用“Comedy”替换。

运行应用程序，试着根据题材和标题查询一下：



在本节中，我们创建了查询的方法和视图，使用户可以根据电影的标题和题材进行查询。在下一节中，我们将结束如何为 **Movie** 模型添加属性，以及如何添加一个自动创建测试数据库的初始值。

9: 添加新字段

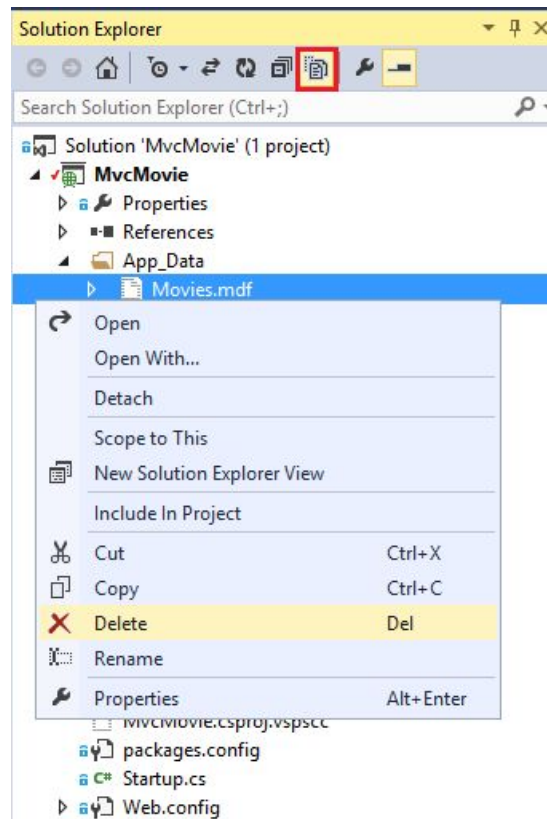
在本节中，我们将使用 **Entity Framework Code First** 数据迁移功能将模型类的改变应用到数据库中。

默认情况下，当我们使用 **Entity Framework Code First** 自动创建一个数据库，像我们之前教程中讲的那样，**Code First** 添加一个 **table** 帮我们跟踪数据库结构是否与模型类同步。如果不同步，**Entity Framework** 将抛出一个错误，这样更方便我们在开发的时候发现问题，否则只能在运行时通过晦涩的错误来查找了。

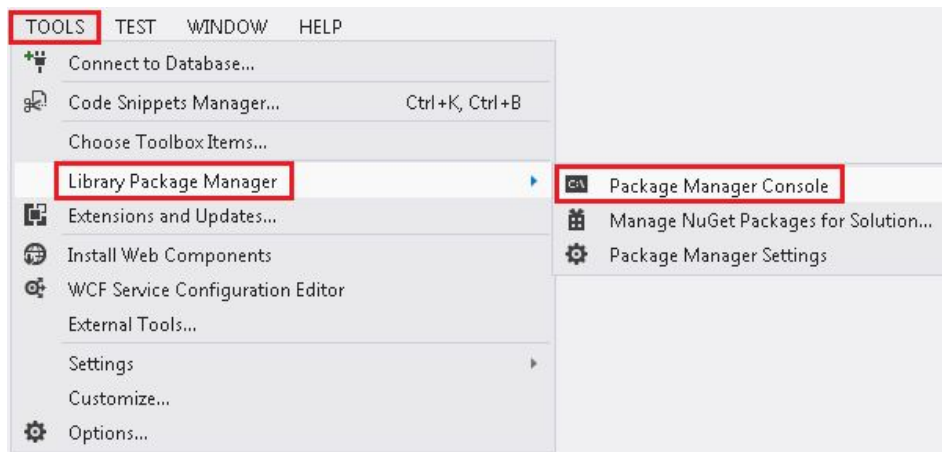
为模型更改设置 **Code First** 数据迁移

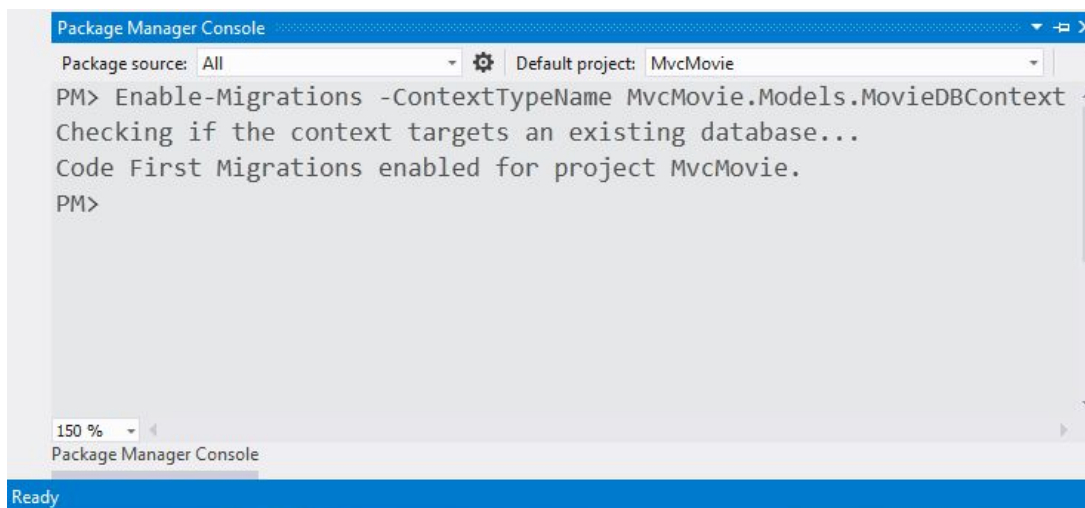
在解决方案资源管理器中，删除自动创建的 **Movies.mdf** 文件。

在工具菜单中，选择“库程序包管理器”>“程序包管理器控制台”：

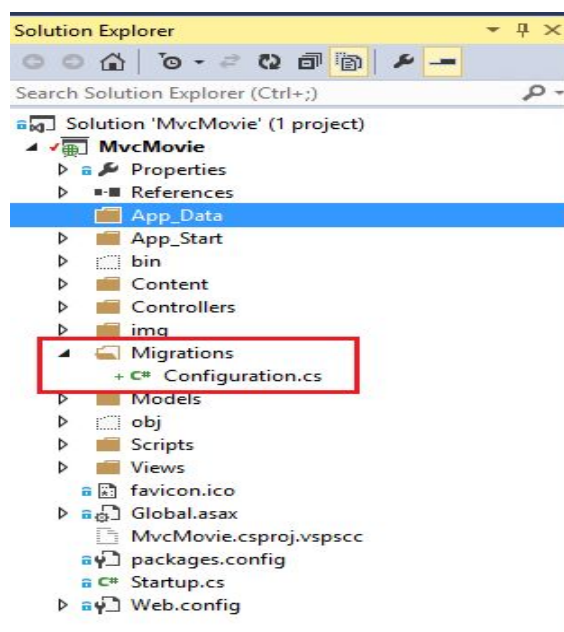


在“程序包管理器控制台”窗口中输入：`Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDbContext`





Enable-Migrations 命令创建了一个 Migrations 文件夹和 Configuration.cs 文件。



打开 Configuration.cs 文件，使用以下代码替换 Seed 方法：

```
protected override void Seed(MvcMovie.Models.MovieDbContext context)
{
    context.Movies.AddOrUpdate(i => i.Title,
        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    }
);
}

```

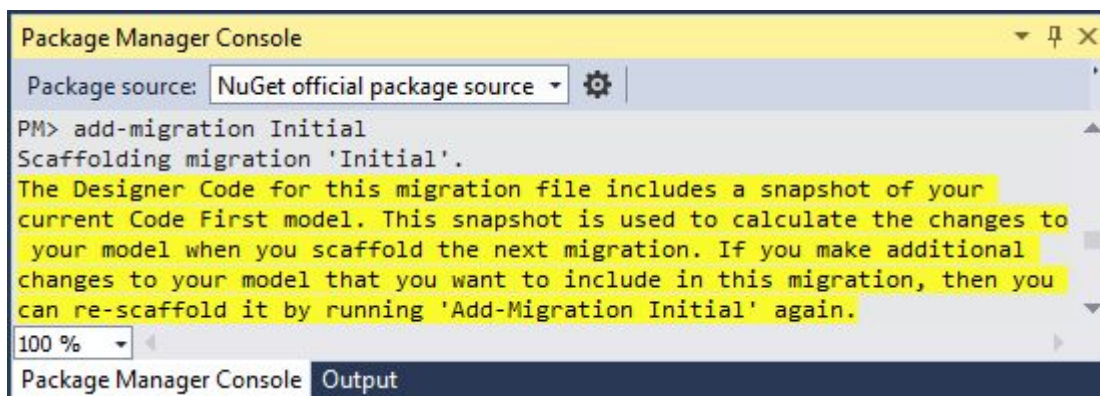
使用这段代码的时候，需要添加 `using MvcMovie.Models` 的引用。

Code First 数据迁移在每次迁移（在程序包管理器控制台中调用 `update-database`）的时候都会调用 `Seed` 方法。

在进行下一步之前，先编译解决方案，否则下一步会出错误。

下一步，为初始化迁移创建一个 `DbMigration` 类。这次迁移创建一个新数据库，这也是我们为什么要删除之前的数据库的原因。

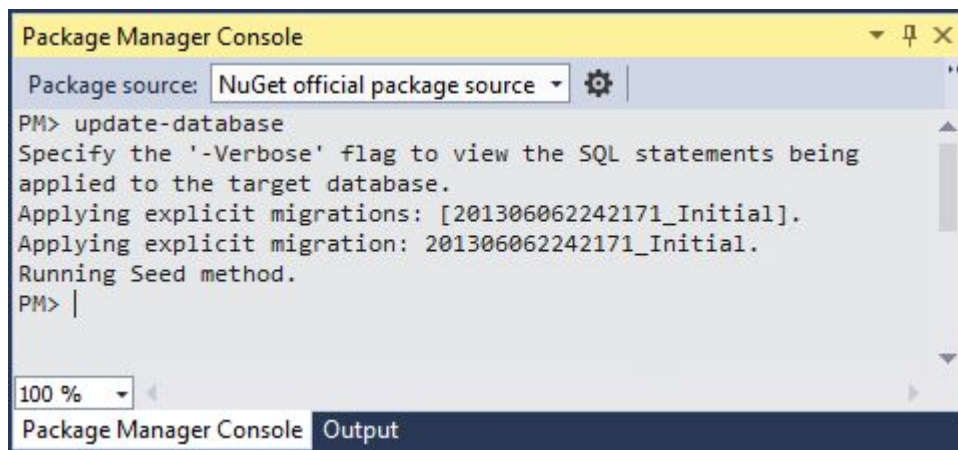
在“程序包管理器控制台”窗口，输入命令 `add-migration Initial` 创建初始化迁移。名称“Initial”是随意命名的，它用来命名创建好的迁移文件。



Code First Migrations 创建在 `Migrations` 文件夹中创建了一个文件（文件名是 `{DateStamp}_Initial.cs`），这个类包含了创建数据库结构的代码。迁移文件的文件名以

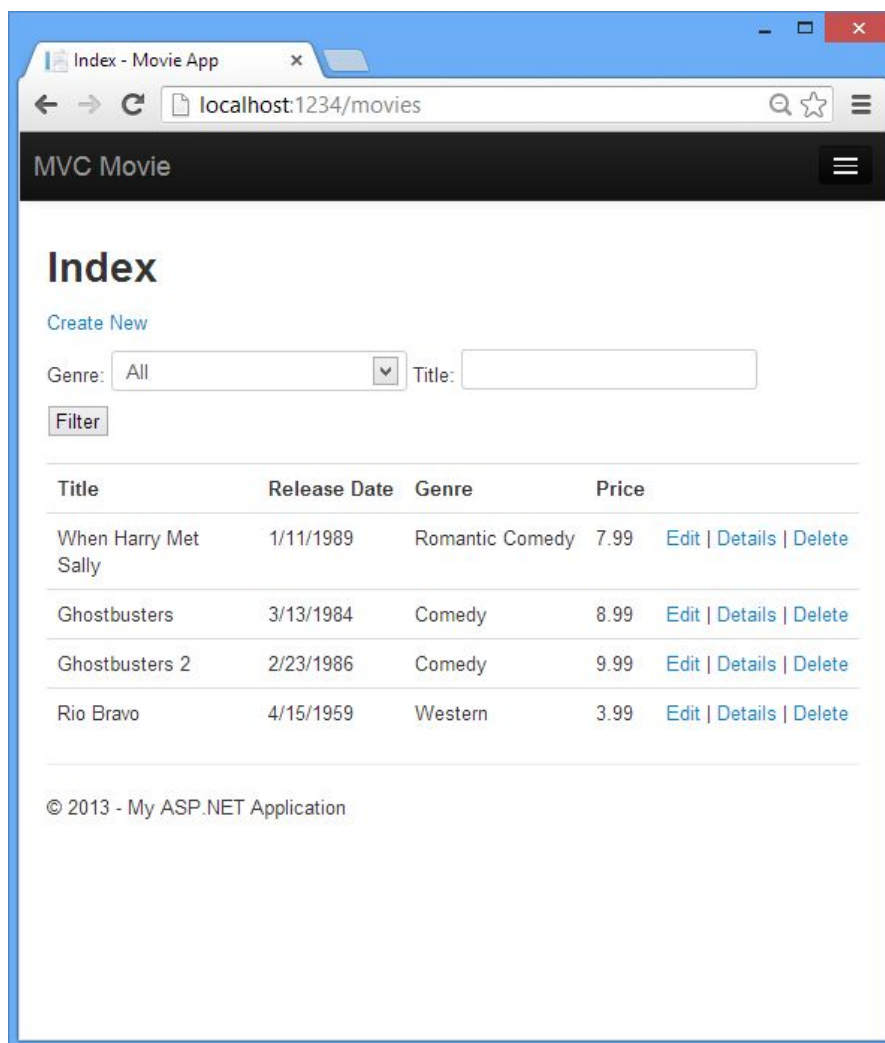
DateStamp 开头是为了更好的排序，打开 `{DateStamp}_Initial.cs` 文件，它包含了为数据库 MovieDB 创建 Movies 表的指令。当你使用下面的命令更新数据库时，`{DateStamp}_Initial.cs` 文件将会运行并创建数据库结构，然后将执行 `Seed` 方法将测试数据插入数据库中。

在“程序包管理器控制台”中输入命令 `update-database`：



```
Package Manager Console
Package source: NuGet official package source
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being
applied to the target database.
Applying explicit migrations: [201306062242171_Initial].
Applying explicit migration: 201306062242171_Initial.
Running Seed method.
PM> |
```

运行应用程序，浏览/Movies 地址，我们在 `Seed` 方法中添加的数据如下：



为 **Movie** 模型添加 **Rating** 字段

上面的内容一直在介绍如何进行数据迁移，现在开始为 **Movie** 类添加 **Rating** 字段，打开 **Movie.cs** 文件，为它添加一个 **Rating** 字段，添加后的代码如下：

```
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; }
}
```

编译解决方案。

现在我们已经更新了 **Movie** 类，你还需要修改 **|Views|Movies|Index.cshtml** 和 **|Views|Movies|Create.cshtml** 视图。修改后的代码如下：

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
<p> @Html.ActionLink("Create New", "Create")
</p>@using (Html.BeginForm("Index", "Movies", FormMethod.Get))
{
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString") <br />
        <input type="submit" value="Filter" />
    </p>
}

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Genre)
        </th>
    </tr>
</table>
```

```

        <th>                @Html.DisplayNameFor(model => model.Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Rating)
        </th>
        <th></th>
    </tr>
    @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>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>                @Html.ActionLink("Edit", "Edit", new { id = item.ID }) |
                @Html.ActionLink("Details", "Details", new { id = item.ID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.ID })
            </td>
        </tr>    }
    </table>

```



修改后的 Create.cshtml:



```

@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>Create</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset class="form-horizontal">
        <legend>Movie</legend>
        <div class="control-group">

```

```

@Html.LabelFor(model => model.Title, new { @class = "control-label" })
    <div class="controls">
@Html.EditorFor(model => model.Title)
    @Html.ValidationMessageFor(model => model.Title, null, new { @class = "help-inline" })
    </div>
</div>

    <div class="control-group">
        @Html.LabelFor(model => model.ReleaseDate, new { @class = "control-label" })
        <div class="controls">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate, null, new { @class
= "help-inline" })
            </div>
        </div>

        <div class="control-group">
@Html.LabelFor(model => model.Genre, new { @class = "control-label" })
        <div class="controls">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre, null, new { @class = "help-inline" })
            </div>
        </div>

        <div class="control-group">
@Html.LabelFor(model => model.Price, new { @class = "control-label" })
        <div class="controls">
@Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price, null, new { @class =
"help-inline" })
            </div>
        </div>

        <div class="control-group">
@Html.LabelFor(model => model.Rating, new { @class = "control-label" })
        <div class="controls">
@Html.EditorFor(model => model.Rating)
            @Html.ValidationMessageFor(model => model.Rating, null, new { @class =
"help-inline" })
            </div>
        </div>

        <div class="form-actions no-color">
            <input type="submit" value="Create" class="btn" />
        </div>

```

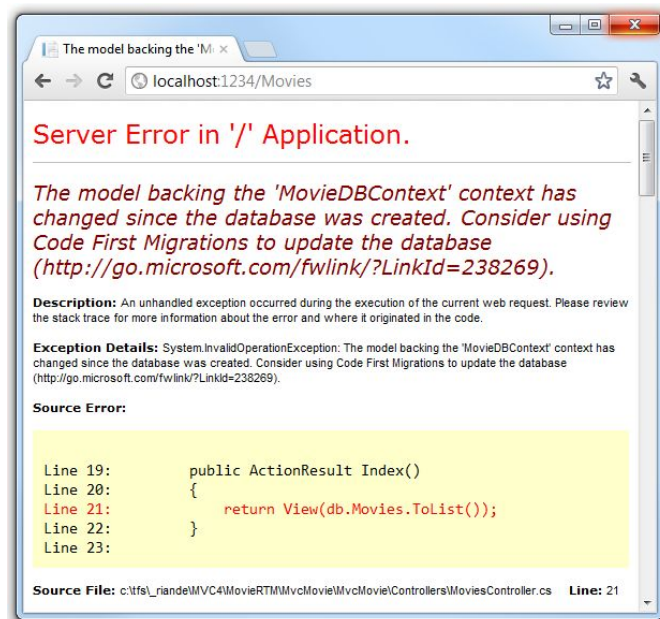
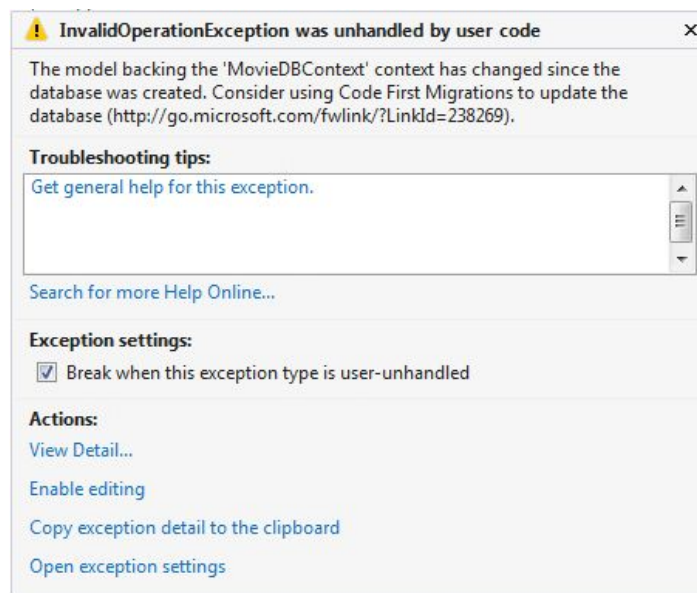


```

</fieldset>}
<div>    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```

现在我们已经程序中为 **Rating** 字段做成了修改。再次运行程序，浏览 **/movies** 地址，这时我们会得到一个错误：



出现这个错误的原因是 **Movie** 模型类发生了变化，而与它对应的数据表 **Movie** 中并不存在 **Rating** 字段。

解决这个问题有以下几种途径：

- 让 **Entity Framework** 自动删除并根据新的模型自动创建数据库。这种方式在早起开发过程中的测试数据库中非常方便，它可以快速的修改模型和数据库结构。另一方面，这样做将会使你丢失已有的数据，因此这种方式不能用在生产环境的数据库中。
- 在数据库中加上 **Rating** 字段，使数据库和 **Model** 类的结构相同。这种方式的优点是能够保留数据，你可以手动修改或使用数据库脚本修改。
- 使用 **Code First** 迁移来升级数据库结构。

在本教程中，我们使用 **Code First** 迁移。

更新 **Seed** 方法，使它为 **Rating** 字段提供一个值。

打开 **Migrations\Configuration.cs** 文件，为每一个 **Movie** 对象的 **Rating** 字段赋值。

修改后的 **Seed** 方法：

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "PG",
    Price = 7.99M
},
```

重新编译解决方案，然后打开“程序包管理器控制台”，执行命令：**add-migration Rating**

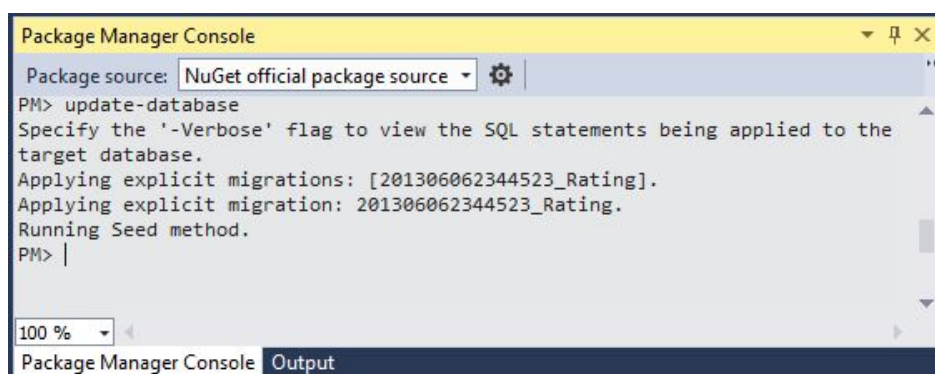
add-migration 命令告诉迁移程序去检查当前的 **Movie** 模型与当前数据库之间的差异，创建迁移数据库到最新模型的代码。名称 **Rating** 是可以随便命名的，此处用来命名迁移文件。

当命令执行完成之后，**Visual Studio** 会打开刚刚添加的继承自 **DbMigration** 的类文件，文件中有两个方法 **Up** 和 **Down**，分别用来升级和降级数据库。在 **Up** 方法中我们可以看到为数据库添加列的代码，而 **Down** 方法中的代码则是删除 **Rating** 列。

```
public partial class AddRatingMig : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Movies", "Rating", c => c.String());
    }
    public override void Down()
    {
        DropColumn("dbo.Movies", "Rating");
    }
}
```

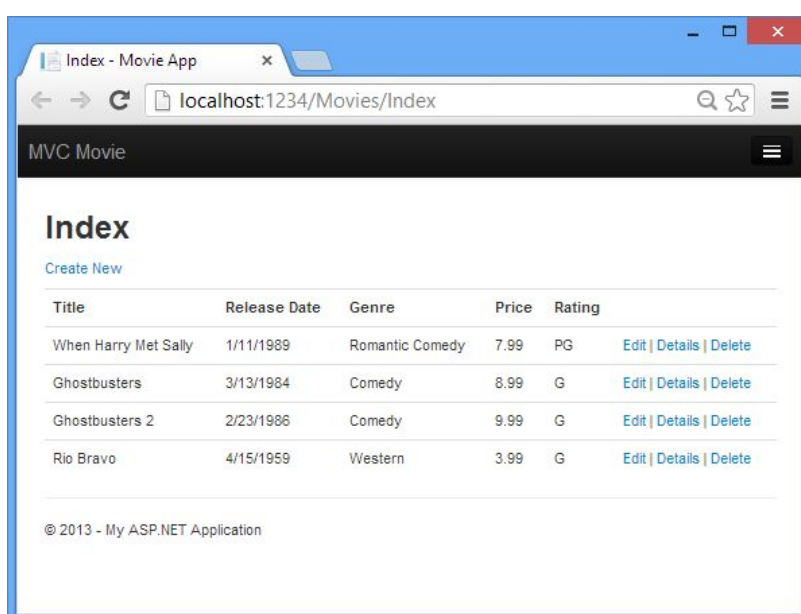
编译解决方案，然后执行命令 **update-database**。“程序包管理器控制台”窗口的输出如下

图（Rating 前的时间戳可能不尽相同）：



```
Package Manager Console
Package source: NuGet official package source
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the
target database.
Applying explicit migrations: [201306062344523_Rating].
Applying explicit migration: 201306062344523_Rating.
Running Seed method.
PM> |
```

刷新我们出错的页面，你能看到已经加入了 Rating 字段：



点击“Create New”链接试着添加一个电影信息，不要忘记为 Rating 字段赋值。

Firefox

Create - Movie App

localhost:1234/Movies/Create

MVC Movie

Create

Movie

Title

Release Date

Genre

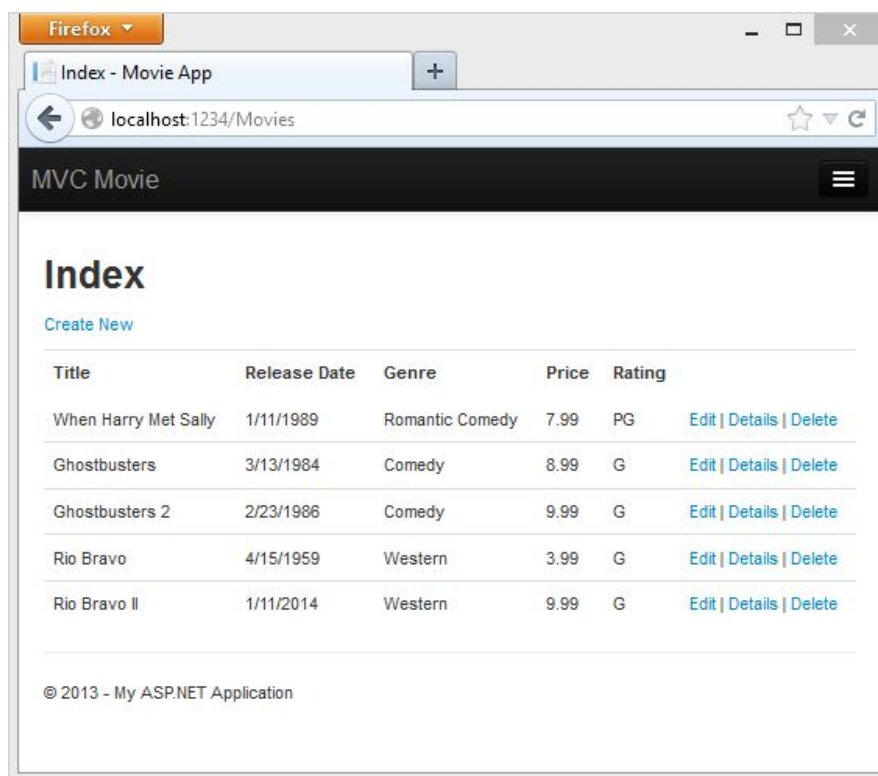
Price

Rating

[Back to List](#)

© 2013 - My ASP.NET Application

输入完成之后点击“Create”按钮，保存电影信息。



你还需要在 **Edit**、**Details** 和 **Delete** 视图添加 **Rating** 字段。

如果你再次执行 "update-database" 命令，将不会做出任何修改，因为数据库结构和模型的结构已经相同了。

现在，通过项目中使用数据迁移，我们在添加字段或更新模型结构的时候不用再删除数据库了。在下一节中，我们将对结构做出更多的更改，并使用数据迁移来更新数据库。

10: 添加验证

在本节中，我们将为 **Movie** 模型添加验证逻辑，并确认验证规则在用户试图使用程序创建和编辑电影时有效。

DRY 原则

ASP.NET MVC 的一个核心原则是 DRY (Don't Repeat Yourself - 不做重复的事情)。ASP.NET MVC 鼓励你一次性的指定功能或行为，然后应用程序的其它地方通过映射得到它，这样一来就减少了大量的代码，从而减少了出错误的可能性，并且更易于维护。

ASP.NET MVC 和 Entity Framework Code First 提供的验证能是 DRY 原则的不错的实践。你可以在一处（在模型类中）定义验证规则，从而在应用程序中的所有地方都可以使用这个规则。

接下来让我们看看如何在现在的 **Movie** 中添加高级的验证规则吧。

为模型添加验证规则

现在我们开始为 **Movie** 类添加一些验证规则。

打开文件 **Movie.cs**，注意命名空间 **System.ComponentModel.DataAnnotations** 并不包含 **System.Web.DataAnnotations** 提供了内置的验证特性，你可以将它们用在任何类或属性中（它还包含了像 **DataType** 这样的格式化的特性，它们不参与任何验证）。

为 **Movie** 类添加一些内置的验证规则，修改后的代码如下：

```
public class Movie
{
    public int ID { get; set; }
    [Required]
    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }
    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    [Required]
    public string Genre { get; set; }
    [Range(1, 100)]
    [DataType(DataType.Currency)]
    public decimal Price { get; set; }
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

使用数据迁移来更新数据库结构。编译解决方案，然后打开“程序包管理器控制台”窗口，执行下面的命令：

add-migration DataAnnotations
update-database

当这两个命令执行完成之后，**Visual Studio** 为我们创建了 **DataAnnotations** 类，它继承自 **DbMigration**。打开文件，在它的 **Up** 方法中，你会看到升级结构的代码：

```
public override void Up()
{
    AlterColumn("dbo.Movies", "Title", c => c.String(nullable: false, maxLength: 60));
    AlterColumn("dbo.Movies", "Genre", c => c.String(nullable: false));
    AlterColumn("dbo.Movies", "Rating", c => c.String(nullable: false, maxLength: 5));
}
```

从代码中可以看出，**Title**、**Genre** 和 **Rating** 三个字段不再允许为空（这意味着你必须输入

一个值)。Rating 字段的最大长度为5，Title 的最大长度为60，最小长度为3。

Code First 确保在保存到数据库的时候使用你指定的规则对数据进行验证，例如，下面的代码在调用 **SaveChanges** 的时候会抛出一个错误：



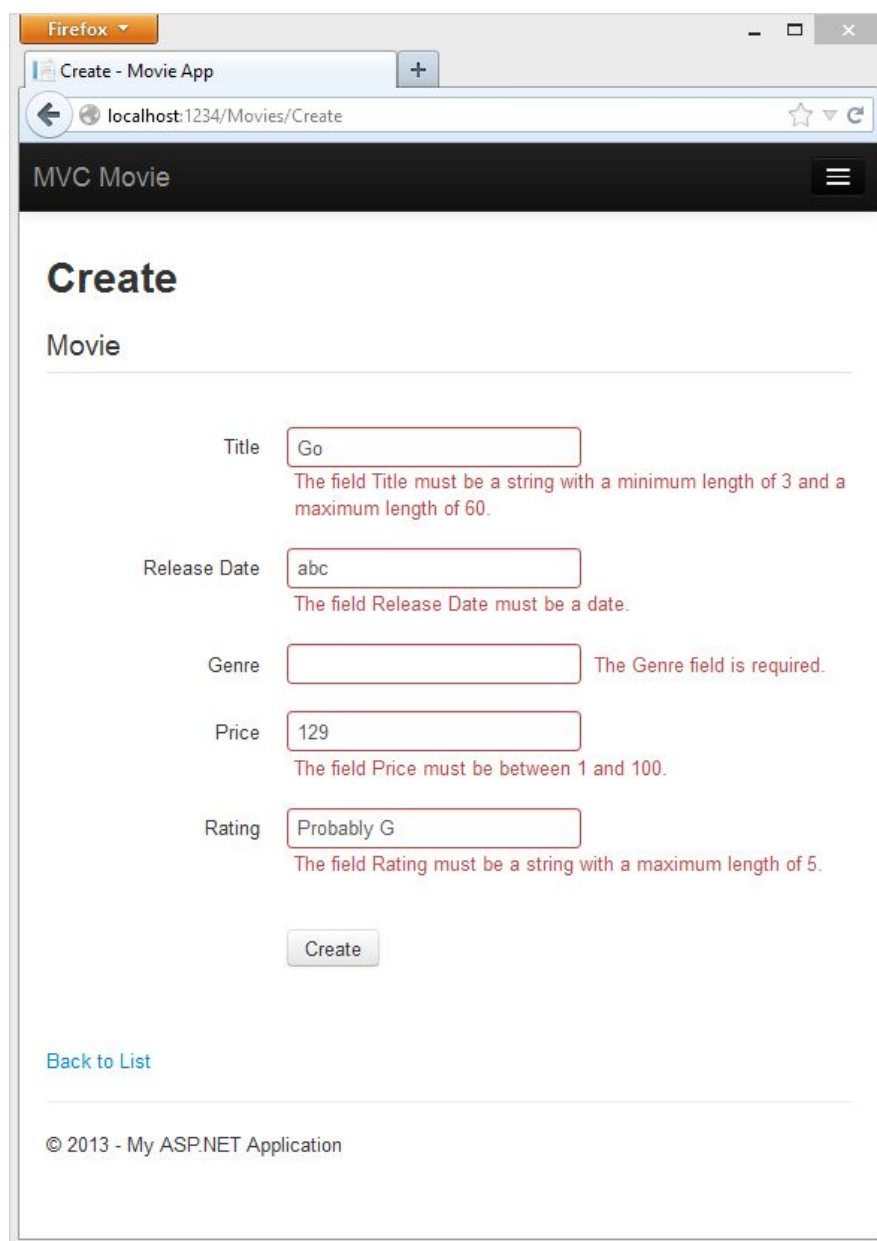
```
MovieDbContext db = new MovieDbContext();  
Movie movie = new Movie();  
movie.Title = "Gone with the Wind";  
db.Movies.Add(movie);  
db.SaveChanges(); // <= 会引发一个服务器端的错误，因为 movie 的必须的字段没有赋值
```



验证规则在保存的时候自动生效使得程序变得更为健壮，它可以在是你忘记去验证，而在不经意间阻止不合法的进入数据库。

ASP.NET MVC 客户端验证

运行应用程序，浏览地址 `/movies`，点击“Create New”链接添加一个电影。如果我们在输入过程中出现一些不合法的数据，客户端将会显示一些错误，这是通过 **jQuery** 客户端验证来实现的。下面是一些错误信息：



当出现错误的时候，文本框会被加上红色的边框，并且会显示一段错误描述信息。这些错误信息可以在客户端（使用 **Javascript** 和 **jQuery**）和服务端（当客户端 **Javascript** 无效时）生效。

一个真正的好处是，你不需要在 **MoviesController** 或 **Create.cshtml** 中修改一行代码来启用客户端验证，控制器和视图会根据我们之前定义在 **Movie** 类中的验证特性自动选择验证规则。

表单数据出现错误的时候不会被提交到服务器端。

验证是如何工作的

你可能会觉得奇怪，客户端验证是如何在没有修改控制器或视图代码的情况下生成的。下面的代码显示了 **MovieController** 的 **Create** 方法，它和我们前面的教程中的 **Create** 代码一样，并没有经过修改：


```

//
// GET: /Movies/Create
public ActionResult Create()
{
    return View();
}
//
// POST: /Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}

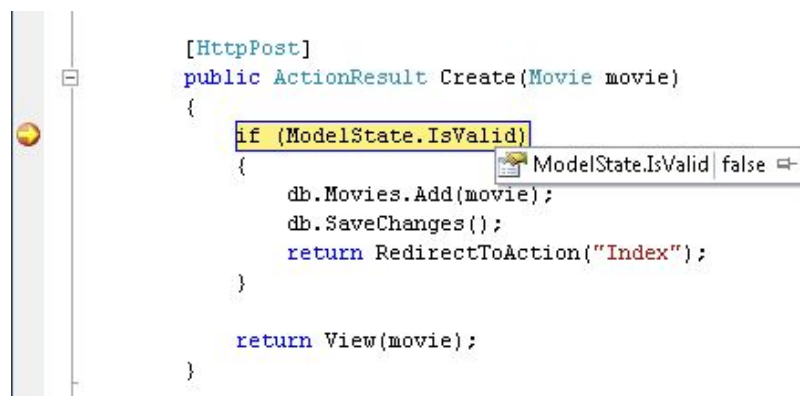
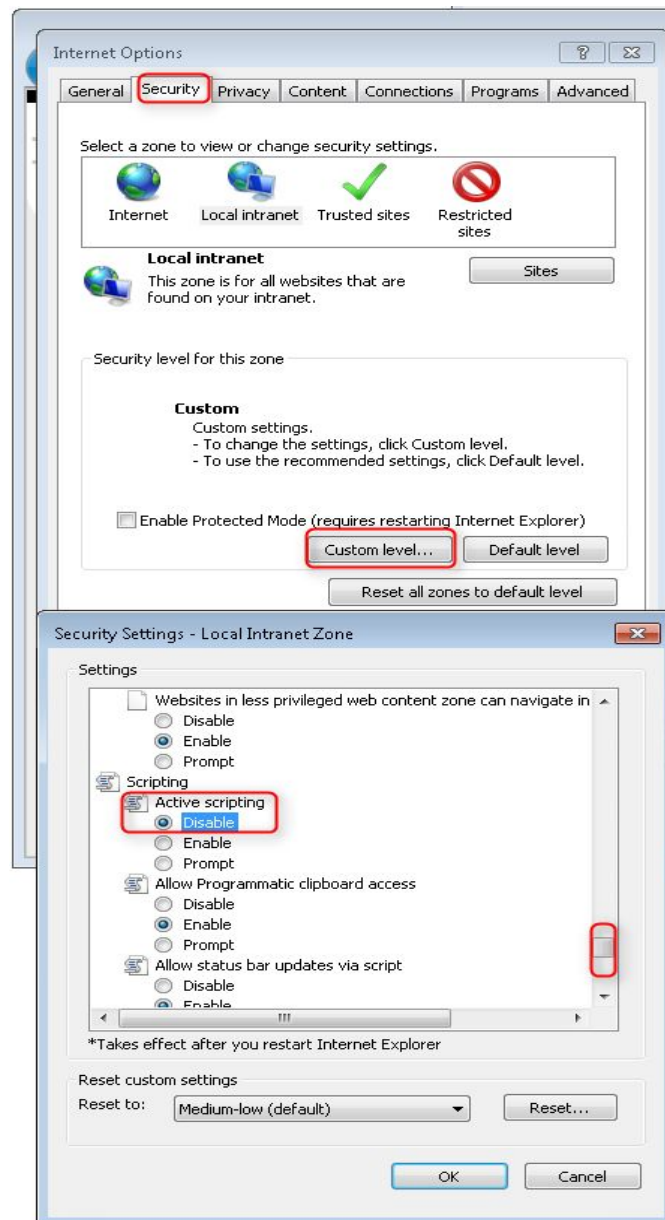
```

第一个 **Create** 方法显示一个表单，第二个 **Create** 方法用来处理 **POST** 请求提交的表单数据。第二个 **Create** 方法调用 **ModelState.IsValid** 来检查 **movie** 数据是否存在验证错误，调用这个方法检查验证规则，如果存在错误，**Create** 方法将会重新显示这个表单，如果没有，则会将 **Movie** 数据保存到数据库。在我们的例子中，当验证出现错误的时候表单不会提交到服务器，第二个 **Create** 方法将不会调用。如果你禁用了 **Javascript** 客户端验证，第二个 **Create** 方法则会调用 **ModelState.IsValid** 对数据进行检查。

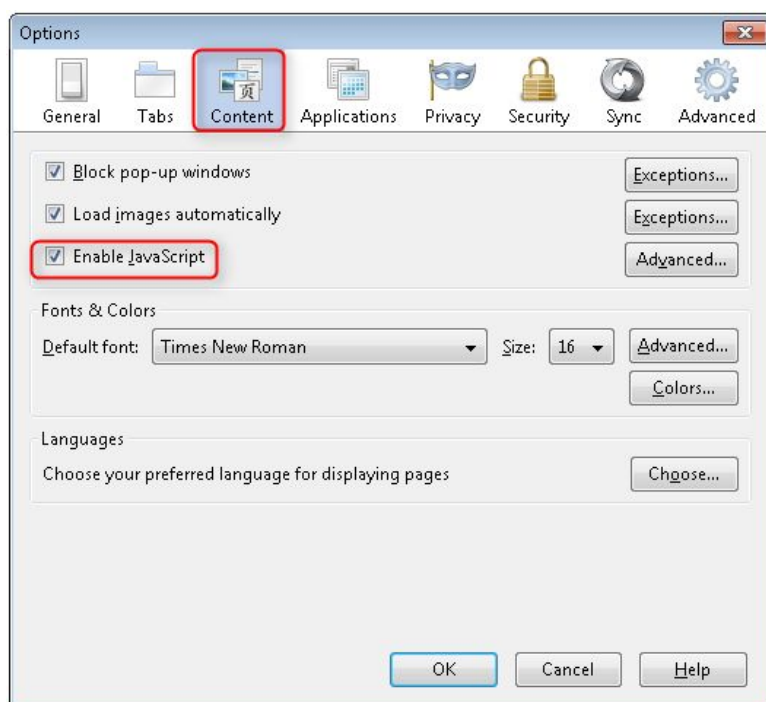
你可以通过在 **HttpPost Create** 方法中添加断点来监视是否被调用。当客户端出现错误的时候表单将不会提交，如果我们禁用客户端的 **Javascript**，表单将错误的数据提交到服务器，断点将会跟踪到。下面我们介绍一下如何在 **IE** 和 **Google** 浏览器中禁用 **Javascript**。

在 **IE** 中禁用 **Javascript**

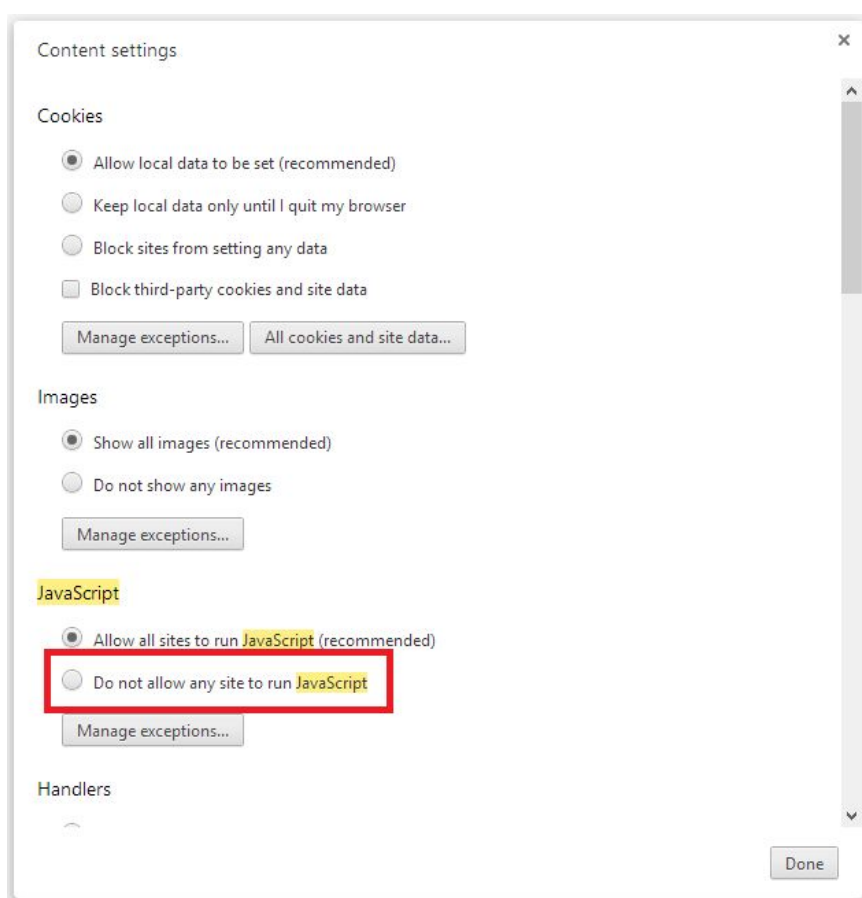
打开 **Internet** 选项，在安全选项中选中“本地 **Intranet**”，点击“自定义级别”按钮：



在火狐（FireFox）中禁用 Javascript:



在 Google Chrome 中禁用 Javascript:



下面是 `Create.cshtml` 视图的代码，它被控制器中的 `Create` 方法用来显示初始的 form 表单，或在发生错误时重新显示带错误信息的表单数据。

```

@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>Create</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)
    <fieldset class="form-horizontal">
        <legend>Movie</legend>
        <div class="control-group">
            @Html.LabelFor(model => model.Title, new
{ @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title, null, new { @class =
"help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.ReleaseDate,
new { @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model =>
model.ReleaseDate)
                @Html.ValidationMessageFor(model => model.ReleaseDate, null, new
{ @class = "help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.Genre, new
{ @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Genre)
                @Html.ValidationMessageFor(model => model.Genre, null, new { @class =
"help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.Price, new
{ @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Price)
                @Html.ValidationMessageFor(model => model.Price, null, new { @class =
"help-inline" })
            </div>
        </div>
        <div class="control-group">
            @Html.LabelFor(model => model.Rating, new
{ @class = "control-label" })
            <div class="controls">
                @Html.EditorFor(model => model.Rating)
                @Html.ValidationMessageFor(model => model.Rating, null, new { @class =

```

```

"help-inline" })
    </div>
</div>
<div class="form-actions no-color">
    <input type="submit" value="Create" class="btn" />
</div>
</fieldset>}
<div> @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}

```



模型格式化输出

打开文件 `Movie.cs`，检查 `Movie` 类，`System.ComponentModel.DataAnnotations` 命名空间除了一套内置的验证特性外，还提供了格式化特性。我们已经在 `ReleaseDate` 和 `Price` 字段用到过 `DataType` 枚举，下面的代码展示了 `ReleaseDate` 和 `Price` 属性中使用的 `DisplayFormat` 特性：



```

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

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

```



`DataType` 特性不是验证特性，他们用来告诉试图引擎如何绘制 **HTML**。在上面的例子中，`DataType.Date` 特性使 `ReleaseDate` 显示的时候只显示日期部分，而不显示时间。下面的 `DataType` 特性不验证数据的格式：

```

[DataType(DataType.EmailAddress)]
[DataType(DataType.PhoneNumber)]
[DataType(DataType.Url)]

```

这些特性只为试图引擎格式化显示数据时提供建议。你可以使用 `RegularExpression` 特性连验证数据的格式。

除了使用现成的 `DataType` 格式化特性之外，你还可以明确指定 `DataFormatString` 值。下面的代码展示了 `ReleaseDate` 属性使用格式化字符串的情况，你可以使用它来不显示 `ReleaseDate` 日期的时间部分：

```


[DisplayFormat(DataFormatString = "{0:d}")]
public DateTime ReleaseDate { get; set; }

```

下面的代码将 Price 显示为货币的格式:

```
[DisplayFormat(DataFormatString = "{0:c}")]  
public decimal Price { get; set; }
```


完整的 Movie 类代码如下:



```
public class Movie  
{  
    public int ID { get; set; }  
  
    [Required]  
    [StringLength(60, MinimumLength = 3)]  
    public string Title { get; set; }  
  
    [Display(Name = "Release Date")]  
    [DataType(DataType.Date)]  
    public DateTime ReleaseDate { get; set; }  
  
    [Required]  
    public string Genre { get; set; }  
  
    [Range(1, 100)]  
    [DataType(DataType.Currency)]  
    public decimal Price { get; set; }  
  
    [StringLength(5)]  
    [Required]  
    public string Rating { get; set; }  
}
```



面的代码展示了如何将特性合并在一行显示:



```
public class Movie  
{  
    public int ID { get; set; }  
  
    [Required, StringLength(60, MinimumLength = 3)]  
    public string Title { get; set; }  
  
    [Display(Name = "Release Date"), DataType(DataType.Date)]  
    public DateTime ReleaseDate { get; set; }  
  
    [Required]
```

```

public string Genre { get; set; }

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

[Required, StringLength(5)]
public string Rating { get; set; }
}

```

在本系列的下一部分，我们将回顾整个应用程序，并对自动生成的 **Details** 和 **Delete** 方法做一些改进。

11: Details 和 Delete 方法详解

在教程的这一部分，我们将研究一下自动生成的 **Details** 和 **Delete** 方法。

Details 方法

打开 **Movie** 控制器，找到 **Details** 方法。

```

//
// GET: /Movies/Details/5
public ActionResult Details(Int32 id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

```

Code First 通过使用 **Find** 方法找到要显示的 **Movie** 对象。如果 **movie** 对象为 **null**，则返回 **HttpNotFound()**，这个判断非常有必要，试想如果有黑客要攻击你的网站，他们可能将通过修改 URL 来尝试访问你的网站，例如将

<http://localhost:xxxx/Movies/Details/1>

修改为

<http://localhost:xxxx/Movies/Details/12345>

如果你不去验证 **movie** 是否为空，则会将数据库错误返回给黑客，这样会暴漏网站的更多信息。

Delete 和 DeleteConfirmed 方法

```
//  
// GET: /Movies/Delete/5  
public ActionResult Delete(Int32 id)  
{  
    Movie movie = db.Movies.Find(id);  
    if (movie == null)  
    {  
        return HttpNotFound();  
    }  
    return View(movie);  
}  
//  
// POST: /Movies/Delete/5  
[HttpPost, ActionName("Delete")]  
[ValidateAntiForgeryToken]  
public ActionResult DeleteConfirmed(Int32 id)  
{  
    Movie movie = db.Movies.Find(id);  
    db.Movies.Remove(movie);  
    db.SaveChanges();  
    return RedirectToAction("Index");  
}
```

第一个 **Delete** 方法并没有删除 **Movie**，而是返回了可以删除确认页面，在确认页面中，会创建 **HttpPost** 请求，完成 **Movie** 的删除。直接使用 **HttpGet** 请求来删除 **Movie** 会打开一个安全漏洞。

真正删除数据的方法名字是 **DeleteConfirmed**。下面是两个方法的定义：

```
public ActionResult Delete(Int32 id)  
[HttpPost, ActionName("Delete")]  
[ValidateAntiForgeryToken]  
public ActionResult DeleteConfirmed(Int32 id)
```

CLR 对于重构的方法，要求方法名相同，但参数不同。然而，我们在这里用到的两个删除方法，他们都接收一个整形的参数，如果方法名也形同，那么就构成语法错误了。

为了解决这个问题，有以下几种解决办法：

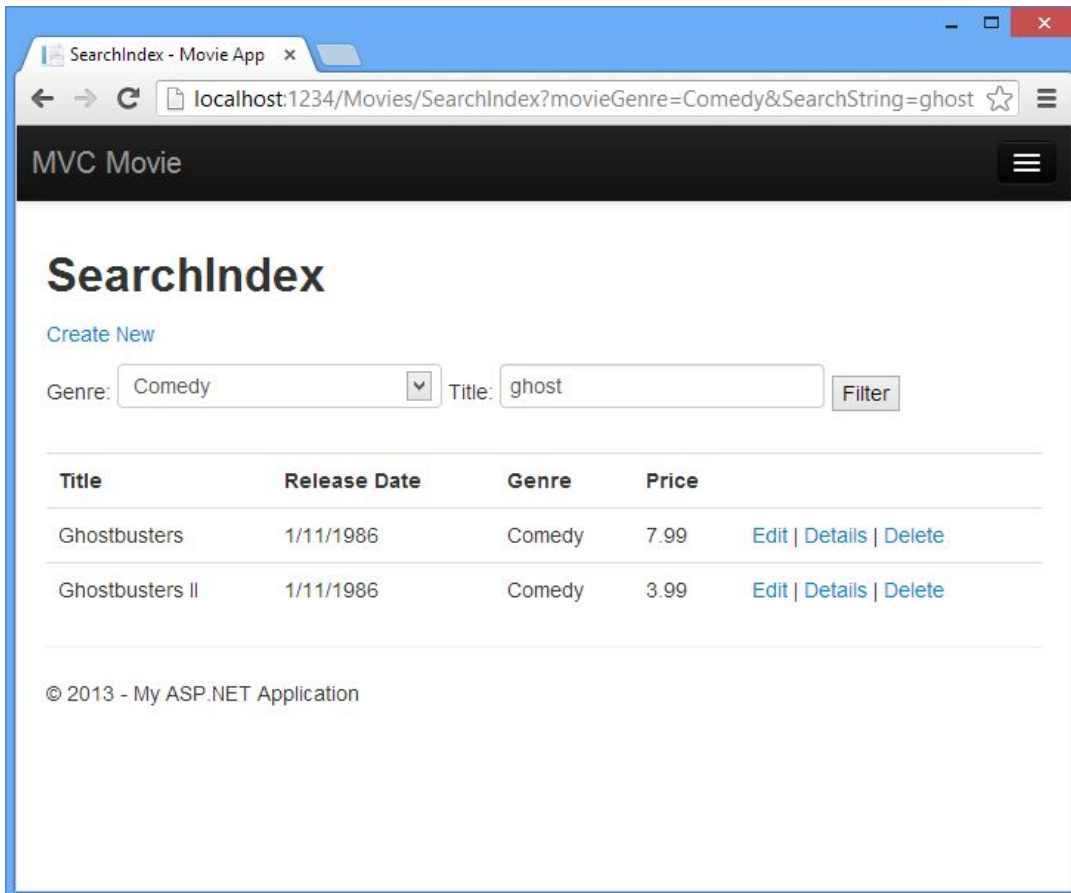
第一种办法是使用不同的方法名。这也是 **ASP.NET MVC 5** 支架所采用的方法。然而，使用这种方法还有一些小问题：**ASP.NET** 通过地址段来映射方法名，如果将方法重命名，路由通常会找不到方法，解决的办法已经在上面的例子中了，我们可以为 **DeleteConfirmed** 方法添加特性 **ActionName("Delete")**。这样一来，当 URL 中包含 **Delete** 的 **POST** 请求都会被分配给方法 **DeleteConfirmed**。

另外一种常用解决的方法是使用相同的名字，然后为 **POST** 方法添加一个未使用的参数。例如，一些开发者会添加 **FormCollection** 类型的参数传递给 **POST** 方法，然后不使用这个参数：

```
public ActionResult Delete(FormCollection fcNotUsed, int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

总结

现在你已经拥有了一个完整的 **ASP.NET MVC 5** 应用程序，你可以使用 **LocalDB** 来存储数据，可以新建、编辑、查找、删除 **Movie** 数据。



Title	Release Date	Genre	Price	
Ghostbusters	1/11/1986	Comedy	7.99	Edit Details Delete
Ghostbusters II	1/11/1986	Comedy	3.99	Edit Details Delete