
ASP.NET MVC 6 Documentation

Release

Microsoft

March 02, 2016

1	Overview of ASP.NET MVC	3
2	Getting Started	5
2.1	Building your first MVC 6 application	5
2.2	Building Your First Web API with MVC 6	87
3	Tutorials	103
3.1	Music Store Tutorial	103
3.2	Creating Backend Services for Native Mobile Applications	103
4	Models	105
4.1	Model Binding	105
4.2	Model Validation	107
4.3	Formatting	108
4.4	Custom Formatters	108
5	Views	109
5.1	Razor Syntax	109
5.2	Dynamic vs Strongly Typed Views	109
5.3	HTML Helpers	109
5.4	Tag Helpers	110
5.5	Partial Views	132
5.6	Injecting Services Into Views	132
5.7	View Components	138
5.8	Creating a Custom View Engine	145
5.9	Building Mobile Specific Views	146
6	Controllers	147
6.1	Controllers, Actions, and Action Results	147
6.2	Routing to Controller Actions	149
6.3	Error Handling	149
6.4	Filters	149
6.5	Dependency Injection and Controllers	149
6.6	Testing Controller Logic	154
6.7	Areas	154
6.8	Working with the Application Model	156
7	Performance	157
7.1	Response Caching	157

8	Security	161
8.1	Authorization Filters	161
8.2	Enforcing SSL	161
8.3	Anti-Request Forgery	161
8.4	Specifying a CORS Policy	162
9	Migration	165
9.1	Migrating From ASP.NET MVC 5 to MVC 6	165
9.2	Migrating Configuration From ASP.NET MVC 5 to MVC 6	179
9.3	Migrating From ASP.NET Web API 2 to MVC 6	180
9.4	Migrating Authentication and Identity From ASP.NET MVC 5 to MVC 6	188
10	Contribute	193

Note: This documentation is a work in progress. Topics marked with a are placeholders that have not been written yet. You can track the status of these topics through our public documentation [issue tracker](#). Learn how you can [contribute](#) on GitHub. Help shape the scope and focus of the ASP.NET content by taking the [ASP.NET 5 Documentation Survey](#).

Overview of ASP.NET MVC

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

Getting Started

2.1 Building your first MVC 6 application

2.1.1 Getting started with ASP.NET MVC 6

By [Rick Anderson](#)

This tutorial will teach you the basics of building an ASP.NET MVC 6 web app using [Visual Studio 2015](#).

Sections

- *[Install Visual Studio and ASP.NET](#)*
- *[Create a web app](#)*

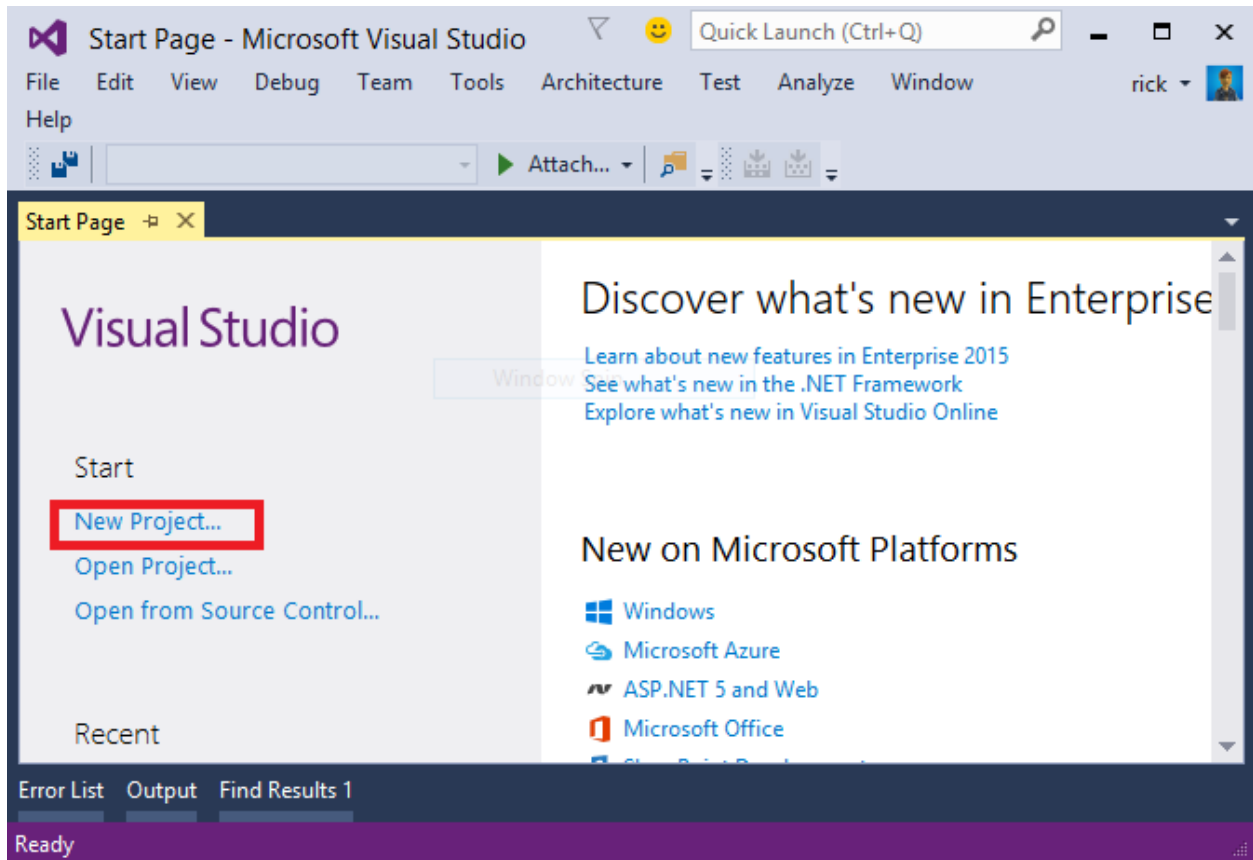
Install Visual Studio and ASP.NET

Visual Studio is an IDE (integrated development environment) for building apps. Similar to using Microsoft Word to write documents, you'll use Visual Studio to create web apps.

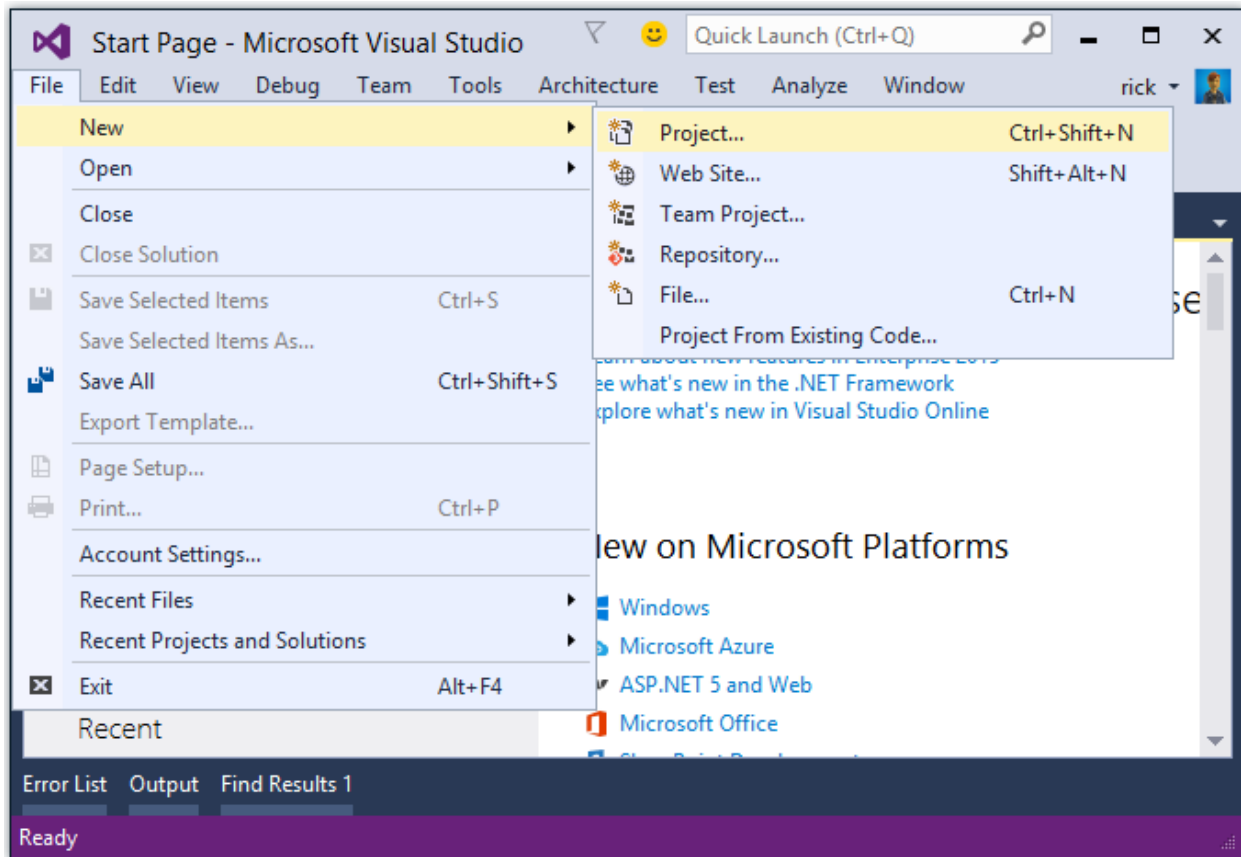
Install [ASP.NET 5](#) and [Visual Studio 2015](#).

Create a web app

From the Visual Studio **Start** page, tap **New Project**.

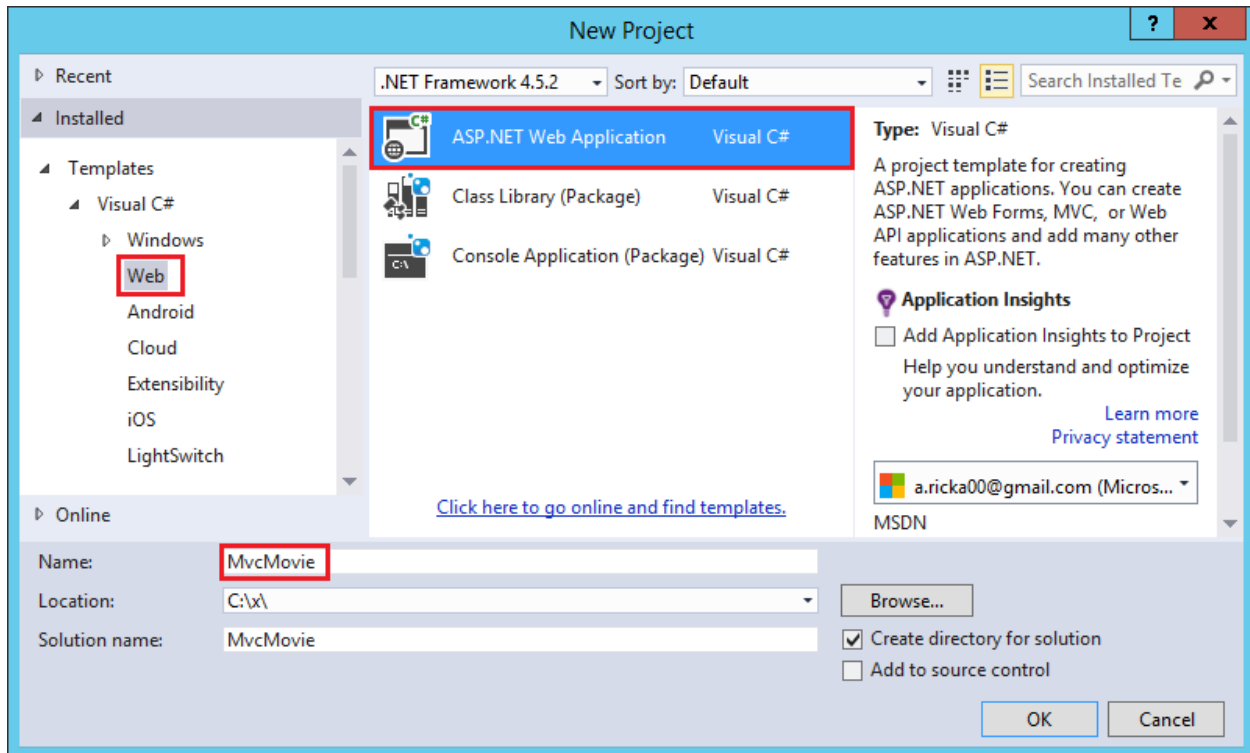


Alternatively, you can use the menus to create a new project. Tap **File > New > Project**.

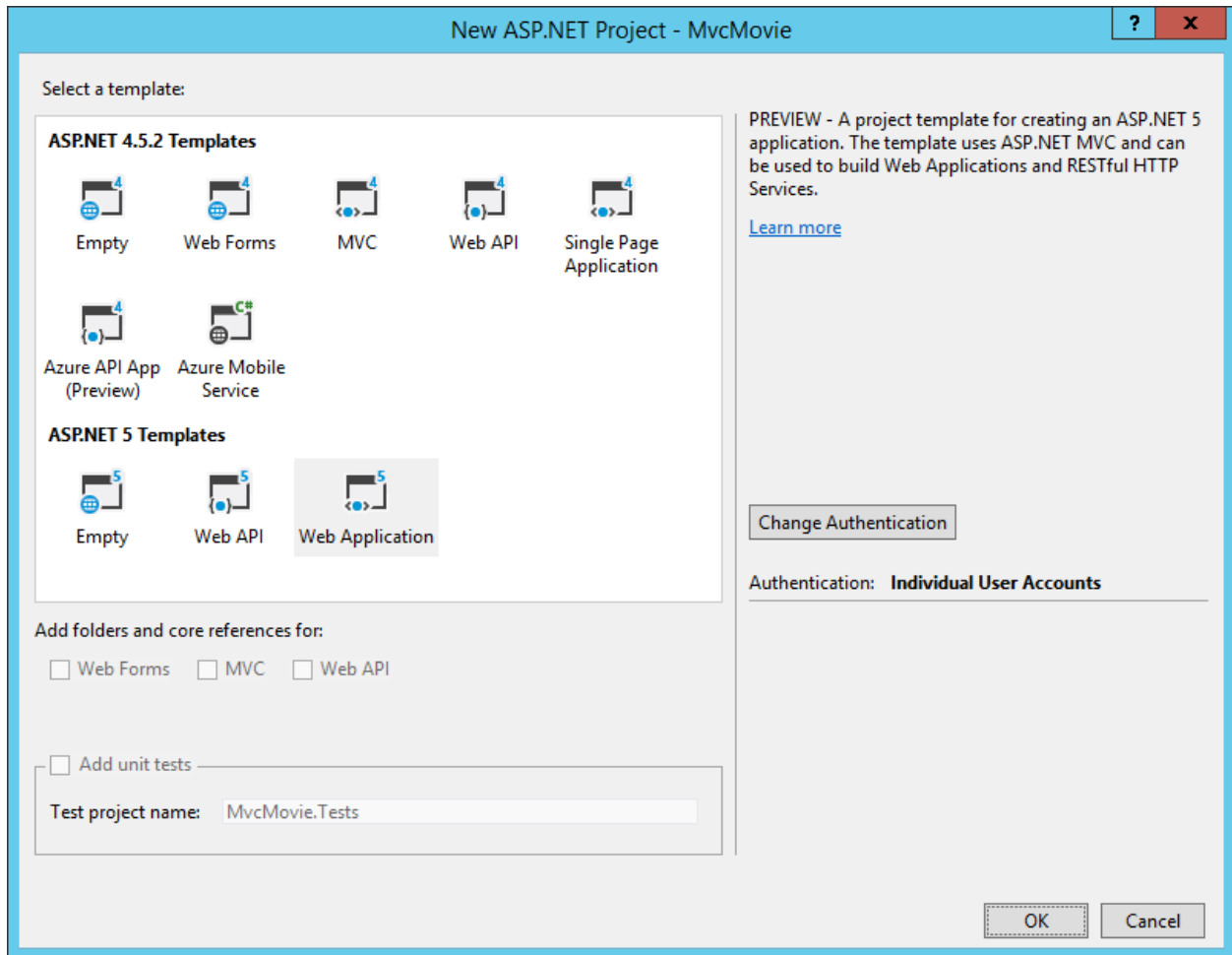


Complete the **New Project** dialog:

- In the left pane, tap **Web**
- In the center pane, tap **ASP.NET Web Application**
- Name the project “MvcMovie” (It’s important to name the project “MvcMovie” so when you copy code, the namespace will match.)
- Tap **OK**

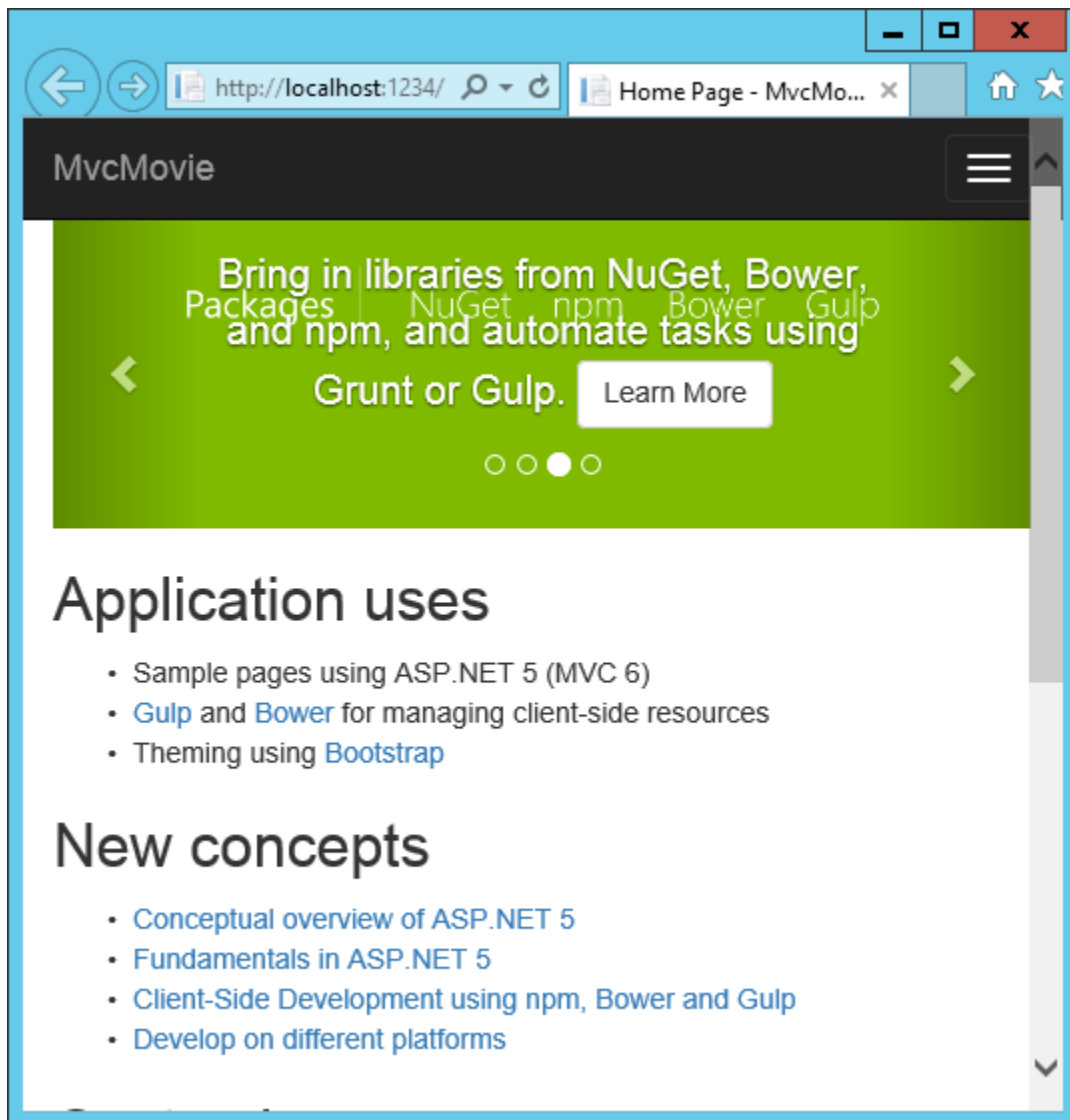


In the **New ASP.NET Project - MvcMovie** dialog, tap **Web Application**, and then tap **OK**.

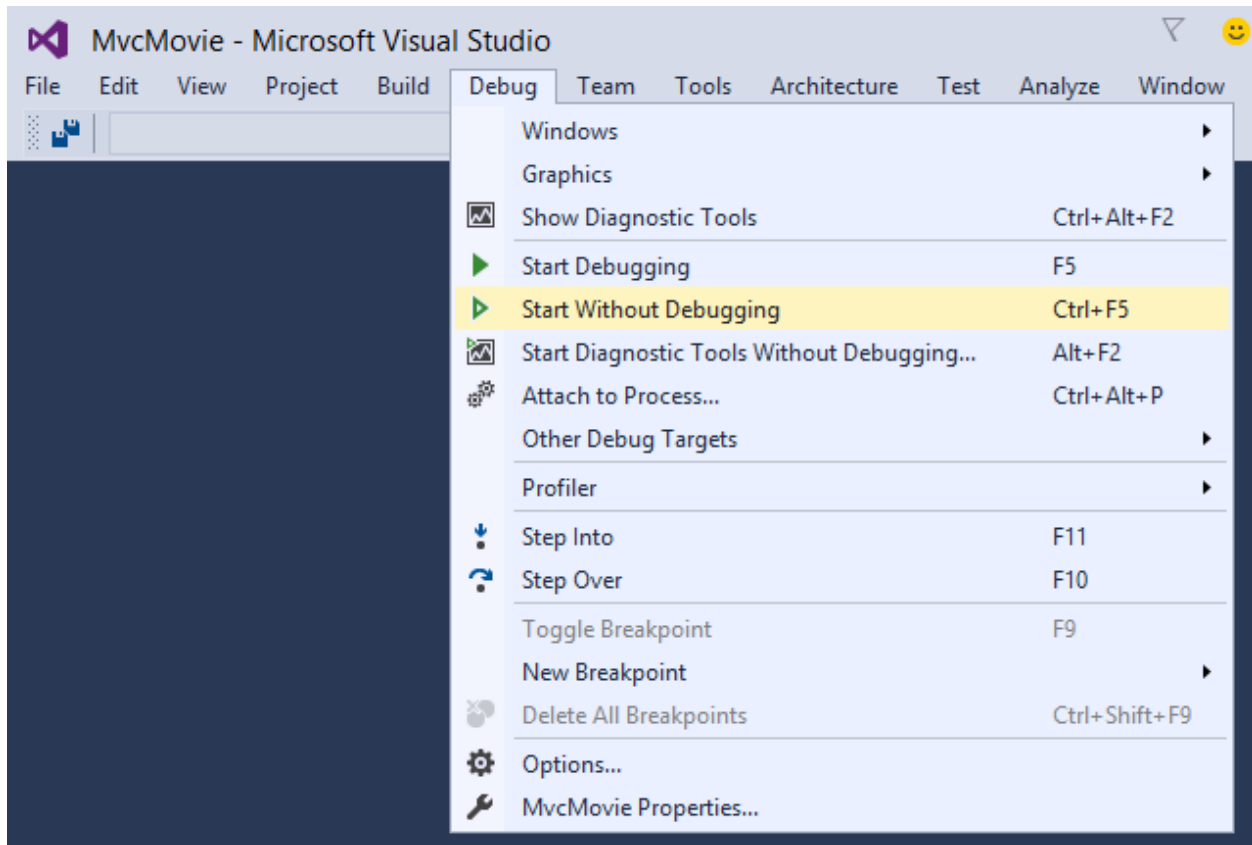


Visual Studio used a default template for the MVC project you just created, so you have a working app right now by entering a project name and selecting a few options. This is a simple “Hello World!” project, and it’s a good place to start,

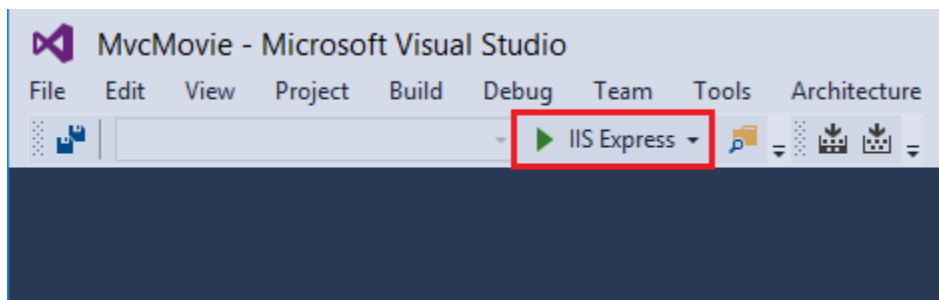
Tap **F5** to run the app in debug mode or **Ctl-F5** in non-debug mode.



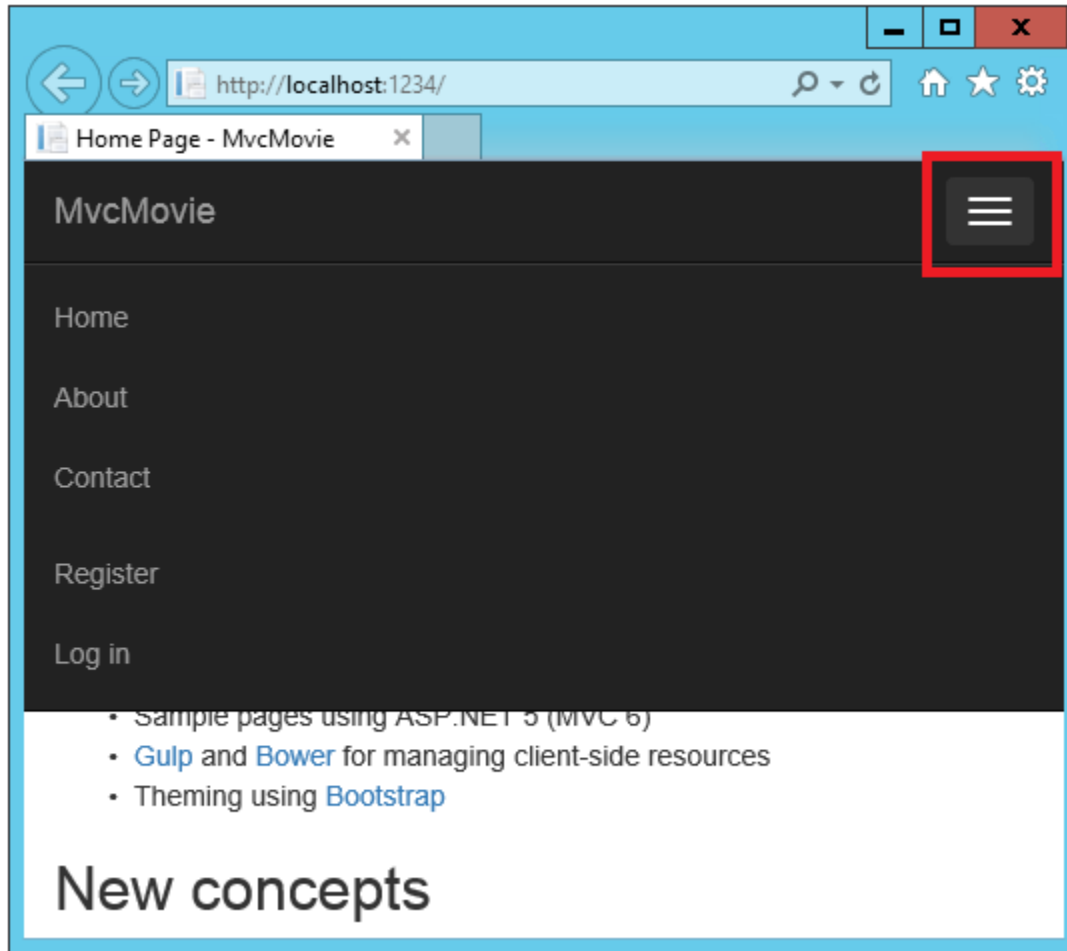
- Visual Studio starts [IIS Express](#) and runs your app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` always points to your own local computer, which in this case is running the app you just created. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the port number is 1234. When you run the app, you'll see a different port number.
- Launching the app with **Ctrl-F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



- You can debug the app by tapping the **IIS Express** button



Right out of the box the default template gives you Home, Contact, About, Register and Log in pages. The browser image above doesn't show these links. Depending on the size of your browser, you might need to click the navigation icon to show them.



In the next part of this tutorial, we'll learn a about MVC and start writing some code.

2.1.2 Adding a controller

By [Rick Anderson](#)

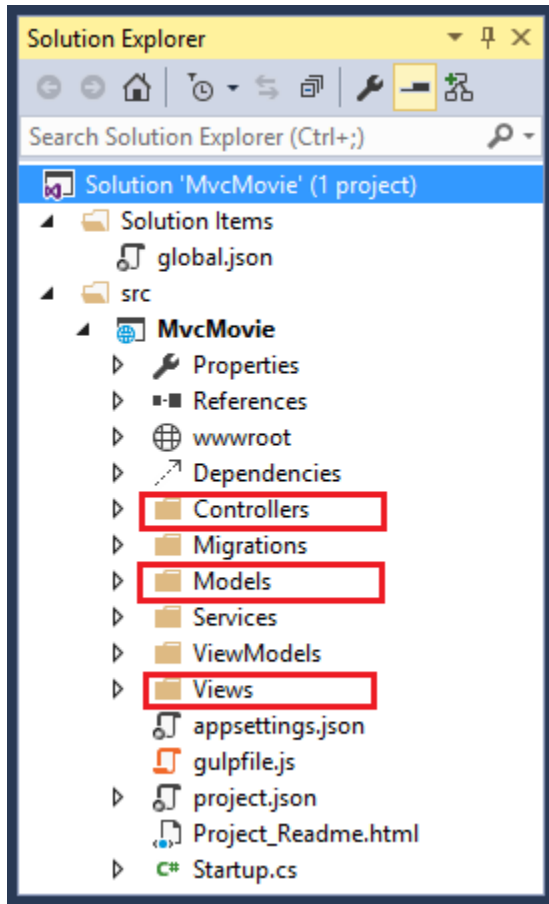
The Model-View-Controller (MVC) architectural pattern separates an app into three main components: the **Model**, the **View**, and the **Controller**. The MVC pattern helps you create apps that are testable and easier to maintain and update than traditional monolithic apps. MVC-based apps contain:

- **Models:** Classes that represent the data of the app and that use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a SQL Server database.
- **Views:** Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **Controllers:** Classes that handle browser requests, retrieve model data, and then specify view templates that return a response to the browser. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database.

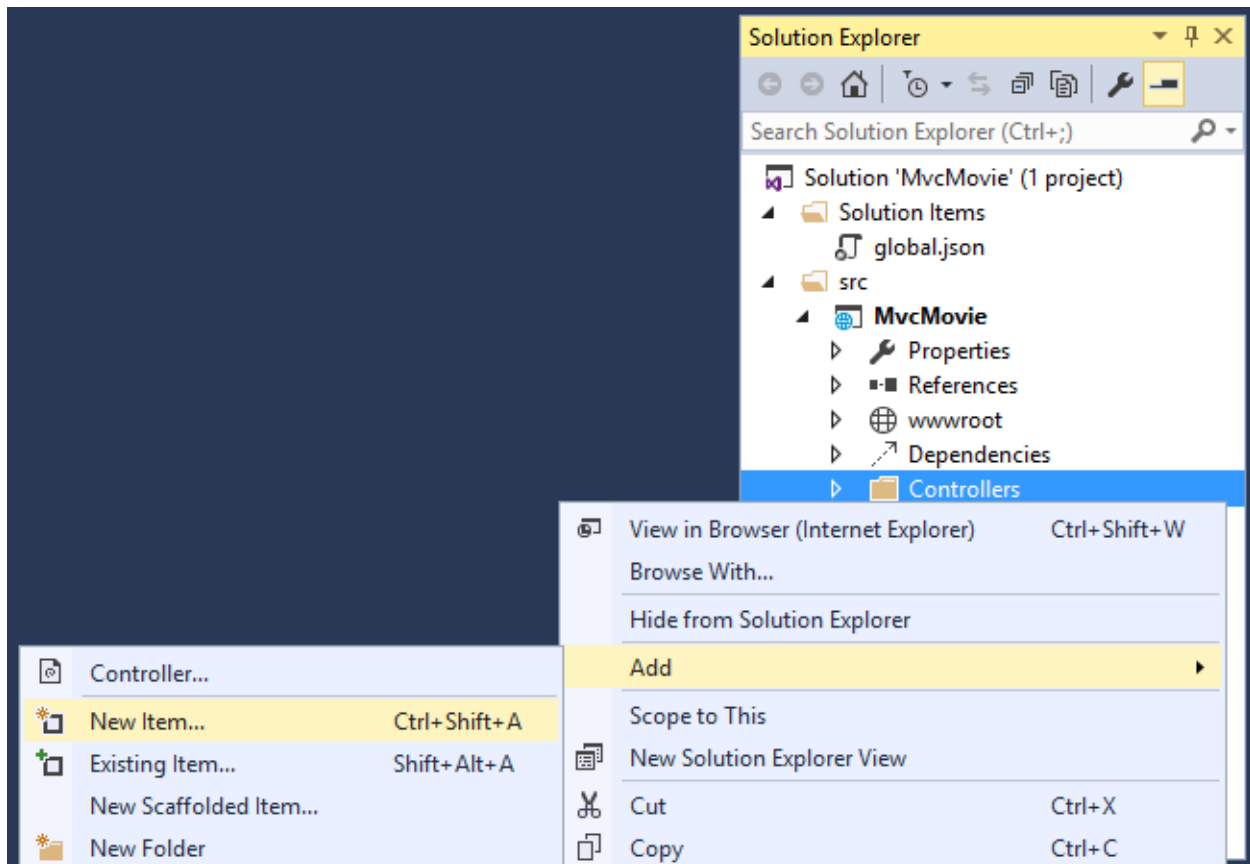
The MVC pattern helps you create applications that separate the different aspects of the app (input logic, business

logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the application. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

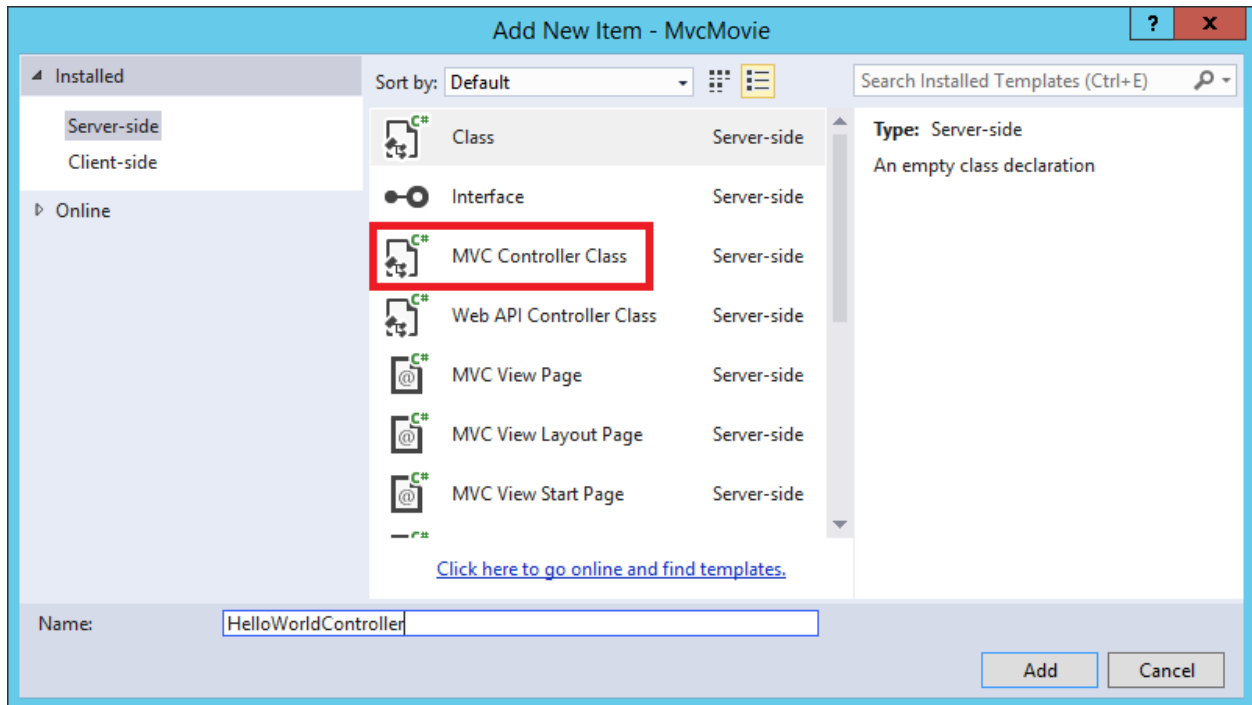
We'll be covering all these concepts in this tutorial series and show you how to use them to build a simple movie app. The following image shows the *Models*, *Views* and *Controllers* folders in the MVC project.



- In **Solution Explorer**, right-click the *Controllers*, and then **Add > New Item**.



- In the **Add New Item - Movie** dialog
 - Tap **MVC Controller Class**
 - Enter the name “HelloWorldController”
 - Tap **Add**



Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```

1 using Microsoft.Extensions.WebEncoders;
2
3 namespace MvcMovie.Controllers
4 {
5     public class HelloWorldController : Controller
6     {
7         //
8         // GET: /HelloWorld/
9
10        public string Index()
11        {
12            return "This is my default action...";
13        }
14
15        //
16        // GET: /HelloWorld/Welcome/
17
18        public string Welcome()
19        {
20            return "This is the Welcome action method...";
21        }
22    }
23 }
```

Every public method in a controller is callable. In the sample above, both methods return a string. Note the comments preceding each method:

```

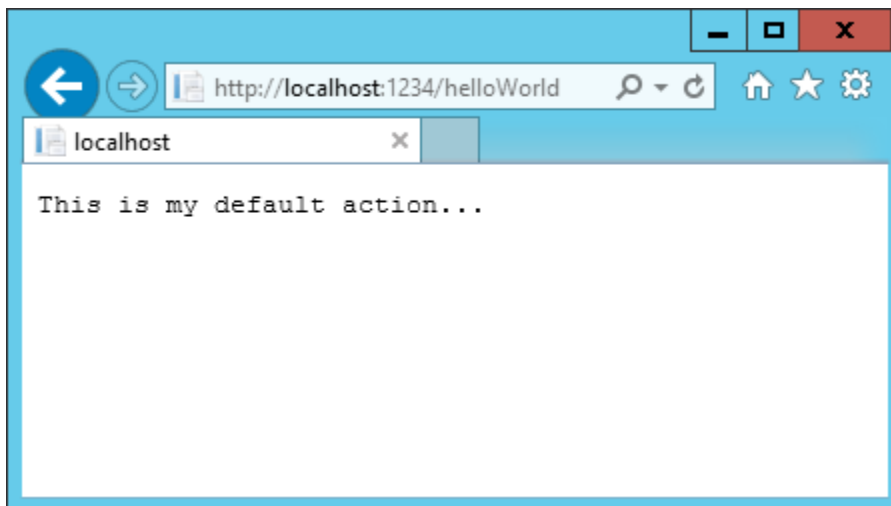
1 public class HelloWorldController : Controller
2 {
3     //
4     // GET: /HelloWorld/
5 }
```

```
6 public string Index()
7 {
8     return "This is my default action...";
9 }
10
11 //
12 // GET: /HelloWorld/Welcome/
13
14 public string Welcome()
15 {
16     return "This is the Welcome action method...";
17 }
18 }
```

The first comment states this is an **HTTP GET** method that is invoked by appending “/HelloWorld/” to the URL. The second comment specifies an **HTTP GET** method that is invoked by appending “/HelloWorld/Welcome/” to the URL. Later on in the tutorial we’ll use the scaffolding engine to generate **HTTP POST** methods.

Let’s test these methods with a browser.

Run the app in non-debug mode (press Ctrl+F5) and append “HelloWorld” to the path in the address bar. (In the image below, <http://localhost:1234/HelloWorld> is used, but you’ll have to replace *1234* with the port number of your app.) The `Index` method returns a string. You told the system to return some HTML, and it did!



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

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

You set the format for routing in the *Startup.cs* file.

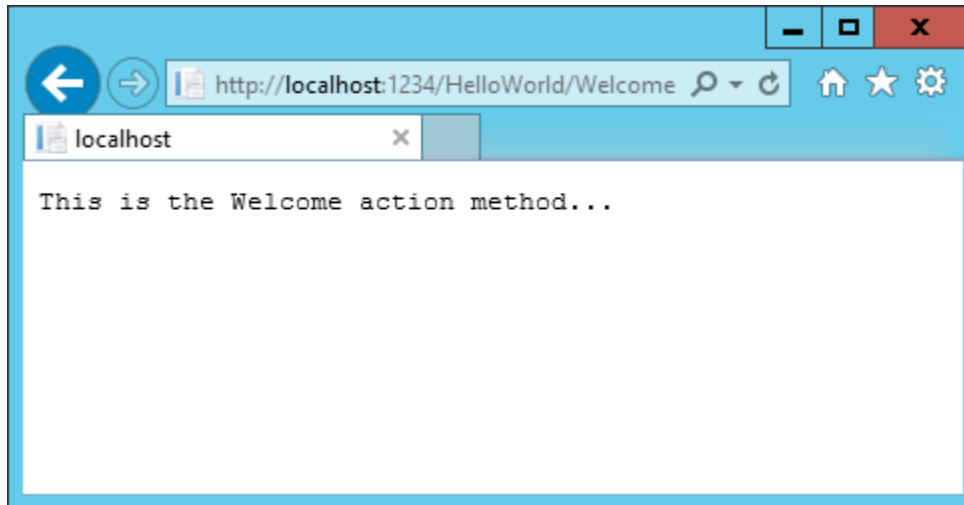
```
1 app.UseMvc(routes =>
2 {
3     routes.MapRoute(
4         name: "default",
5         template: "{controller=Home}/{action=Index}/{id?}");
6 });
```

When you run the app and don’t supply any URL segments, it defaults to the “Home” controller and the “Index” method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:xxxx/HelloWorld` maps

to the `HelloWorldController` class. The second part of the URL segment determines the action method on the class. So `localhost:xxxx/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that we only had to browse to `localhost:xxxx/HelloWorld` and the `Index` method was called by default. This is because `Index` is the default method that will be called on a controller if a method name is not explicitly specified. The third part of the URL segment (`Parameters`) is for route data. We'll see route data later on in this tutorial.

Browse to `http://localhost:xxxx/HelloWorld/Welcome`. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC routing is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, `/HelloWorld/Welcome?name=Scott&numtimes=4`). Change the `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.

```
1 public string Welcome(string name, int numTimes = 1)
2 {
3     return HtmlEncoder.Default.HtmlEncode(
4         "Hello " + name + ", NumTimes is: " + numTimes);
5 }
```

Note: The code above uses `HtmlEncoder.Default.HtmlEncode` to protect the app from malicious input (namely JavaScript).

Note: In Visual Studio 2015, when you are running without debugging (Ctrl+F5), you don't need to build the app after changing the code. Just save the file, refresh your browser and you can see the changes.

Run your app and browse to:

`http://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4`

(Replace `xxxx` with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See [Model Binding](#) for more information.

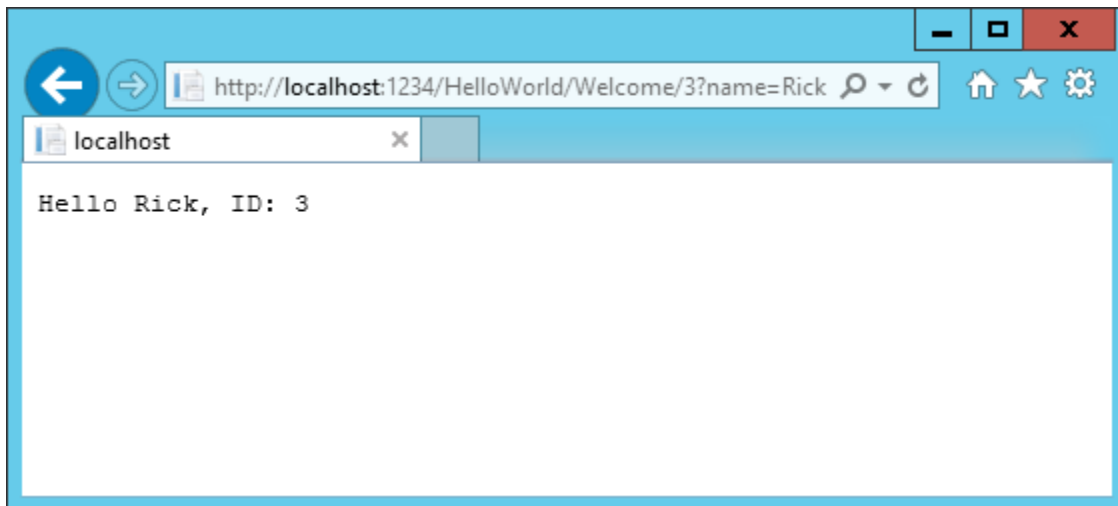


In the sample above, the URL segment (Parameters) is not used, the name and numTimes parameters are passed as query strings. The ? (question mark) in the above URL is a separator, and the query strings follow. The & character separates query strings.

Replace the Welcome method with the following code:

```
1 public string Welcome(string name, int ID = 1)
2 {
3     return HtmlEncoder.Default.HtmlEncode(
4         "Hello " + name + ", ID: " + ID);
5 }
```

Run the application and enter the following URL: `http://localhost:xxx/HelloWorld/Welcome/3?name=Rick`



This time the third URL segment matched the route parameter id. The Welcome method contains a parameter id that matched the URL template in the MapRoute method. The trailing ? (in id?) indicates the id parameter is optional.

```
1 app.UseMvc(routes =>
2 {
3     routes.MapRoute(
4         name: "default",
5         template: "{controller=Home}/{action=Index}/{id?}");
6 });
```

In these examples the controller has been doing the “VC” portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don’t want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead we’ll typically use a separate Razor view template file to help generate the HTML response. We’ll do that in the next tutorial.

2.1.3 Adding a view

By [Rick Anderson](#)

In this section you’re going to modify the `HelloWorldController` class to use Razor view template files to cleanly encapsulate the process of generating HTML responses to a client.

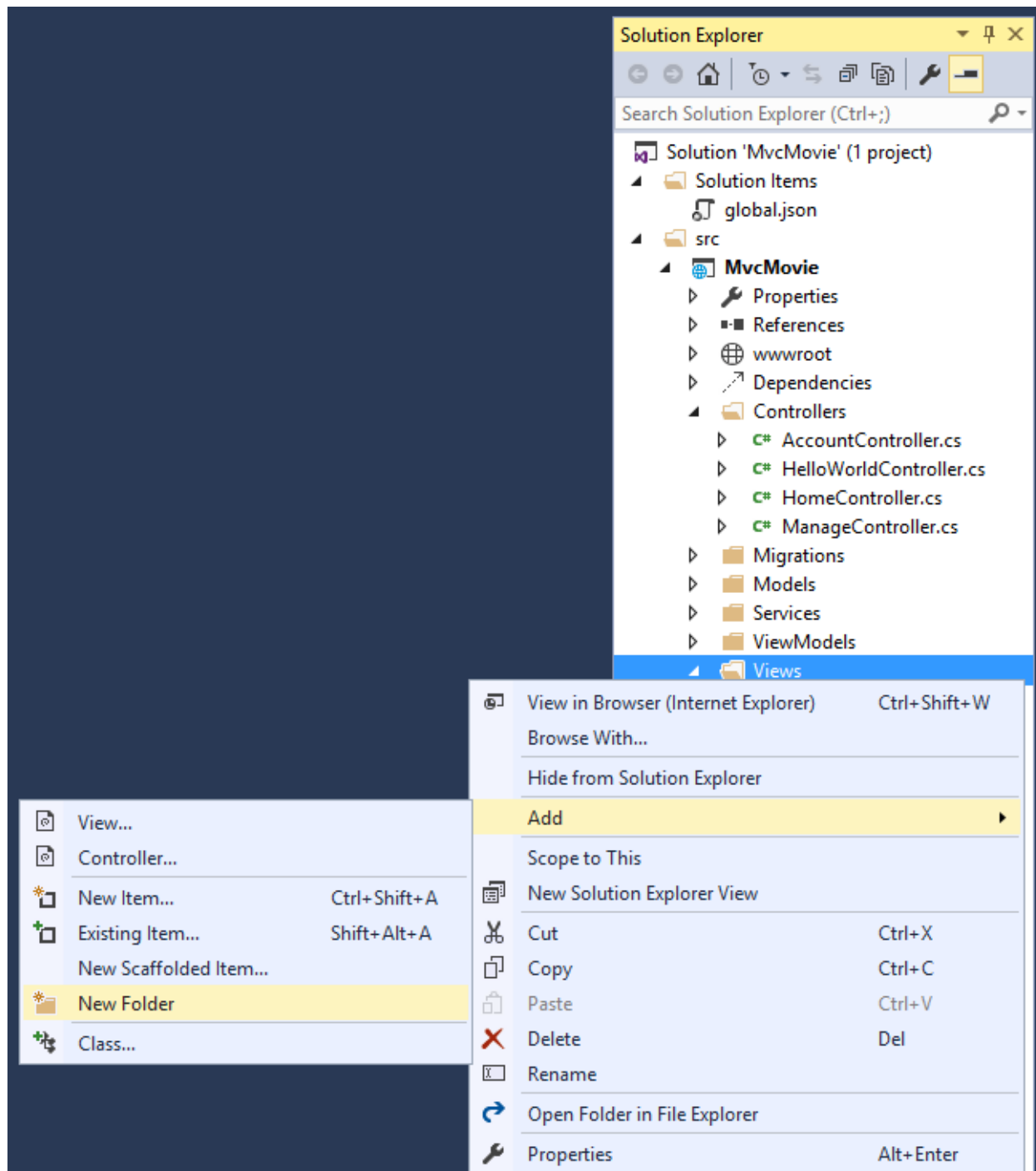
You’ll create a view template file using the Razor view engine. Razor-based view templates have a `.cshtml` file extension, and provide an elegant way to create HTML output using C#. Razor minimizes the number of characters and keystrokes required when writing a view template, and enables a fast, fluid coding workflow.

Currently the `Index` method returns a string with a message that is hard-coded in the controller class. Change the `Index` method to return a `View` object, as shown in the following code:

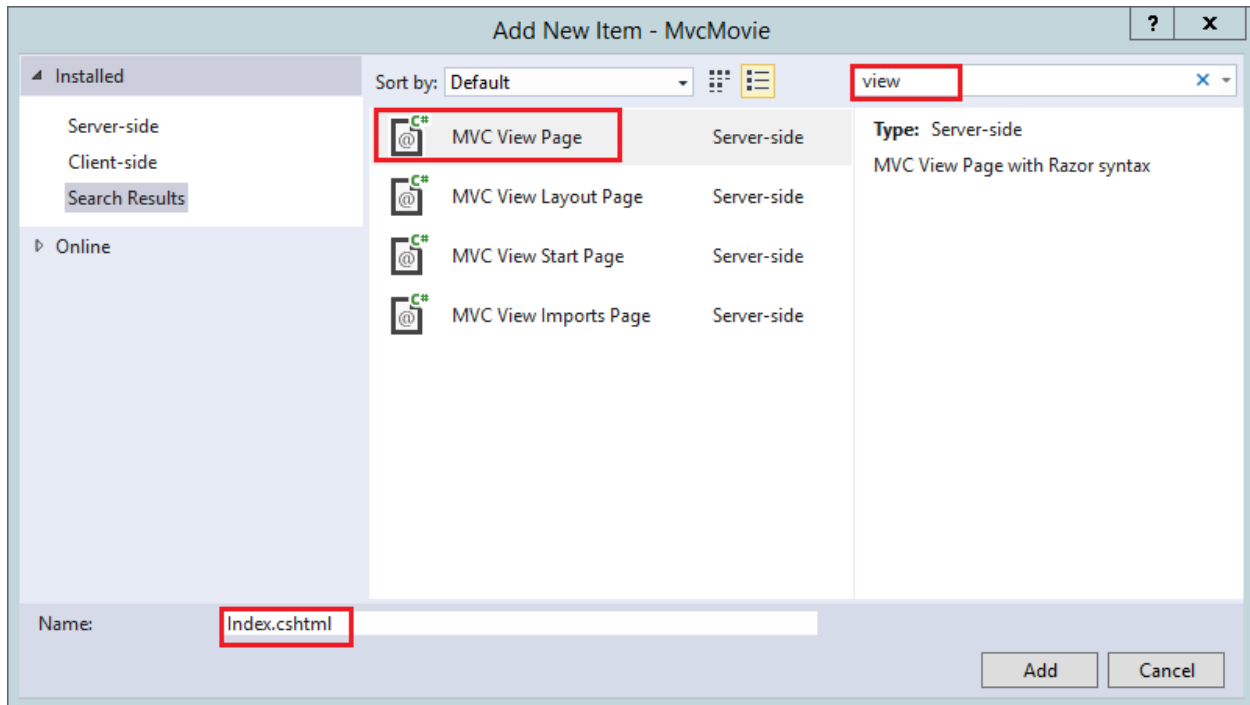
```
public IActionResult Index()
{
    return View();
}
```

The `Index` method above uses a view template to generate an HTML response to the browser. Controller methods (also known as [action methods](#)), such as the `Index` method above, generally return an `IActionResult` (or a class derived from `ActionResult`), not primitive types like `string`.

- Right click on the `Views` folder, and then **Add > New Folder** and name the folder *HelloWorld*.



- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - Movie** dialog
 - In the search box in the upper-right, enter *view*
 - Tap **MVC View Page**
 - In the **Name** box, keep the default *Index.cshtml*
 - Tap **Add**



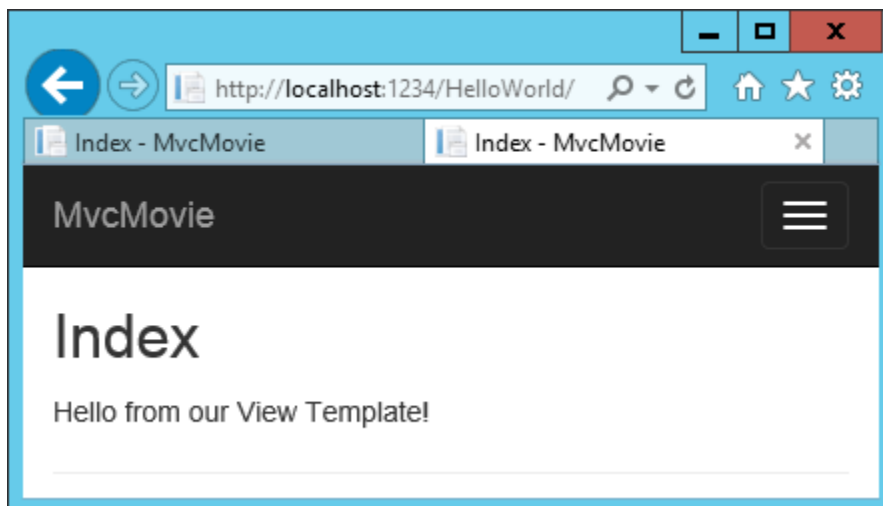
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

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

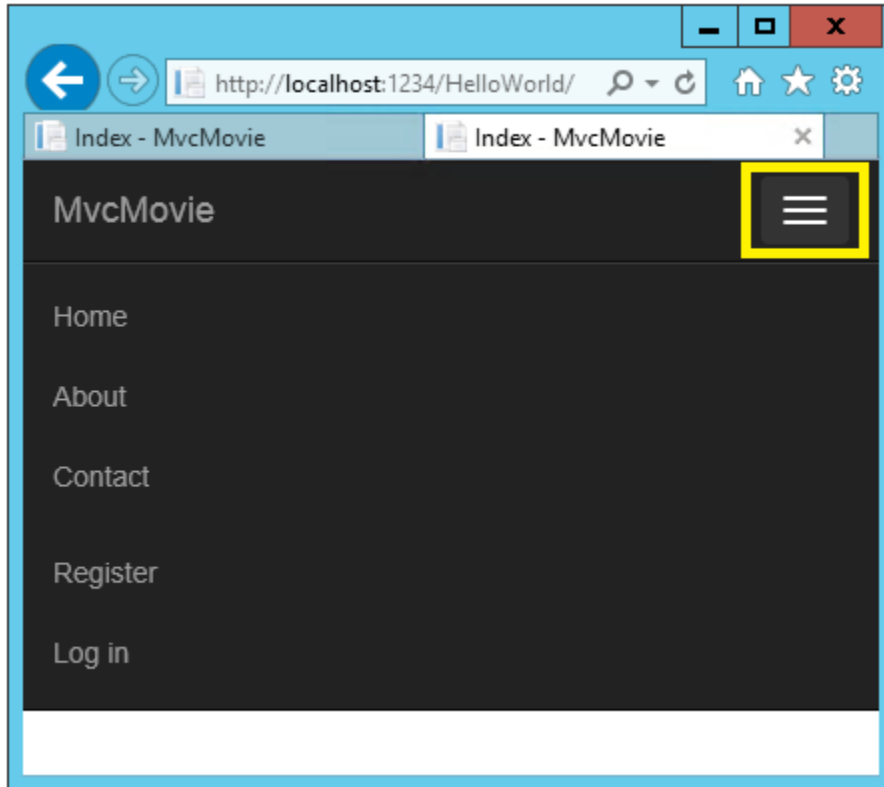
<h2>Index</h2>

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

Navigate to `http://localhost:xxxx/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much work; it simply ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, MVC defaulted to using the *Index.cshtml* view file in the */Views/HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



If your browser window is small (for example on a mobile device), you might need to toggle (tap) the [Bootstrap navigation button](#) in the upper right to see the [Home](#), [About](#), [Contact](#), [Register](#) and [Log in](#) links.



Changing views and layout pages

Tap on the menu links (**MvcMovie**, **Home**, **About**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, “wrapped” in the layout page. For example, if you select the **About** link, the *Views/Home/About.cshtml* view is rendered inside the `RenderBody` method.

Change the contents of the title element. Change the anchor text in the layout template to “MVC Movie” and the controller from `Home` to `Movies` as highlighted below:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>@ViewData["Title"] - Movie App</title>
7
8     <environment names="Development">
9       <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
10      <link rel="stylesheet" href="~/css/site.css" />
11    </environment>
12    <environment names="Staging,Production">
13      <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/css/bootstrap

```

```

14         asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
15         asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallba
16         <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
17     </environment>
18 </head>
19 <body>
20     <div class="navbar navbar-inverse navbar-fixed-top">
21         <div class="container">
22             <div class="navbar-header">
23                 <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="
24                     <span class="sr-only">Toggle navigation</span>
25                     <span class="icon-bar"></span>
26                     <span class="icon-bar"></span>
27                     <span class="icon-bar"></span>
28                 </button>
29                 <a asp-controller="Movies" asp-action="Index" class="navbar-brand">Mvc Movie</a>
30             </div>
31             <div class="navbar-collapse collapse">
32                 <ul class="nav navbar-nav">
33                     <li><a asp-controller="Home" asp-action="Index">Home</a></li>
34                     <li><a asp-controller="Home" asp-action="About">About</a></li>
35                     <li><a asp-controller="Home" asp-action="Contact">Contact</a></li>
36                 </ul>
37                 @await Html.PartialAsync("_LoginPartial")
38             </div>
39         </div>
40     </div>
41     <div class="container body-content">
42         @RenderBody()
43         <hr />
44         <footer>
45             <p>&copy; 2015 - MvcMovie</p>
46         </footer>
47     </div>
48
49     <environment names="Development">
50         <script src="~/lib/jquery/dist/jquery.js"></script>
51         <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
52         <script src="~/js/site.js" asp-append-version="true"></script>
53     </environment>
54     <environment names="Staging,Production">
55         <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
56             asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
57             asp-fallback-test="window.jQuery">
58         </script>
59         <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.5/bootstrap.min.js"
60             asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
61             asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
62         </script>
63         <script src="~/js/site.min.js" asp-append-version="true"></script>
64     </environment>
65
66     @RenderSection("scripts", required: false)
67 </body>
68 </html>

```

Note: We haven't implemented the `Movies` controller yet, so if you click on that link, you'll get an error.

Save your changes and tap the **About** link. Notice how each page displays the **Mvc Movie** link. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

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

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. You can use the `Layout` property to set a different layout view, or set it to `null` so no layout file will be used.

Now, let's change the title of the `Index` view.

Open *Views/HelloWorld/Index.cshtml*. There are two places to make a change:

- The text that appears in the title of the browser
- The secondary header (`<h2>` element).

You'll make them slightly different so you can see which bit of code changes which part of the app.

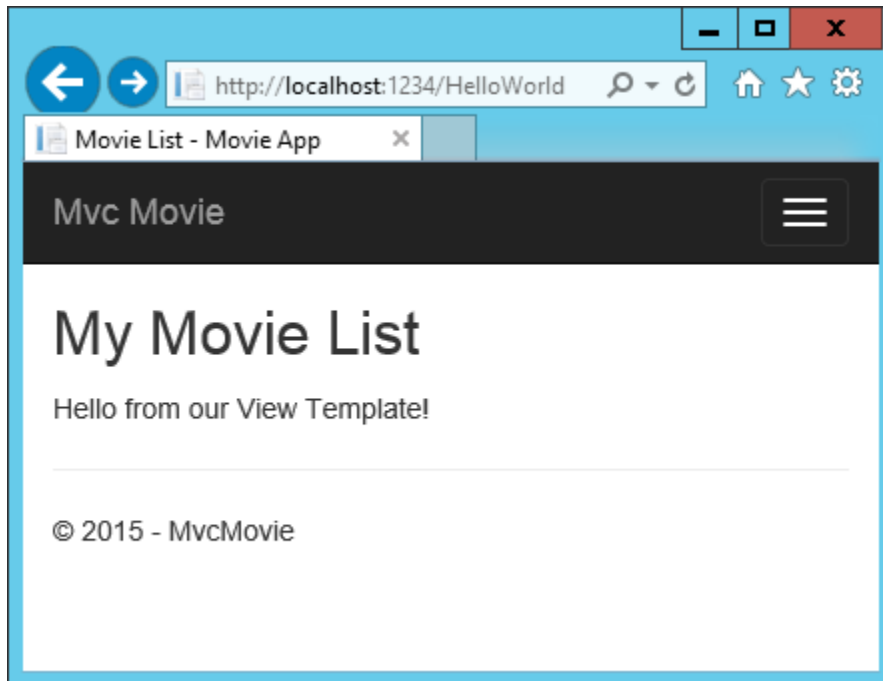
```
1  @{
2      ViewData["Title"] = "Movie List";
3  }
4
5  <h2>My Movie List</h2>
6
7  <p>Hello from our View Template!</p>
```

Line number 2 in the code above sets the `Title` property of the `ViewDataDictionary` to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save your change and refresh the page. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press `Ctrl+F5` in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the **Index.cshtml** view template and the additional " - Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *Views/Shared/_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.



Our little bit of “data” (in this case the “Hello from our View Template!” message) is hard-coded, though. The MVC application has a “V” (view) and you’ve got a “C” (controller), but no “M” (model) yet. Shortly, we’ll walk through how create a database and retrieve model data from it.

Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let’s first talk about passing information from the controller to a view. Controller classes are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming browser requests, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: A view template should never perform business logic or interact with a database directly. Instead, a view template should work only with the data that’s provided to it by the controller. Maintaining this “separation of concerns” helps keep your code clean, testable and more maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `numTimes` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let’s change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

Return to the `HelloWorldController.cs` file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means you can put whatever you want in to it; the `ViewData` object has no defined properties until you put something inside it. The [MVC model binding system](#) automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete `HelloWorldController.cs` file looks like this:

```
using Microsoft.AspNet.Mvc;

namespace MvcMovie.Controllers
```

```
{  
    public class HelloWorldController : Controller  
    {  
        public IActionResult Index()  
        {  
            return View();  
        }  
  
        public IActionResult Welcome(string name, int numTimes = 1)  
        {  
            ViewData["Message"] = "Hello " + name;  
            ViewData["NumTimes"] = numTimes;  
  
            return View();  
        }  
    }  
}
```

The ViewData dictionary object contains data that will be passed to the view. Next, you need a Welcome view template.

- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - Movie** dialog
 - In the search box in the upper-right, enter *view*
 - Tap **MVC View Page**
 - In the **Name** box, enter *Welcome.cshtml*
 - Tap **Add**

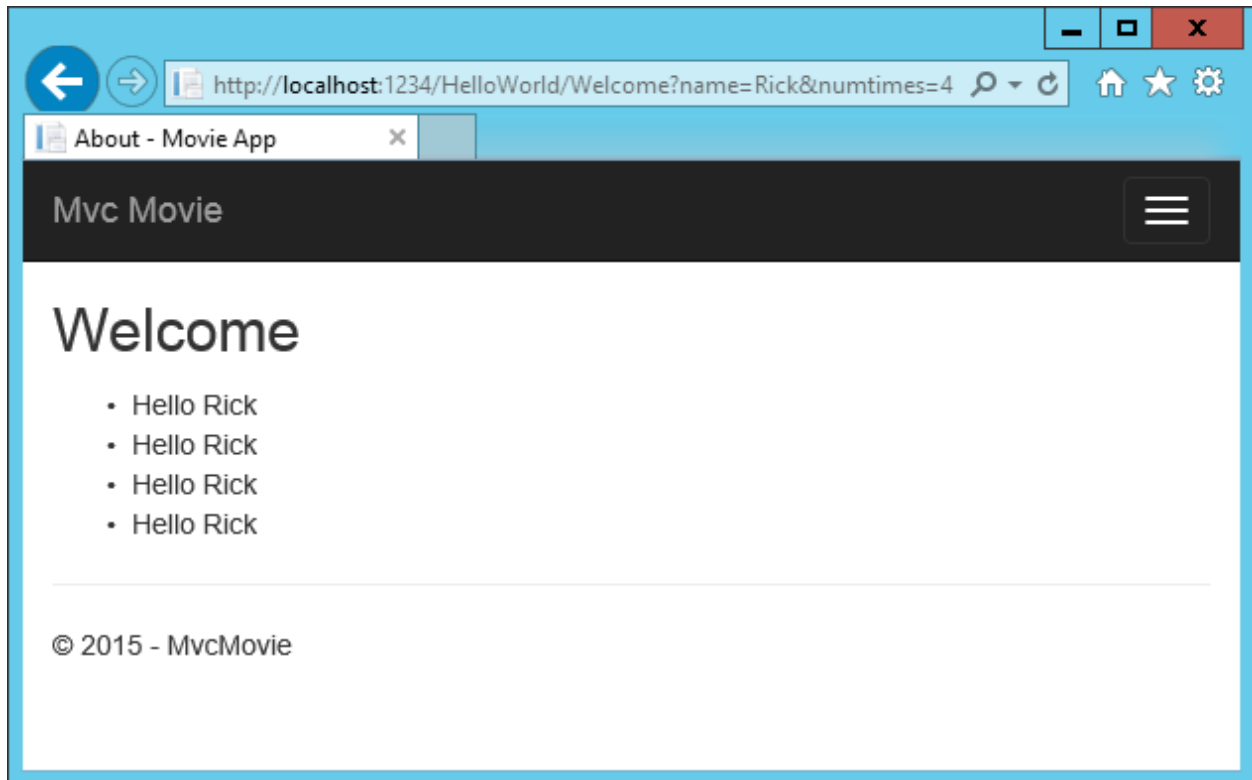
You'll create a loop in the *Welcome.cshtml* view template that displays “Hello” NumTimes. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{  
    ViewData["Title"] = "About";  
}  
  
<h2>Welcome</h2>  
  
<ul>  
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)  
    {  
        <li>@ViewData["Message"]</li>  
    }  
</ul>
```

Save your changes and browse to the following URL:

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

Data is taken from the URL and passed to the controller using the **model binder**. The controller packages the data into a ViewData dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, we used the `ViewData` dictionary to pass data from the controller to a view. Later in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See [Dynamic V Strongly Typed Views](#) for more information.

Well, that was a kind of an “M” for model, but not the database kind. Let’s take what we’ve learned and create a database of movies.

2.1.4 Adding a model

By [Rick Anderson](#)

In this section you’ll add some classes for managing movies in a database. These classes will be the “**Model**” part of the MVC app.

You’ll use a .NET Framework data-access technology known as the [Entity Framework](#) to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from “plain-old CLR objects.”) You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow. If you are required to create the database first, you can still follow this tutorial to learn about MVC and EF app development.

Adding Model Classes

In Solution Explorer, right click the *Models* folder > **Add** > **Class**.

```
1 using System;
2
3 public class Movie
```

```

4 {
5     public int ID { get; set; }
6     public string Title { get; set; }
7     public DateTime ReleaseDate { get; set; }
8     public string Genre { get; set; }
9     public decimal Price { get; set; }
10 }

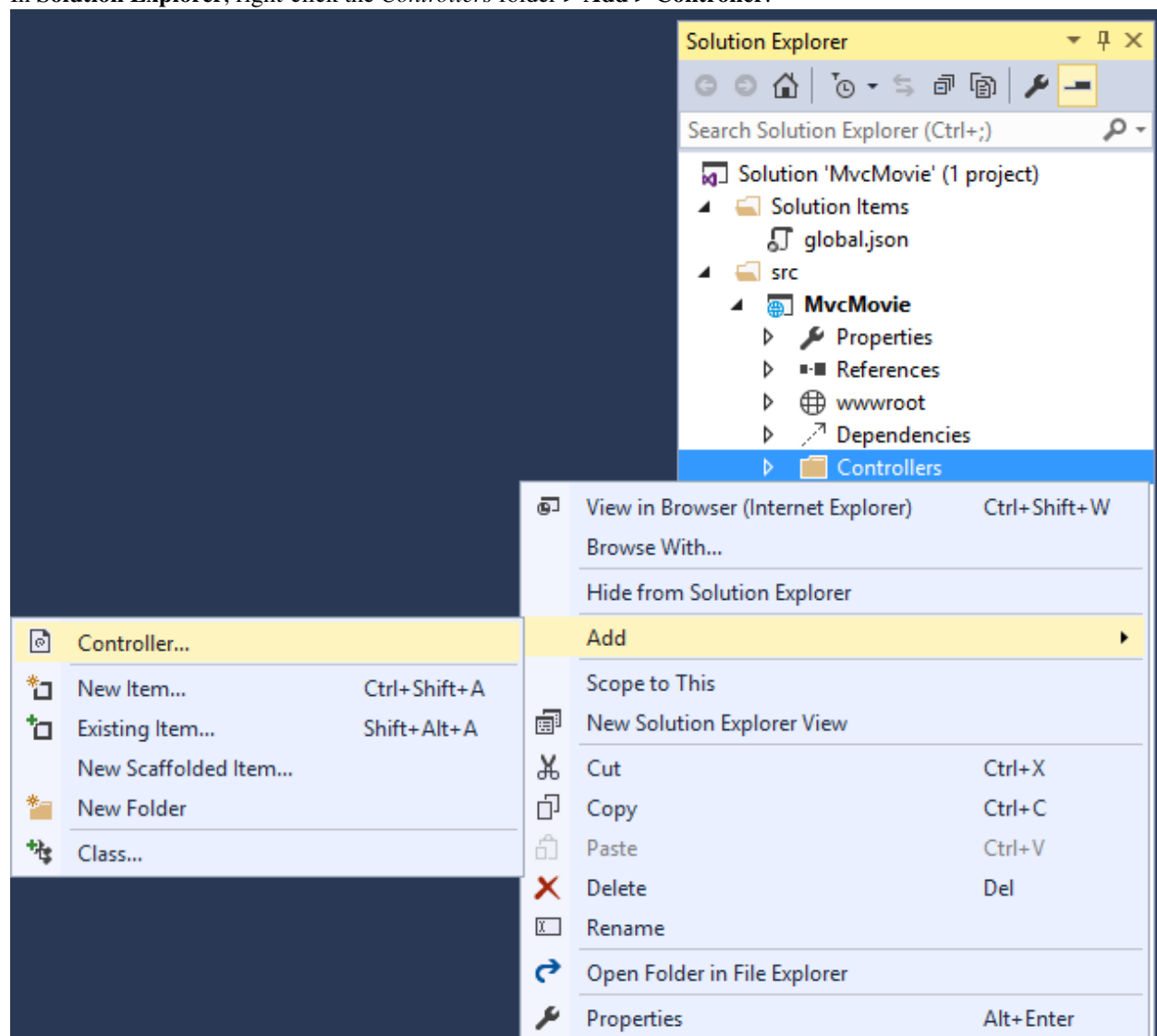
```

In addition to the properties you'd expect to model a movie, the `ID` field is required by the DB for the primary key.

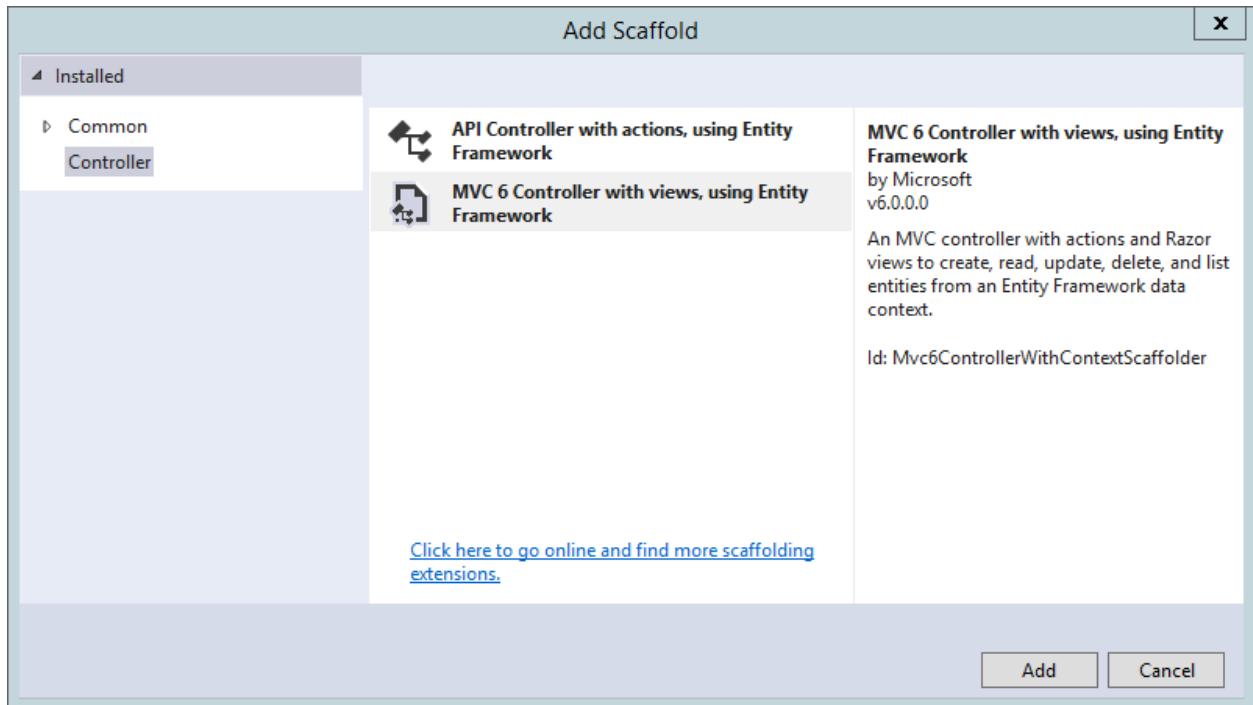
Build the project. If you don't build the app, you'll get an error in the next section. We've finally added a **Model** to our MVC app.

Scaffolding a controller

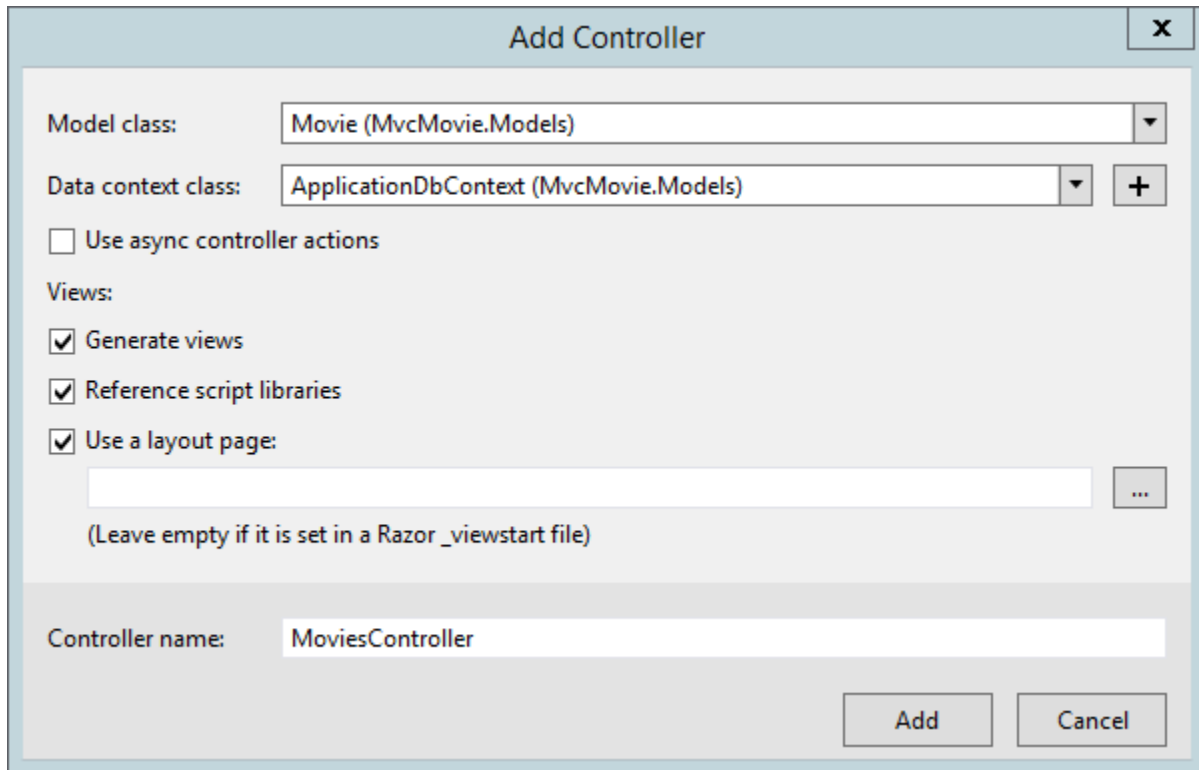
In **Solution Explorer**, right-click the *Controllers* folder > **Add > Controller**.



In the **Add Scaffold** dialog, tap **MVC 6 Controller with views, using Entity Framework > Add**.



- Complete the **Add Controller** dialog
 - **Model class:** *Movie(MvcMovie.Models)*
 - **Data context class:** *ApplicationDbContext(MvcMovie.Models)*
 - **Controller name:** Keep the default *MoviesController*
 - **Views::** Keep the default of each option checked
 - **Controller name:** Keep the default *MoviesController*
 - Tap **Add**



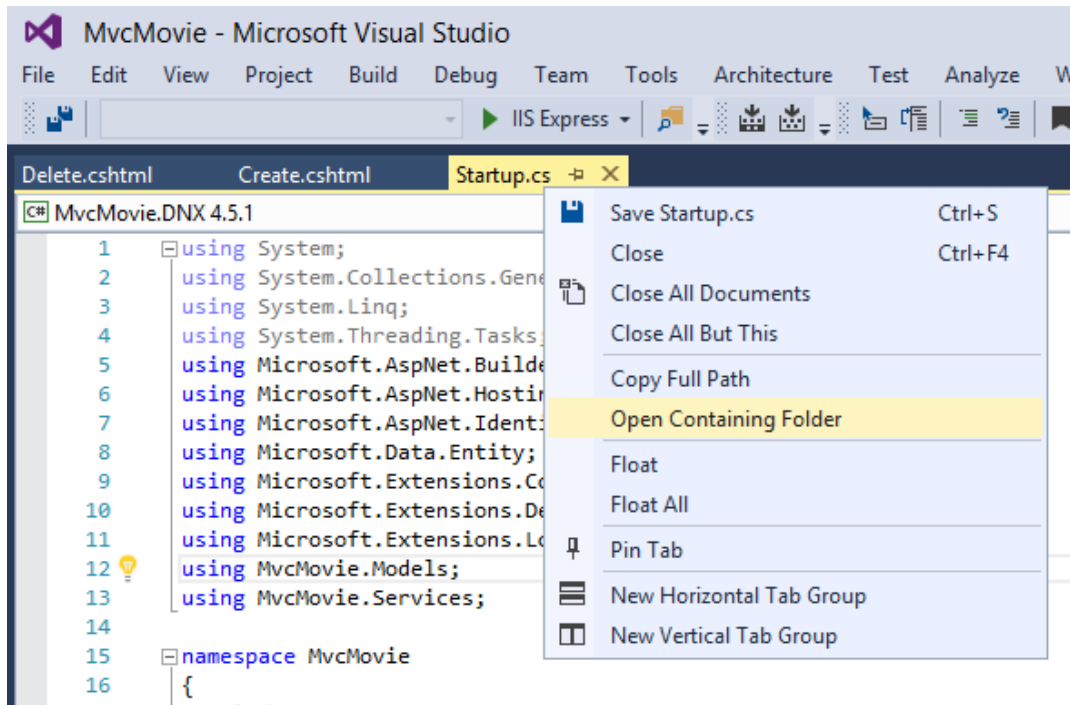
The Visual Studio scaffolding engine creates the following:

- A movies controller (MoviesController.c)
- Create, Delete, Details, Edit and Index Razor view files
- Migrations classes
 - The `CreateIdentitySchema` class creates the [ASP.NET Identity membership database](#) tables. The Identity database stores user login information that is needed for authentication. We won't cover authentication in this tutorial, for that you can follow [Additional resources](#) at the end of this tutorial.
 - The `ApplicationDbContextModelSnapshot` class creates the EF entities used to access the Identity database. We'll talk more about EF entities later in the tutorial.

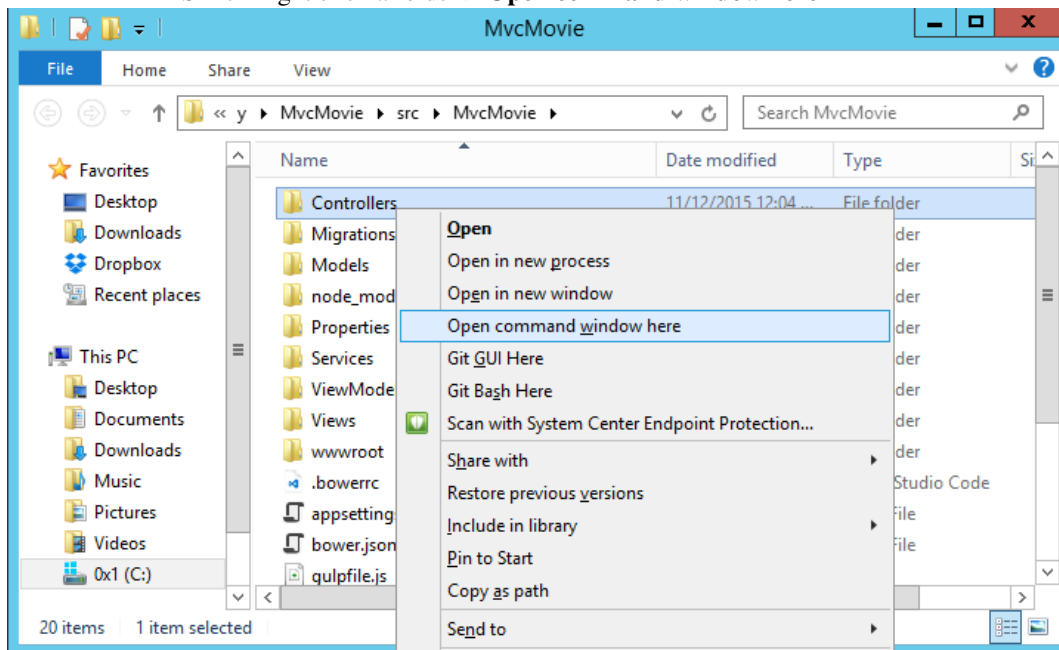
Visual Studio automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as *scaffolding*). You'll soon have a fully functional web application that lets you create, list, edit, and delete movie entries.

Use data migrations to create the database

- Open a command prompt in the project directory (MvcMovie/src/MvcMovie). Follow these instructions for a quick way to open a folder in the project directory.
 - Open a file in the root of the project (for this example, use *Startup.cs*.)
 - Right click on *Startup.cs* > **Open Containing Folder**.



– Shift + right click a folder > **Open command window here**



– Run `cd ..` to move back up to the project directory

- Run the following commands in the command prompt:

```
dnu restore
dnvm use 1.0.0-rc1-update1 -p
dnx ef migrations add Initial
dnx ef database update
```

```

C:\Windows\system32\cmd.exe

C:\y\MvcMovie\src\MvcMovie>dnu restore
Microsoft .NET Development Utility Clr-x86-1.0.0-rc1-16147

Restoring packages for C:\y\MvcMovie\src\MvcMovie\project.json
Writing lock file C:\y\MvcMovie\src\MvcMovie\project.lock.json
Restore complete, 8152ms elapsed

NuGet Config files used:
  C:\Users\riande\AppData\Roaming\NuGet\nuget.config

C:\y\MvcMovie\src\MvcMovie>dnvm use 1.0.0-rc1-final -p
Adding C:\Users\riande\.dnx\runtimes\dnx-clr-win-x86.1.0.0-rc1-final\bin to process PATH
Adding C:\Users\riande\.dnx\runtimes\dnx-clr-win-x86.1.0.0-rc1-final\bin to user PATH

C:\y\MvcMovie\src\MvcMovie>dnx ef migrations add Initial
An operation was scaffolded that may result in the loss of data. Please review the migration
Done. To undo this action, use 'ef migrations remove'

C:\y\MvcMovie\src\MvcMovie>dnx ef database update
Applying migration '00000000000000000000000000000000_CreateIdentitySchema'.
Applying migration '20151112220059_Initial'.
Done.

C:\y\MvcMovie\src\MvcMovie>

```

- `dnu restore` This command looks at the dependencies in the *project.json* file and downloads them. For more information see [Working with DNX Projects](#) and [DNX Overview](#).
- `dnvm use <version>` **dnvm** is the .NET Version Manager, which is a set of command line utilities that are used to update and configure .NET Runtime. In this case we're asking **dnvm** add the 1.0.0-rc1 ASP.NET 5 runtime to the PATH environment variable of the current shell.
- `dnx` DNX stands for .NET Execution Environment.
 - `dnx ef` The `ef` command is specified in the *project.json* file:

```

1  "commands": {
2    "web": "Microsoft.AspNet.Server.Kestrel",
3    "ef": "EntityFramework.Commands"
4  },

```

- `dnx ef migrations add Initial` Creates a class named `Initial`

```
public partial class Initial : Migration
```

The parameter “Initial” is arbitrary, but customary for the first (*initial*) database migration. You can safely ignore the warning may result in the loss of data, it is dropping foreign key constraints and not any data. The warning is a result of the initial create migration for the `Identity` model not being up-to-date. This will be fixed in the next version.

- `dnx ef database update` Updates the database, that is, applies the migrations.

Test the app

- Run the app and tap the **Mvc Movie** link

- Tap the **Create New** link and create a movie

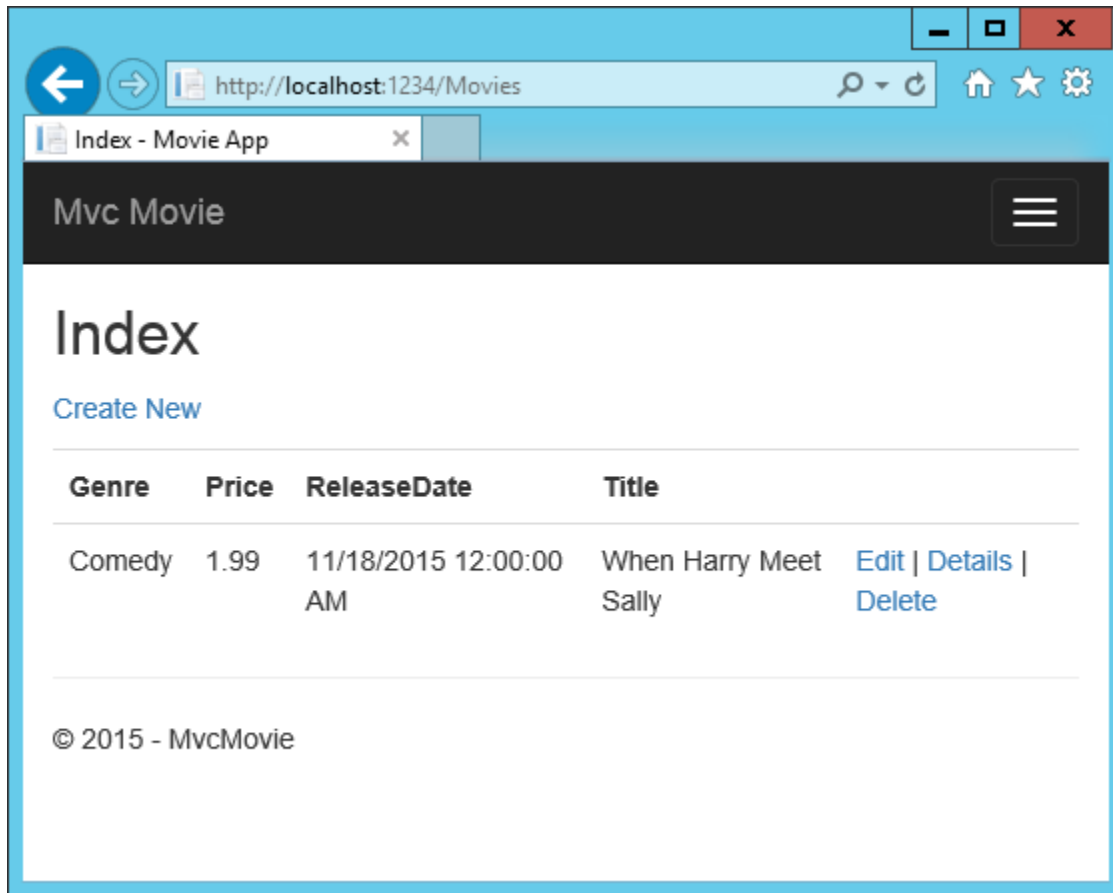
The screenshot shows a mobile browser window with the address bar displaying `http://localhost:1234/Movies`. The browser has a single tab titled 'Create - Movie App'. The app's header is 'Mvc Movie' with a hamburger menu icon. The main content area is titled 'Create Movie' and contains a form with the following fields:

- Genre**: A text input field containing 'Comedy'.
- Price**: A text input field containing '1.99'.
- ReleaseDate**: A text input field containing '11-18-15'.
- Title**: A text input field containing 'When Harry Meet Sally'.

Below the form fields is a 'Create' button. At the bottom of the form area is a link labeled 'Back to List'. The footer of the app displays '© 2015 - MvcMovie'.

Note: You may not be able to enter decimal points or commas in the `Price` field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See [Additional resources](#) for more information. For now, just enter whole numbers like 10.

Tapping **Create** causes the form to be posted to the server, where the movie information is saved in a database. You are then redirected to the `/Movies` URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

Examining the Generated Code

Open the *Controllers/MoviesController.cs* file and examine the generated `Index` method. A portion of the movie controller with the `Index` method is shown below:

```
public class MoviesController : Controller
{
    private ApplicationDbContext _context;

    public MoviesController(ApplicationDbContext context)
    {
        _context = context;
    }

    public IActionResult Index() {
        return View(_context.Movie.ToList());
    }
}
```

The constructor uses [Dependency Injection](#) to inject the database context into the controller. The database context is used in each of the [CRUD](#) methods in the controller.

A request to the `Movies` controller returns all the entries in the `Movies` table and then passes the data to the `Index` view.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer *IntelliSense* in the Visual Studio editor. The scaffolding mechanism in Visual Studio used this approach (that is, passing a strongly typed model) with the `MoviesController` class and view templates when it created the methods and views.

Examine the generated `Details` method in the `Controllers/MoviesController.cs` file. The `Details` method is shown below.

```
// GET: Movies/Details/5
public IActionResult Details(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }

    Movie movie = _context.Movie.Single(m => m.ID == id);
    if (movie == null)
    {
        return HttpNotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data, for example `http://localhost:1234/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment)
- The action to `details` (the second URL segment)
- The `id` to `1` (the last URL segment)

You could also pass in the `id` with a query string as follows:

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

If a `Movie` is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the `Views/Movies/Details.cshtml` file:

```
@model MvcMovie.Models.Movie

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

<h2>Details</h2>

<div>
    <h4>Movie</h4>
    <hr />
```

```

<dl class="dl-horizontal">
  <dt>
    @Html.DisplayNameFor(model => model.Genre)
  </dt>
  <dd>
    @Html.DisplayFor(model => model.Genre)
  </dd>
  <dt>
    @Html.DisplayNameFor(model => model.Price)
  </dt>
  <dd>
    @Html.DisplayFor(model => model.Price)
  </dd>
  <dt>
    @Html.DisplayNameFor(model => model.ReleaseDate)
  </dt>
  <dd>
    @Html.DisplayFor(model => model.ReleaseDate)
  </dd>
  <dt>
    @Html.DisplayNameFor(model => model.Title)
  </dt>
  <dd>
    @Html.DisplayFor(model => model.Title)
  </dd>
</dl>
</div>
<p>
  <a asp-action="Edit" asp-route-id="@Model.ID">Edit</a> |
  <a asp-action="Index">Back to List</a>
</p>

```

By including a `@model` statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Details.cshtml* file:

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

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Details.cshtml* template, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and view templates also pass a `Movie` model object.

Examine the *Index.cshtml* view template and the `Index` method in the `Movies` controller. Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the `Index` action method to the view:

```

public IActionResult Index()
{
    return View(_context.Movie.ToList());
}

```

When you created the movies controller, Visual Studio automatically included the following `@model` statement at the top of the *Index.cshtml* file:

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

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* template, the code loops through the movies with a

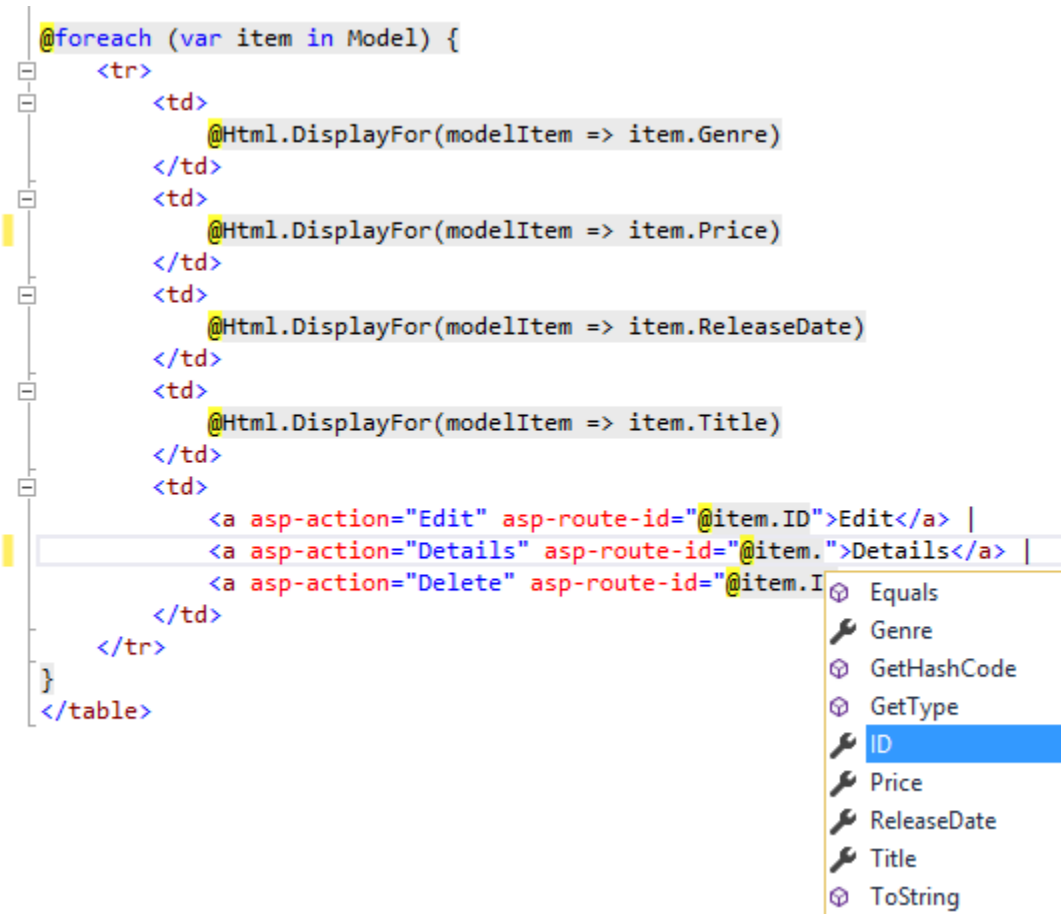
foreach statement over the strongly typed Model object:

```

1 @model IEnumerable<MvcMovie.Models.Movie>
2
3 @{
4     ViewData["Title"] = "Index";
5 }
6
7 <h2>Index</h2>
8
9 <p>
10     <a asp-action="Create">Create New</a>
11 </p>
12 <table class="table">
13     <tr>
14         <th>
15             @Html.DisplayNameFor(model => model.Genre)
16         </th>
17         <th>
18             @Html.DisplayNameFor(model => model.Price)
19         </th>
20         <th>
21             @Html.DisplayNameFor(model => model.ReleaseDate)
22         </th>
23         <th>
24             @Html.DisplayNameFor(model => model.Title)
25         </th>
26         <th></th>
27     </tr>
28
29     @foreach (var item in Model) {
30         <tr>
31             <td>
32                 @Html.DisplayFor(modelItem => item.Genre)
33             </td>
34             <td>
35                 @Html.DisplayFor(modelItem => item.Price)
36             </td>
37             <td>
38                 @Html.DisplayFor(modelItem => item.ReleaseDate)
39             </td>
40             <td>
41                 @Html.DisplayFor(modelItem => item.Title)
42             </td>
43             <td>
44                 <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
45                 <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
46                 <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
47             </td>
48         </tr>
49     }
50 </table>

```

Because the Model object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile-time checking of the code and full *IntelliSense* support in the code editor:



Note: The RC1 version of the scaffolding engine generates HTML Helpers to display fields (`@Html.DisplayNameFor(model => model.Genre)`). The next version will use [Tag Helpers](#) to render fields.

You now have a database and pages to display, edit, update and delete data. In the next tutorial, we'll work with the database.

Additional resources

- [Tag Helpers](#)
- [Create a secure ASP.NET MVC app and deploy to Azure](#)
- [Working with DNX Projects](#)
- [DNX Overview](#)

2.1.5 Working with SQL Server LocalDB

By [Rick Anderson](#)

The `ApplicationDbContext` class handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the [dependency injection](#) container in the `ConfigureServices` method in the `Startup.cs` file:

```

1 public void ConfigureServices(IServiceCollection services)
2 {
3     // Add framework services.
4     services.AddEntityFramework()
5         .AddSqlServer()
6         .AddDbContext<ApplicationDbContext>(options =>
7             options.UseSqlServer(Configuration["Data:DefaultConnection:ConnectionString"]));

```

The ASP.NET 5 [Configuration](#) system reads the `Data:DefaultConnection:ConnectionString`. For local development, it gets the connection string from the `appsettings.json` file:

```

1 {
2     "Data": {
3         "DefaultConnection": {
4             "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=aspnet5-MvcMovie-53e157ca-bf3b-461
5         }
6     },

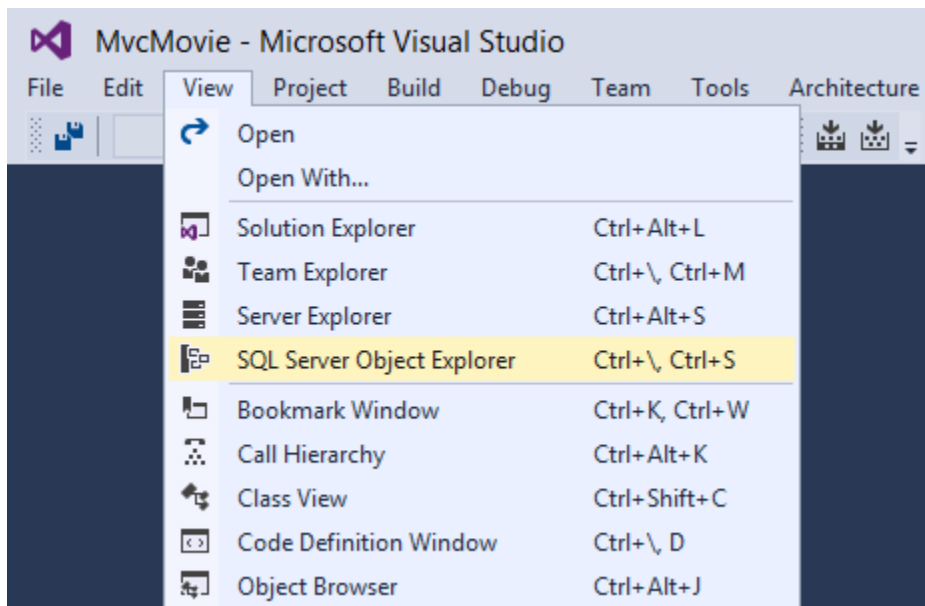
```

When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See [Configuration](#).

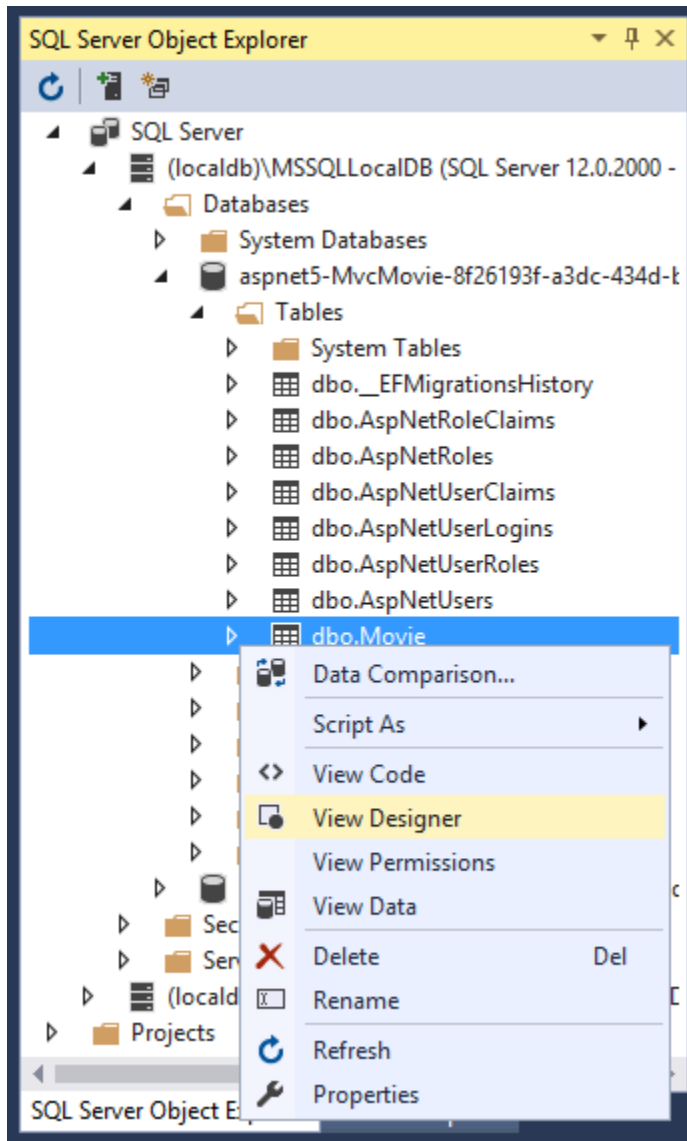
SQL Server Express LocalDB

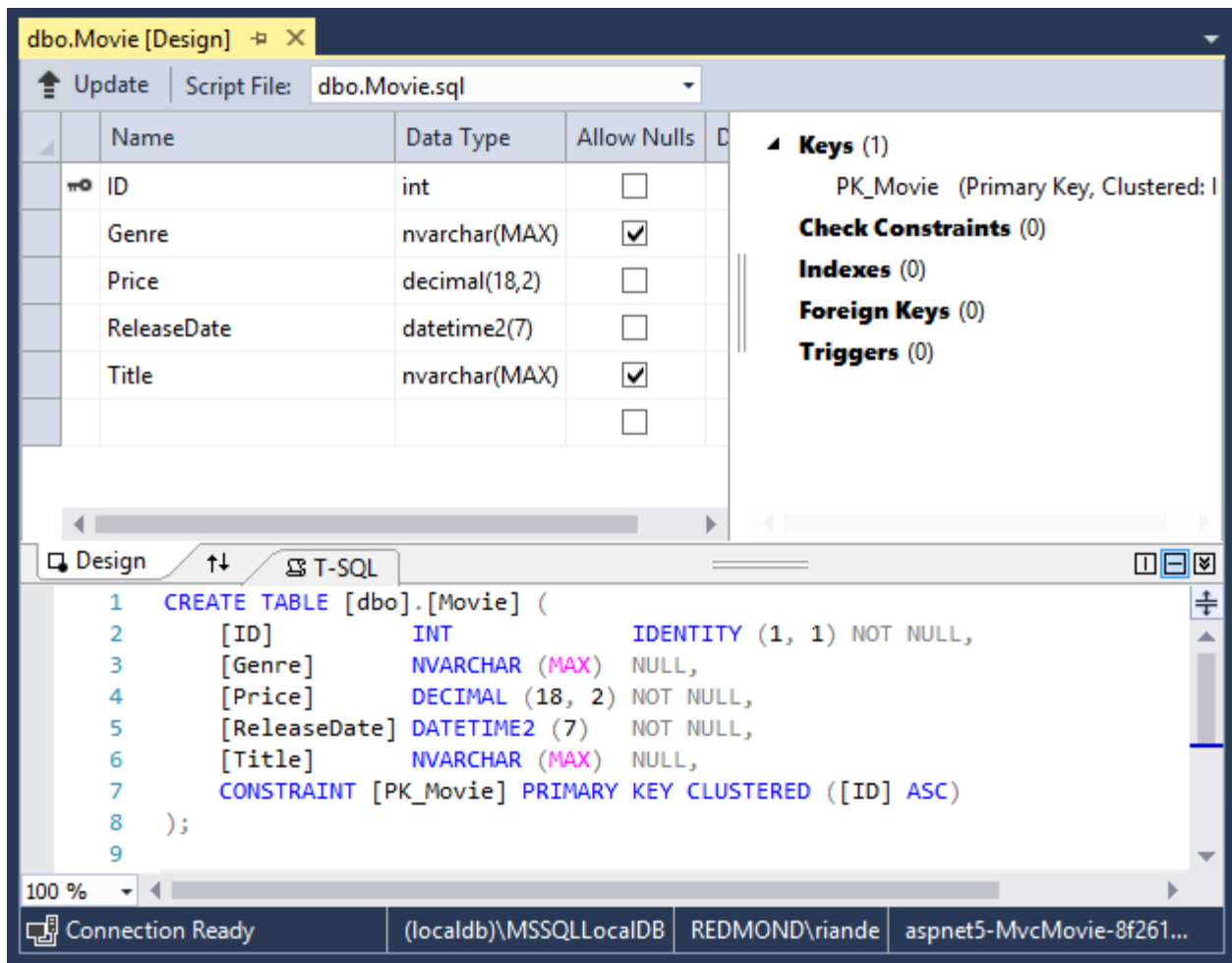
LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates “*.mdf” files in the `C:/Users/<user>` directory.

- From the **View** menu, open **SQL Server Object Explorer (SSOX)**.



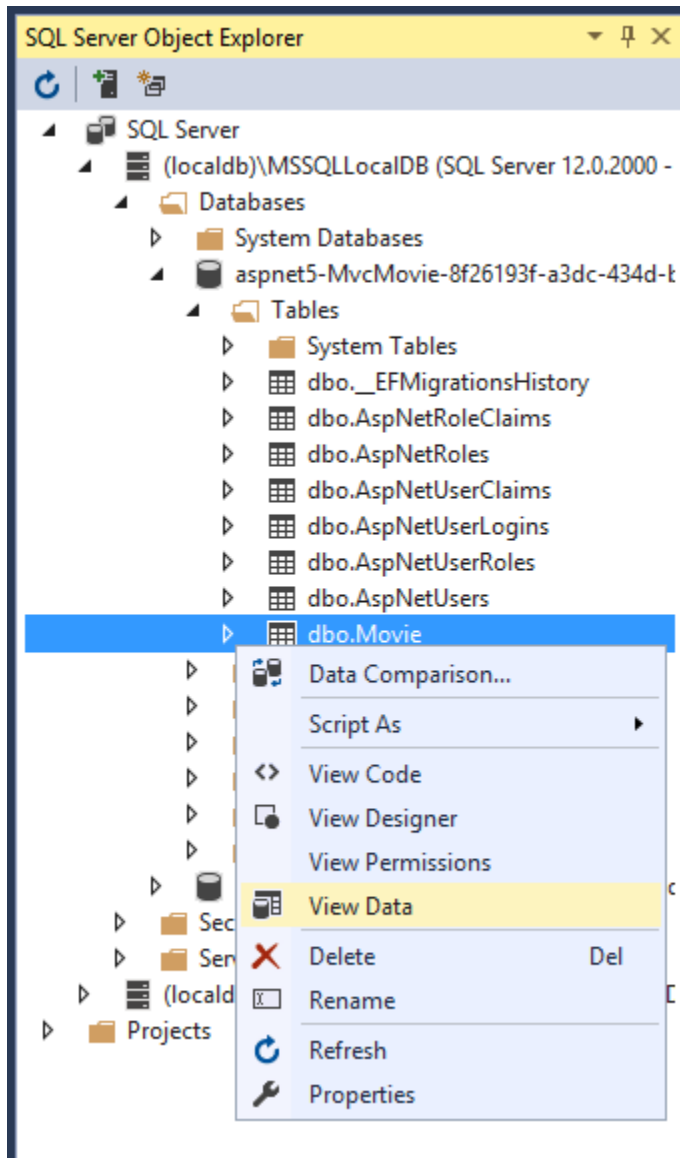
- Right click on the `Movie` table > **View Designer**

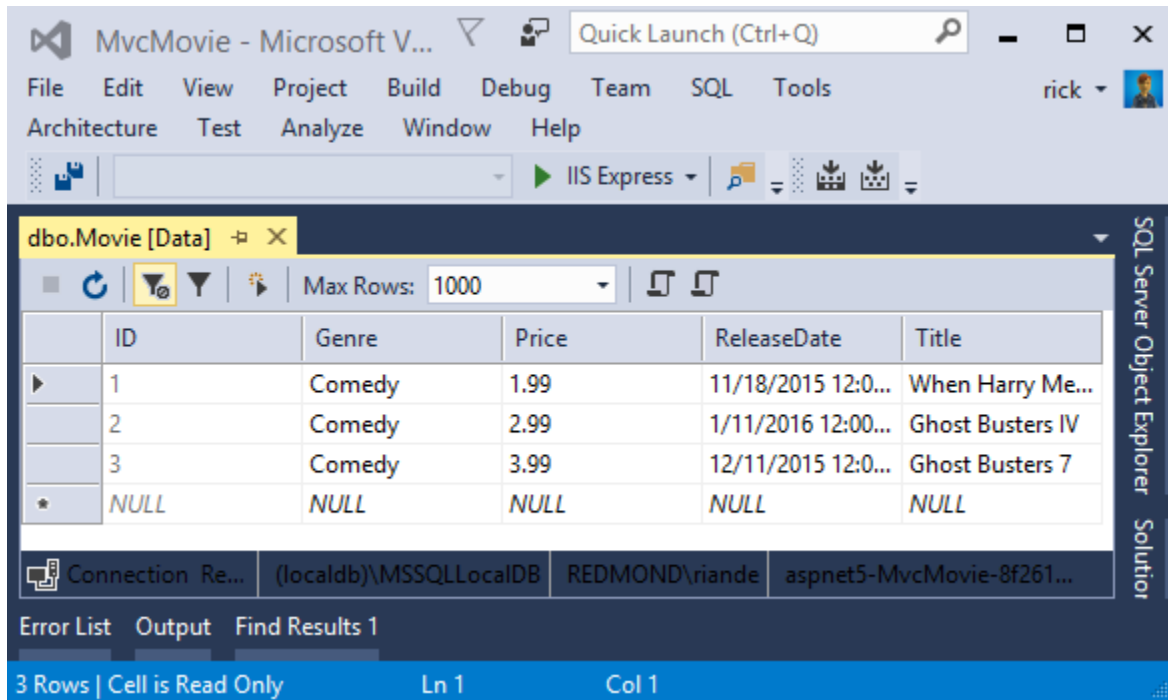




Note the key icon next to ID. By default, EF will make a property named ID the primary key.

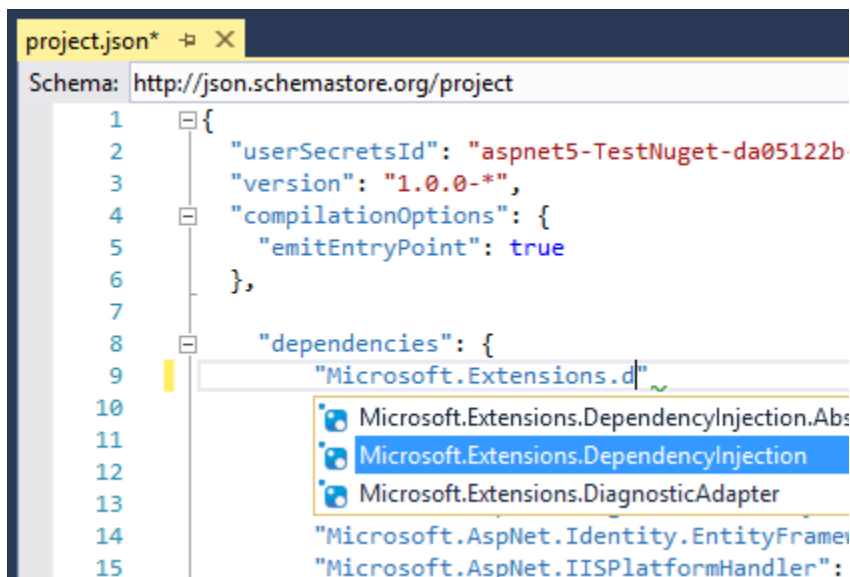
- Right click on the `Movie` table > **View Data**





Seed the database

We'll take advantage of [Dependency Injection \(DI\)](#) to seed the database. You add server side dependencies to ASP.NET 5 projects in the `project.json` file. Open `project.json` and add the Microsoft DI package. IntelliSense helps us add the package.



The DI package is highlighted below:

```

{
  "userSecretsId": "aspnet5-MvcMovie-53e157ca-bf3b-46b7-bb3f-82ac58612f5e",
  "version": "1.0.0-*",
  "compilationOptions": {

```

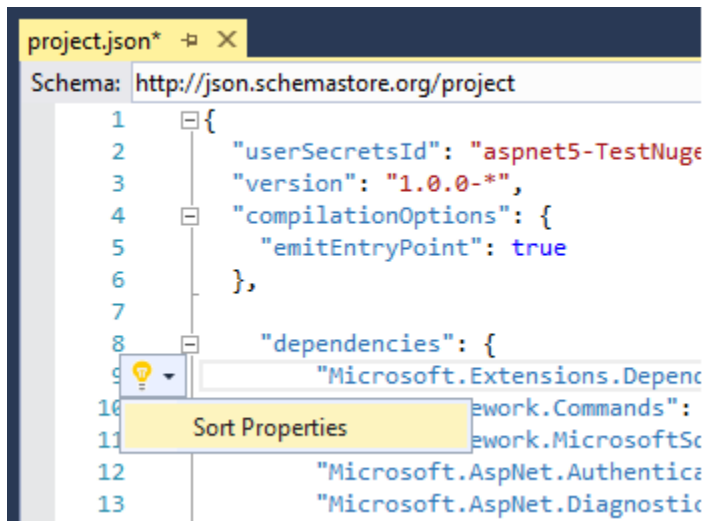
```

    "emitEntryPoint": true
  },

  "dependencies": {
    "Microsoft.Extensions.DependencyInjection": "1.0.0-rc1-final",
    "EntityFramework.MicrosoftSqlServer": "7.0.0-rc1-final",
    "Microsoft.AspNet.Authentication.Cookies": "1.0.0-rc1-final",

```

Optional: Tap the *quick actions* light bulb icon and select **Sort Properties**.



Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```

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

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            var context = serviceProvider.GetService<ApplicationDbContext>();

            if (context.Database == null)
            {
                throw new Exception("DB is null");
            }

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

            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-1-11"),
                    Genre = "Romantic Comedy",

```



```

        Price = 7.99M
    },

    new Movie
    {
        Title = "Ghostbusters ",
        ReleaseDate = DateTime.Parse("1984-3-13"),
        Genre = "Comedy",
        Price = 8.99M
    },

    new Movie
    {
        Title = "Ghostbusters 2",
        ReleaseDate = DateTime.Parse("1986-2-23"),
        Genre = "Comedy",
        Price = 9.99M
    },

    new Movie
    {
        Title = "Rio Bravo",
        ReleaseDate = DateTime.Parse("1959-4-15"),
        Genre = "Western",
        Price = 3.99M
    }
    );
    context.SaveChanges();
}
}
}

```

The `GetService` method comes from the DI package we just added. Notice if there are any movies in the DB, the seed initializer returns.

```

1  if (context.Movie.Any())
2  {
3      return;    // DB has been seeded
4  }

```

Add the seed initializer to the end of the `Configure` method in the *Startup.cs* file:

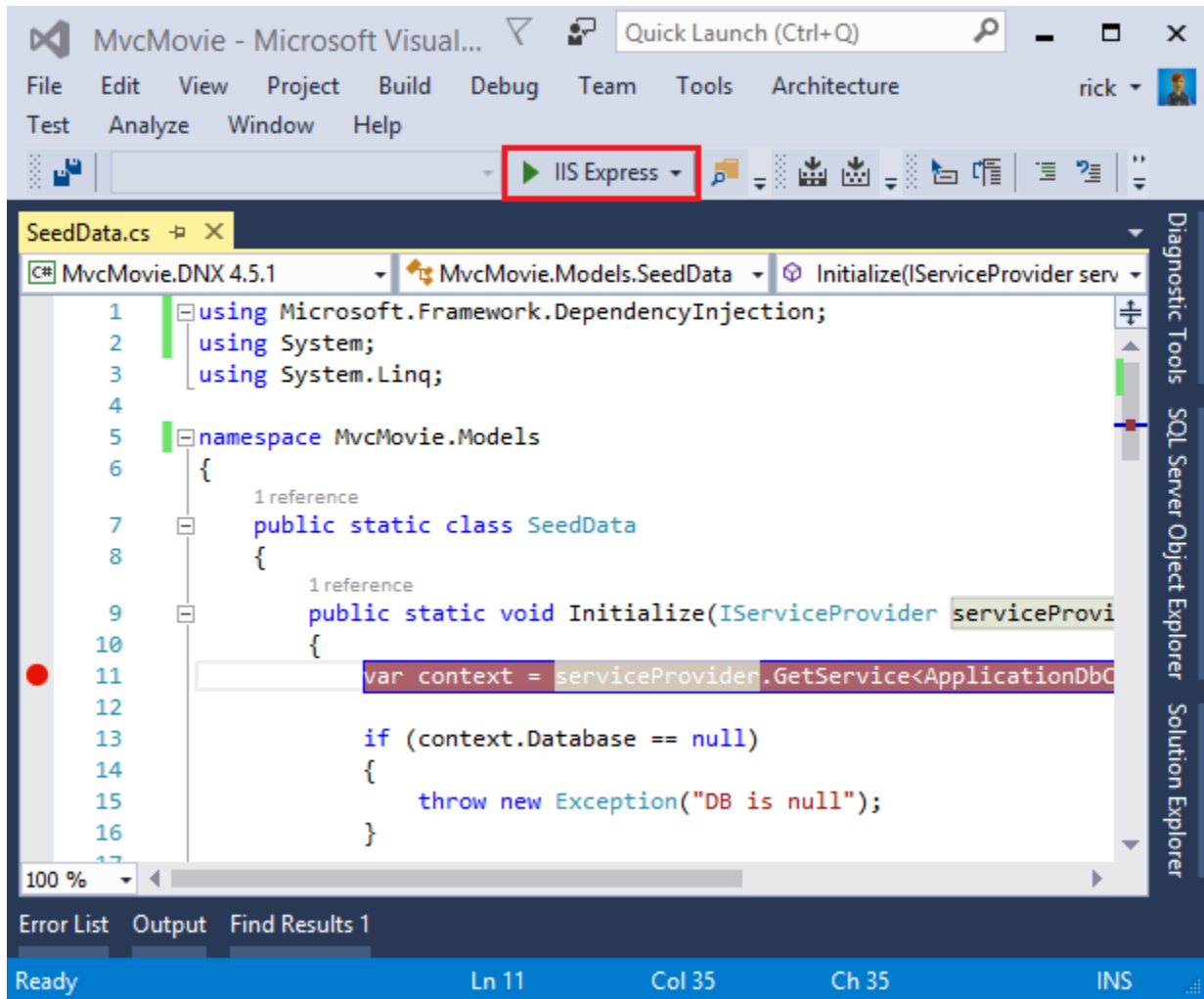
```

1  app.UseMvc(routes =>
2  {
3      routes.MapRoute(
4          name: "default",
5          template: "{controller=Home}/{action=Index}/{id?}");
6  });
7
8  SeedData.Initialize(app.ApplicationServices);
9  }

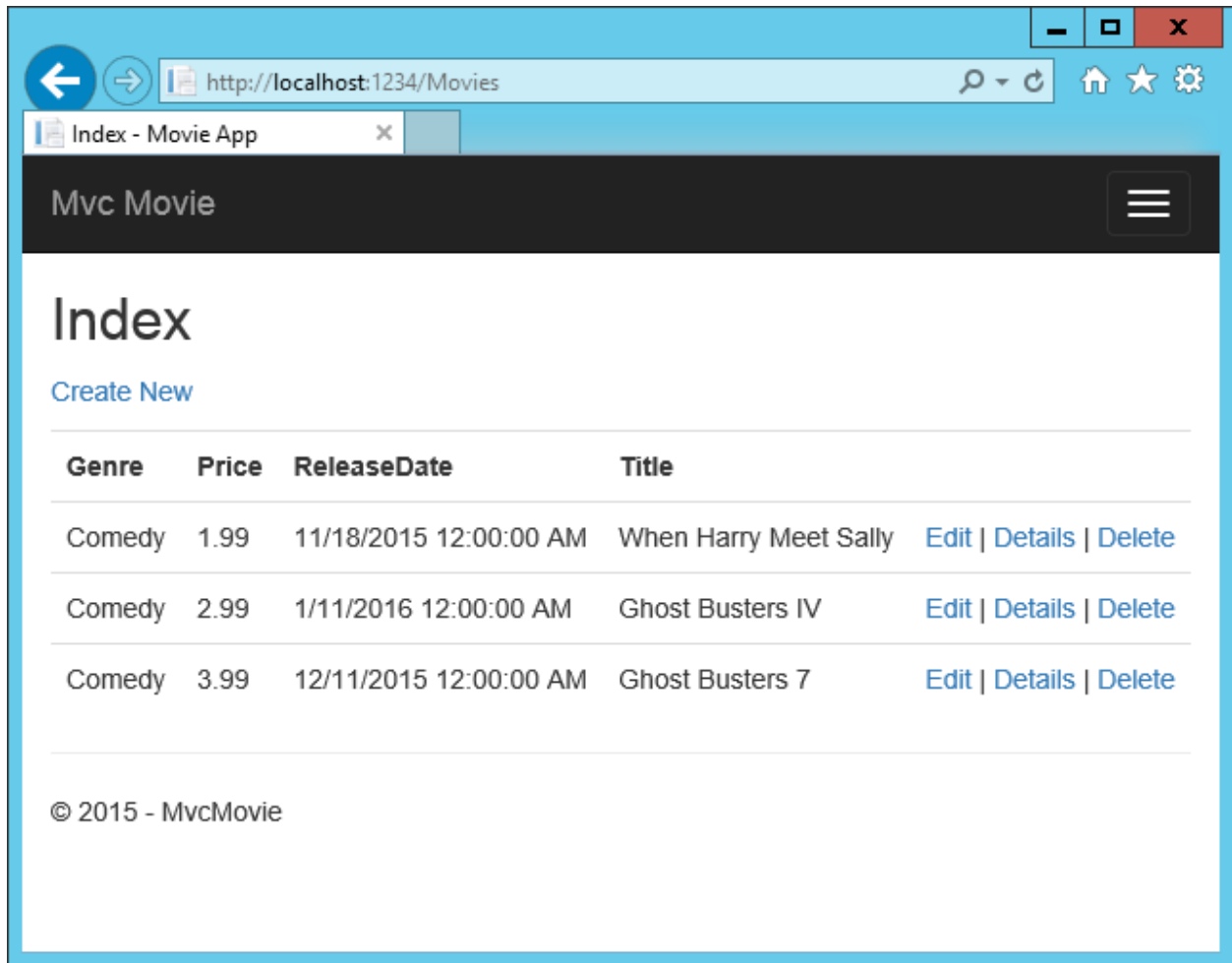
```

Test the app

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize so the seed method runs. You can do this by setting a break point on the first line of the `SeedData Initialize` method, and launching the debugger (Tap F5 or tap the **IIS Express** button).



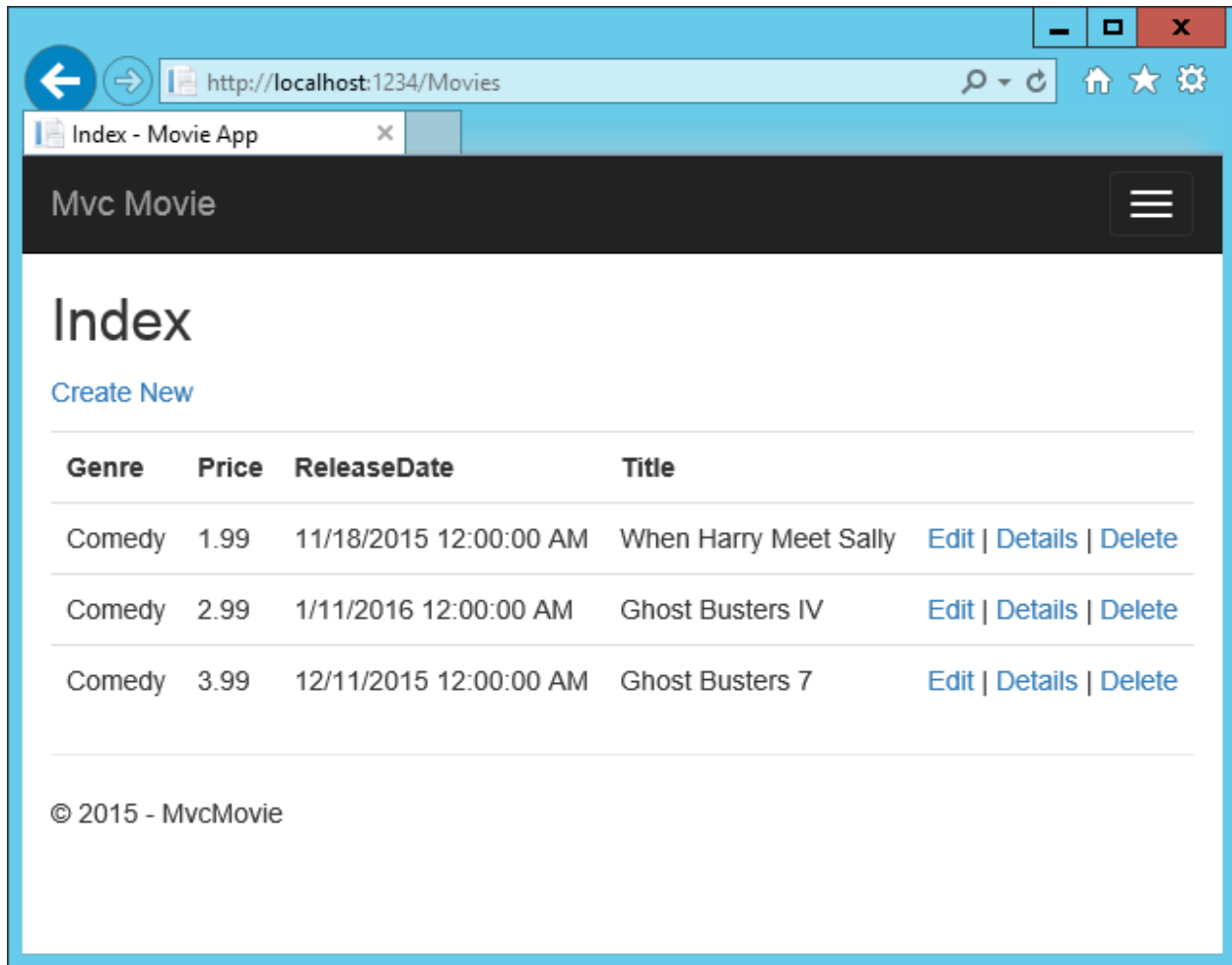
The app shows the seeded data.



2.1.6 Controller methods and views

By Rick Anderson

We have a good start to the movie app, but the presentation is not ideal. We don't want to see the time on the release date and **ReleaseDate** should be two words.



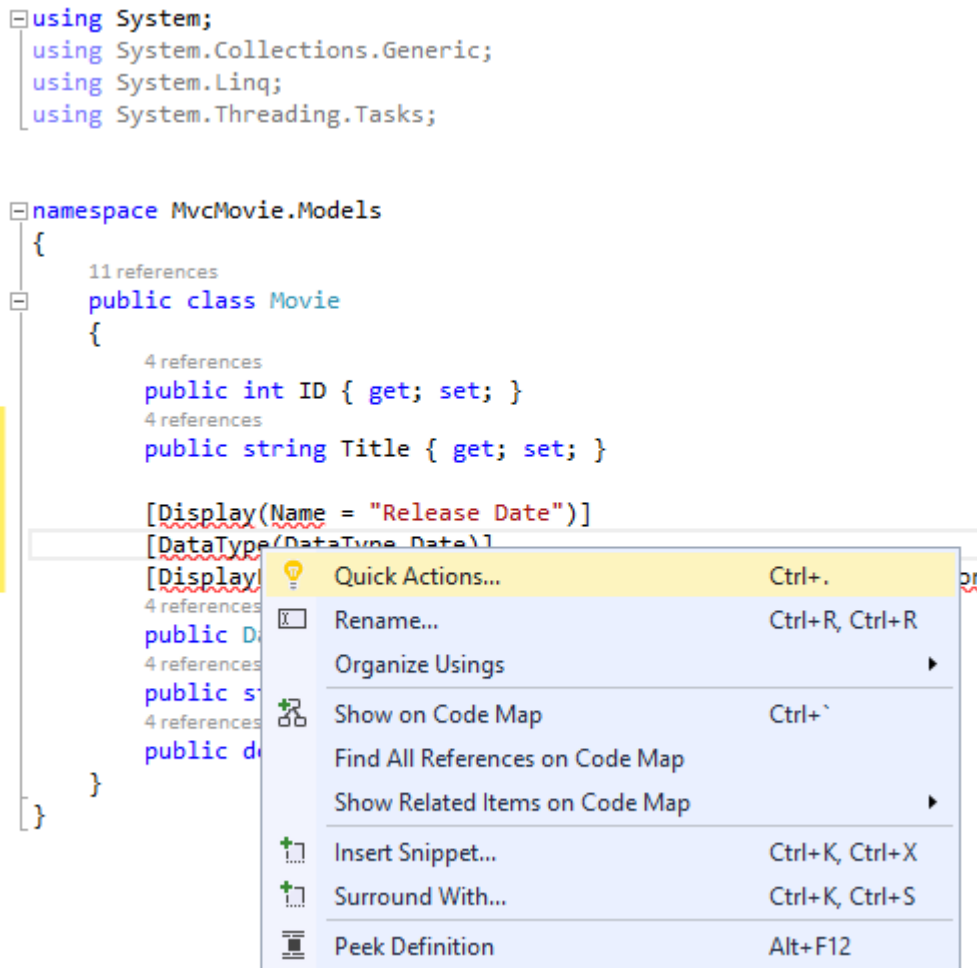
Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```

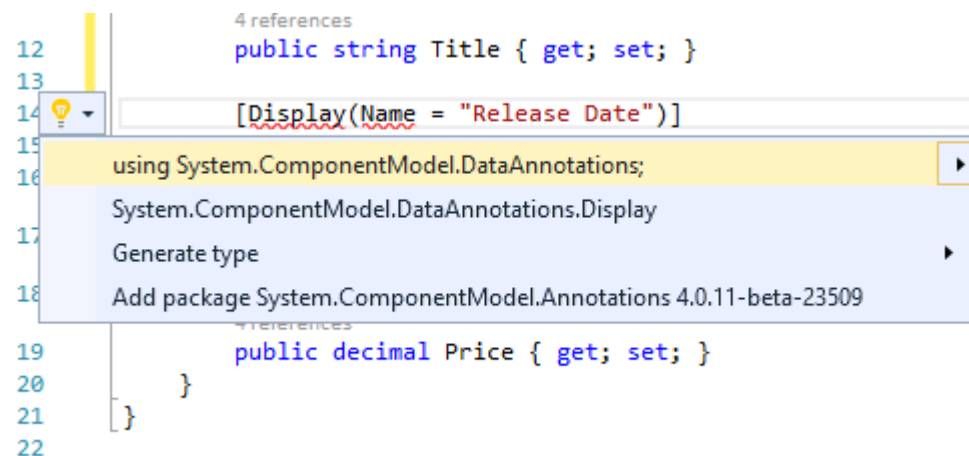
1 public class Movie
2 {
3     public int ID { get; set; }
4     public string Title { get; set; }
5
6     [Display(Name = "Release Date")]
7     [DataType(DataType.Date)]
8     public DateTime ReleaseDate { get; set; }
9     public string Genre { get; set; }
10    public decimal Price { get; set; }
11 }

```

- Right click on a red squiggly line > **Quick Actions**.

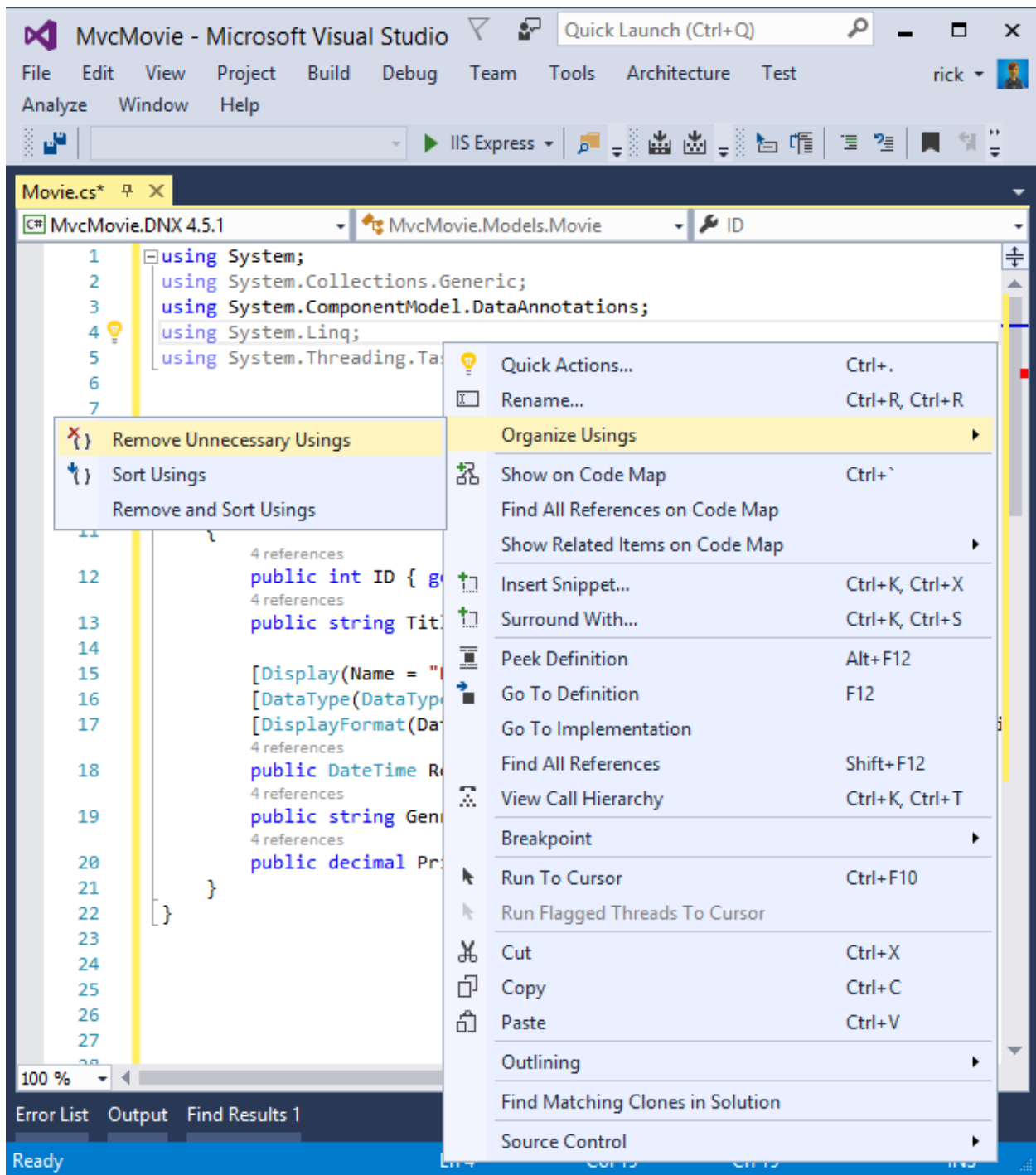


- Tap `using System.ComponentModel.DataAnnotations;`



Visual studio adds `using System.ComponentModel.DataAnnotations;`.

Let's remove the using statements that are not needed. They show up by default in a light grey font. Right click anywhere in the *Movie.cs* file > **Organize Usings** > **Remove Unnecessary Usings**.



The completed is shown below:

```

1  using System;
2  using System.ComponentModel.DataAnnotations;
3
4  namespace MvcMovie.Models
5  {
6      public class Movie
7      {
8          public int ID { get; set; }
9      }
```

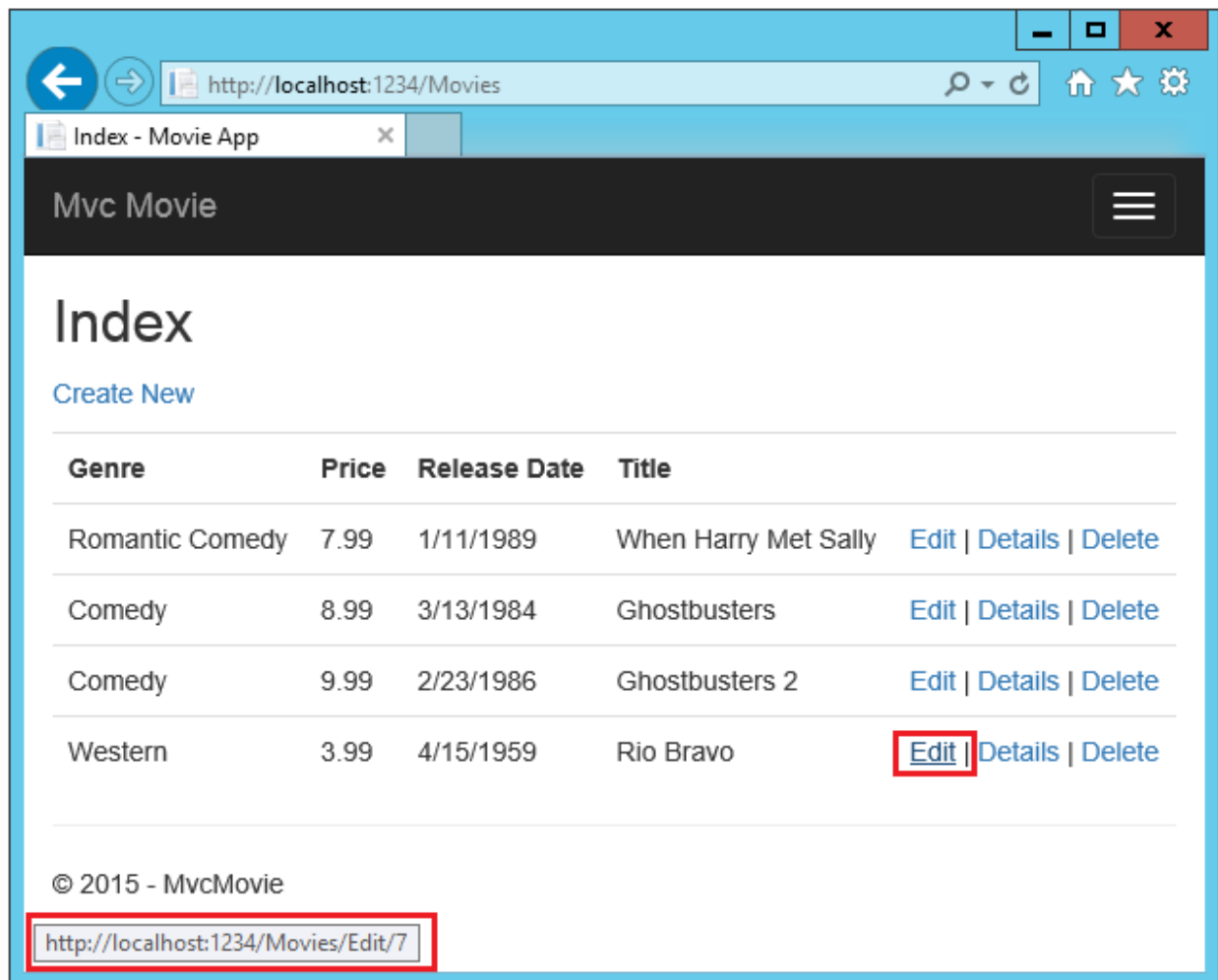
```

9      public string Title { get; set; }
10
11      [Display(Name = "Release Date")]
12      [DataType(DataType.Date)]
13      public DateTime ReleaseDate { get; set; }
14      public string Genre { get; set; }
15      public decimal Price { get; set; }
16  }
17  }

```

We'll cover [DataAnnotations](#) in the next tutorial. The [Display](#) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](#) attribute specifies the type of the data, in this case it's a date, so the time information stored in the field is not displayed.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



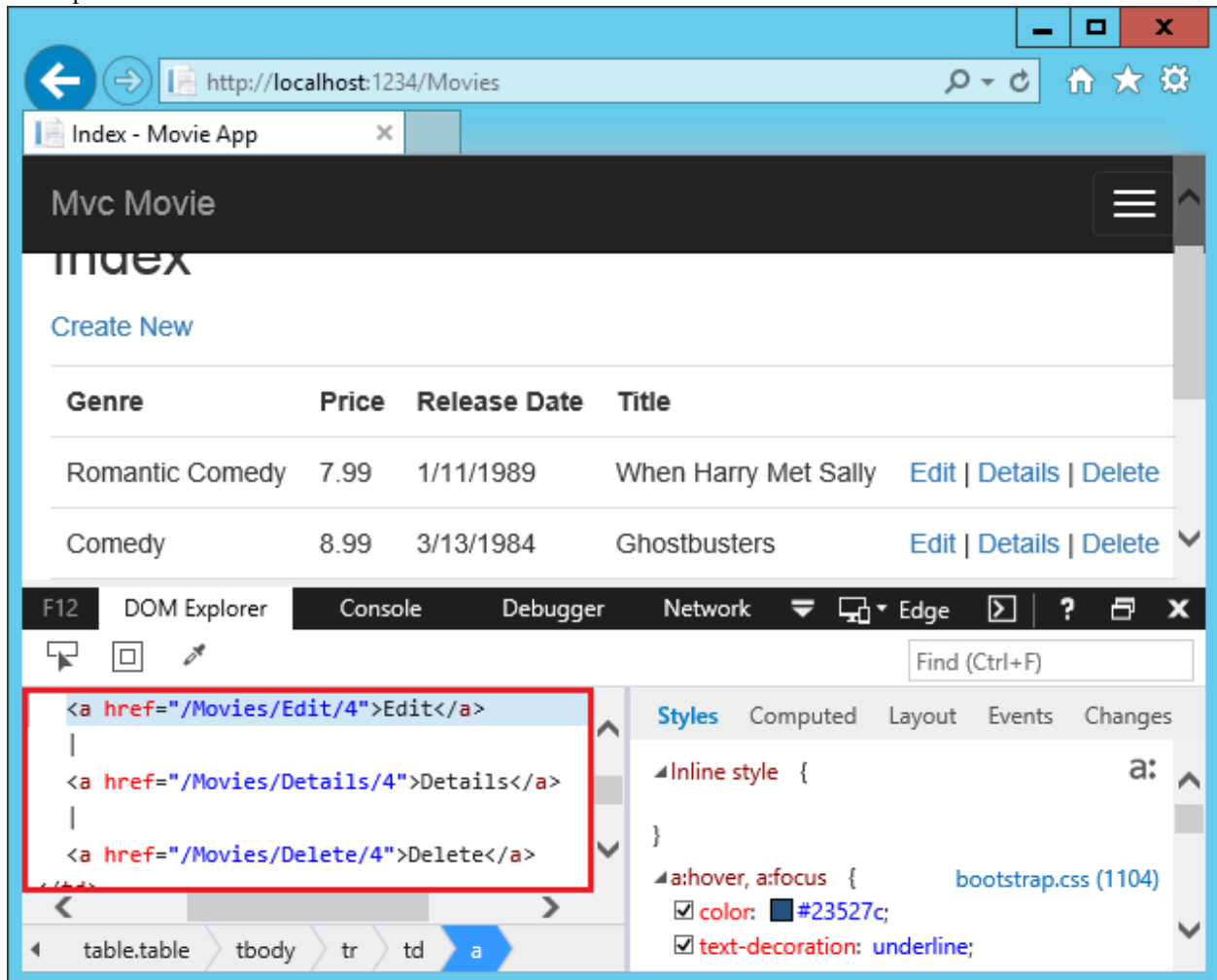
The **Edit**, **Details**, and **Delete** links are generated by the MVC 6 [Anchor Tag Helper](#) in the `Views/Movies/Index.cshtml` file.

```

1  <td>
2      <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
3      <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
4      <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
5  </td>

```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the [Anchor Tag Helper](#) dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the **F12** tools to examine the generated markup. The **F12** tools are shown below.



Recall the format for routing set in the *Startup.cs* file.

```

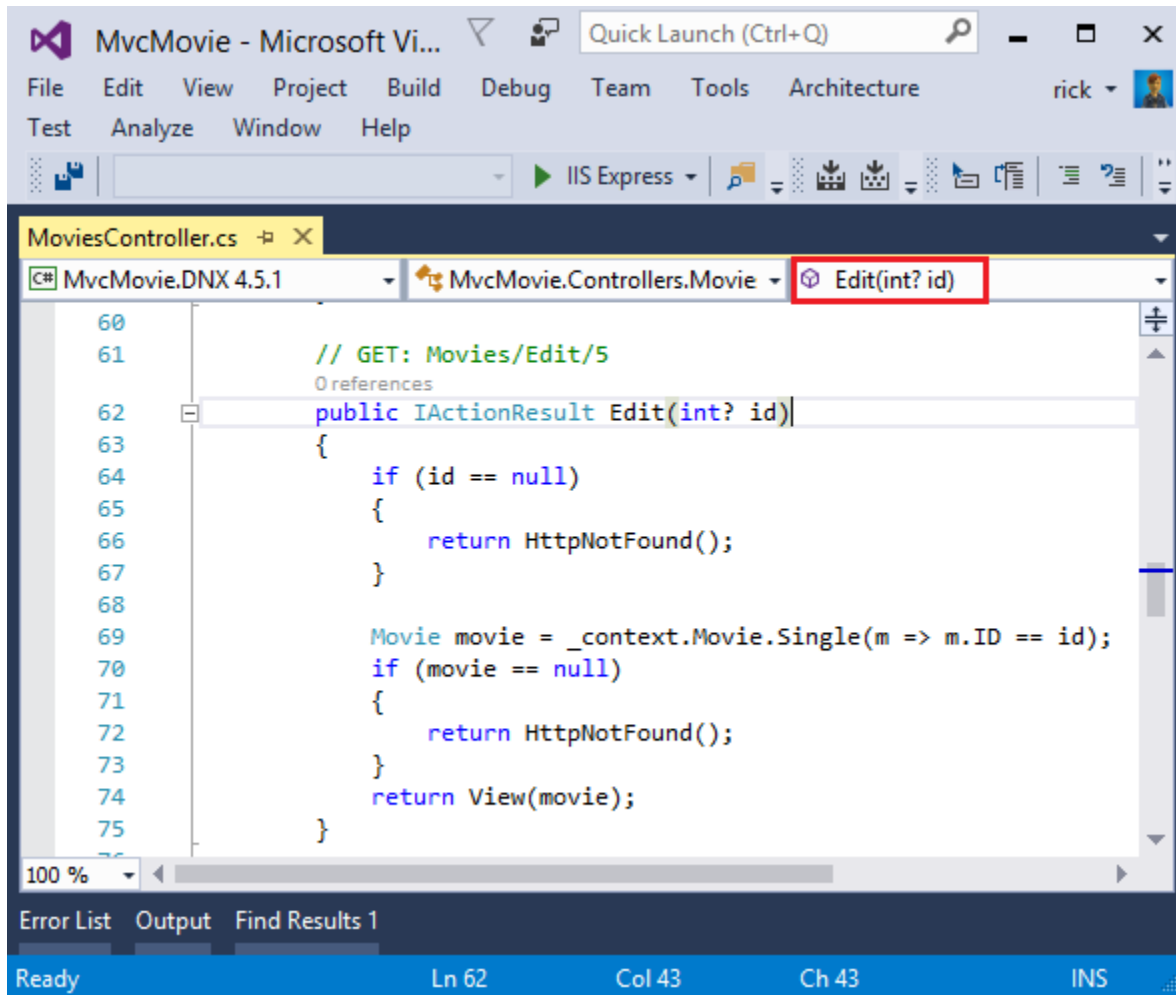
1  app.UseMvc(routes =>
2  {
3      routes.MapRoute(
4          name: "default",
5          template: "{controller=Home}/{action=Index}/{id?}");
6  });

```

ASP.NET translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` of 4. (Controller methods are also known as [action methods](#).)

[Tag Helpers](#) are one of the most popular new features in ASP.NET 5. See [Additional resources](#) for more information.

Open the `Movies` controller and examine the two `Edit` action methods:



```

1 public IActionResult Edit(int? id)
2 {
3     if (id == null)
4     {
5         return HttpNotFound();
6     }
7
8     Movie movie = _context.Movie.Single(m => m.ID == id);
9     if (movie == null)
10    {
11        return HttpNotFound();
12    }
13    return View(movie);
14 }
15
16 // POST: Movies/Edit/5
17 [HttpPost]
18 [ValidateAntiForgeryToken]
19 public IActionResult Edit(Movie movie)
20 {
21     if (ModelState.IsValid)
22     {
23         _context.Update(movie);
24         _context.SaveChanges();

```

```
25         return RedirectToAction("Index");
26     }
27     return View(movie);
28 }
```

Note: The scaffolding engine generated code above has a serious [over-posting security vulnerability](#). Be sure you understand how to protect from over-posting before you publish your app. This security vulnerability should be fixed in the next release.

Replace the HTTP POST Edit action method with the following:

```
1 // POST: Movies/Edit/6
2 [HttpPost]
3 [ValidateAntiForgeryToken]
4 public IActionResult Edit(
5     [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
6 {
7     if (ModelState.IsValid)
8     {
9         _context.Update(movie);
10        _context.SaveChanges();
11        return RedirectToAction("Index");
12    }
13    return View(movie);
14 }
```

The [Bind] attribute is one way to protect against [over-posting](#). You should only include properties in the [Bind] attribute that you want to change. Apply the [Bind] attribute to each of the [HttpPost] action methods. See [Protect your controller from over-posting](#) for more information.

Notice the second Edit action method is preceded by the [HttpPost] attribute.

```
1 // POST: Movies/Edit/6
2 [HttpPost]
3 [ValidateAntiForgeryToken]
4 public IActionResult Edit(
5     [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
6 {
7     if (ModelState.IsValid)
8     {
9         _context.Update(movie);
10        _context.SaveChanges();
11        return RedirectToAction("Index");
12    }
13    return View(movie);
14 }
```

The [HttpPost] attribute specifies that this Edit method can be invoked *only* for POST requests. You could apply the [HttpGet] attribute to the first edit method, but that's not necessary because [HttpGet] is the default.

The [ValidateAntiForgeryToken] attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token in the [Form Tag Helper](#).

```
<form asp-action="Edit">
```

The [Form Tag Helper](#) generates a hidden anti-forgery token that must match the [ValidateAntiForgeryToken] generated anti-forgery token in the Edit method of the Movies controller. For more information, see [Anti-Request Forgery](#).

The `HttpGet Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Single` method, and returns the selected movie to the Edit view. If a movie cannot be found, `HttpNotFound` is returned.

```

1 // GET: Movies/Edit/5
2 public IActionResult Edit(int? id)
3 {
4     if (id == null)
5     {
6         return HttpNotFound();
7     }
8
9     Movie movie = _context.Movie.Single(m => m.ID == id);
10    if (movie == null)
11    {
12        return HttpNotFound();
13    }
14    return View(movie);
15 }

```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the visual studio scaffolding system:

```

@model MvcMovie.Models.Movie

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

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ValidationSummary.ModelOnly" class="text-danger" />
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
            <label asp-for="Genre" class="control-label col-md-2" />
            <div class="col-md-10">
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger" />
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Price" class="control-label col-md-2" />
            <div class="col-md-10">
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger" />
            </div>
        </div>
        @*ReleaseDate and Title removed for brevity.*@

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>

```

```
</form>

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

@section Scripts {
  <script src="~/lib/jquery/dist/jquery.min.js"></script>
  <script src="~/lib/jquery-validation/jquery.validate.min.js"></script>
  <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"></script>
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file — this specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The [Label Tag Helper](#) displays the name of the field (“Title”, “ReleaseDate”, “Genre”, or “Price”). The [Input Tag Helper](#) renders an HTML `<input>` element. The [Validation Tag Helpers](#) displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

```
<form action="/Movies/Edit/7" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <hr />
    <div class="text-danger" />
    <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" />
    <div class="form-group">
      <label class="control-label col-md-2" for="Genre" />
      <div class="col-md-10">
        <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
        <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg="The field Genre is required." />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-md-2" for="Price" />
      <div class="col-md-10">
        <input class="form-control" type="text" data-val="true" data-val-number="The field Price must be a number." />
        <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg="The field Price must be a number." />
      </div>
    </div>
    <!-- Markup removed for brevity -->
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
        <input type="submit" value="Save" class="btn btn-default" />
      </div>
    </div>
  </div>
  <input name="__RequestVerificationToken" type="hidden" value="CfDJ8Inyxp63fRFqUePGvuI5jGZsloJul1" />
</form>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the Save button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the [Form Tag Helper](#).

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

```
// POST: Movies/Edit/6
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(
    [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Update(movie);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the anti-forgery token generator in the [Form Tag Helper](#).

The [ASP.NET MVC model binder](#) takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, the movie data is saved to the `Movies` collection of the database (`ApplicationDbContext` instance). The new movie data is saved to the database by calling the `SaveChanges` method of `ApplicationDbContext`. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

As soon as the client side validation determines the values of a field are not valid, an error message is displayed. If you disable JavaScript, you won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine validation in more detail.

The [Validation Tag Helper](#) in the `Views/Book/Edit.cshtml` view template takes care of displaying appropriate error messages.

http://localhost:1235/Movies/Edit/

Edit - Movie App

Mvc Movie

Edit

Movie

Genre

Western

Price

abc

The field Price must be a number.

Release Date

xyz

Please enter a valid date.

Title

Rio Bravo

Save

[Back to List](#)

© 2015 - MvcMovie

All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post [ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes](#). Modifying data in a HTTP GET method also violates HTTP best practices and the architectural [REST](#) pattern, which specifies that GET requests should not change the state

of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)
- [Anti-Request Forgery](#)
- [Protect your controller from over-posting](#)
- [Form Tag Helper](#)
- [Label Tag Helper](#)
- [Input Tag Helper](#)
- [Validation Tag Helpers](#)
- [Anchor Tag Helper](#)
- [Select Tag Helper](#)

2.1.7 Adding Search

By [Rick Anderson](#)

Adding a Search Method and Search View

In this section you'll add search capability to the Index action method that lets you search movies by genre or name.

Updating Index

Start by updating the Index action method to enable search. Here's the code:

```
1 public IActionResult Index(string searchString)
2 {
3     var movies = from m in _context.Movie
4                   select m;
5
6     if (!String.IsNullOrEmpty(searchString))
7     {
8         movies = movies.Where(s => s.Title.Contains(searchString));
9     }
10
11     return View(movies);
12 }
```

The first line of the Index action method creates a [LINQ](#) query to select the movies:

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

The query is *only* defined at this point, it *has not* been run against the database.

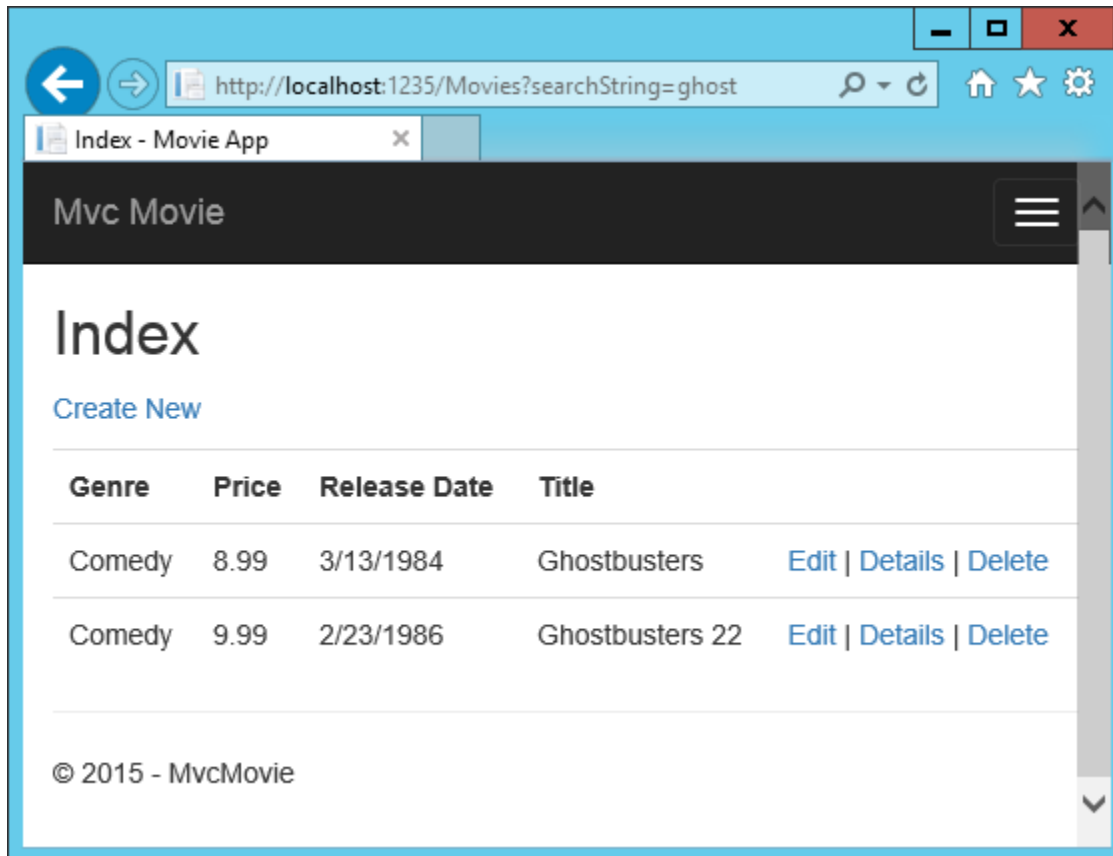
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string, using the following code:

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

The `s => s.Title` code above is a [Lambda Expression](#). Lambdas are used in method-based [LINQ](#) queries as arguments to standard query operator methods such as the [Where](#) method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` or `OrderBy`. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToList` method is called. In the Search sample, the query is executed in the *Index.cshtml* view. For more information about deferred query execution, see [Query Execution](#). **Note:** The `Contains` method is run on the database, not the `c#` code above. On the database, `Contains` maps to `SQL LIKE`, which is case insensitive.

Now you can update the `Index` view that will display the form to the user.

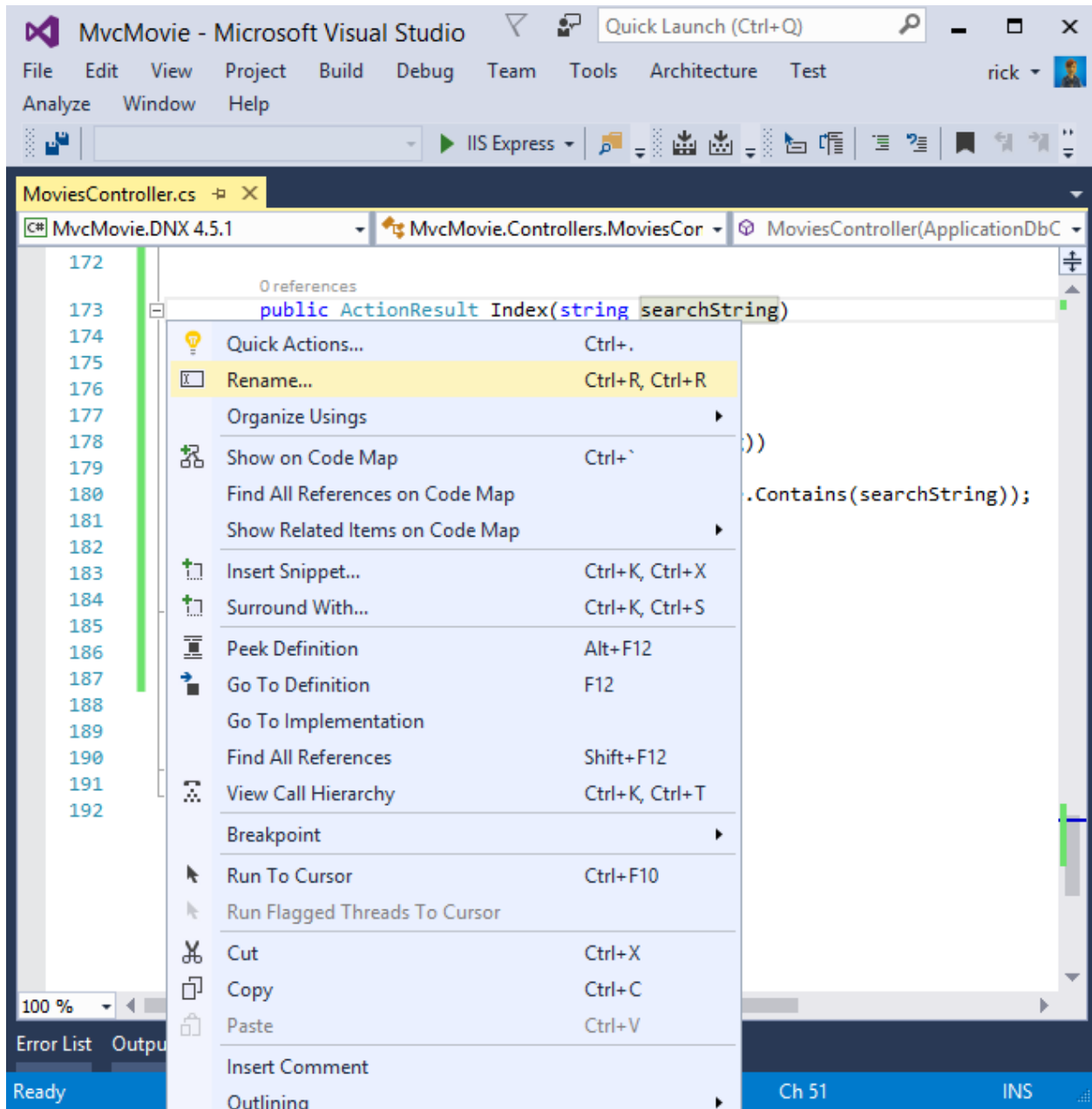
Navigate to `/Movies/Index`. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.

```
template: "{controller=Home}/{action=Index}/{id?}";
```

You can quickly rename the `searchString` parameter to `id` with the **rename** command. Right click on `searchString` > **Rename**.



The rename targets are highlighted.

```
public ActionResult Index(string searchString)
{
    var movies = from m in _context.Movie
                  select m;

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

    return View(movies);
}
```

Change the parameter to `id` and all occurrences of `searchString` change to `id`.

```
public ActionResult Index(string id)
{
    var movies = from m in _context.Movie
                  select m;

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

    return View(movies);
}
```

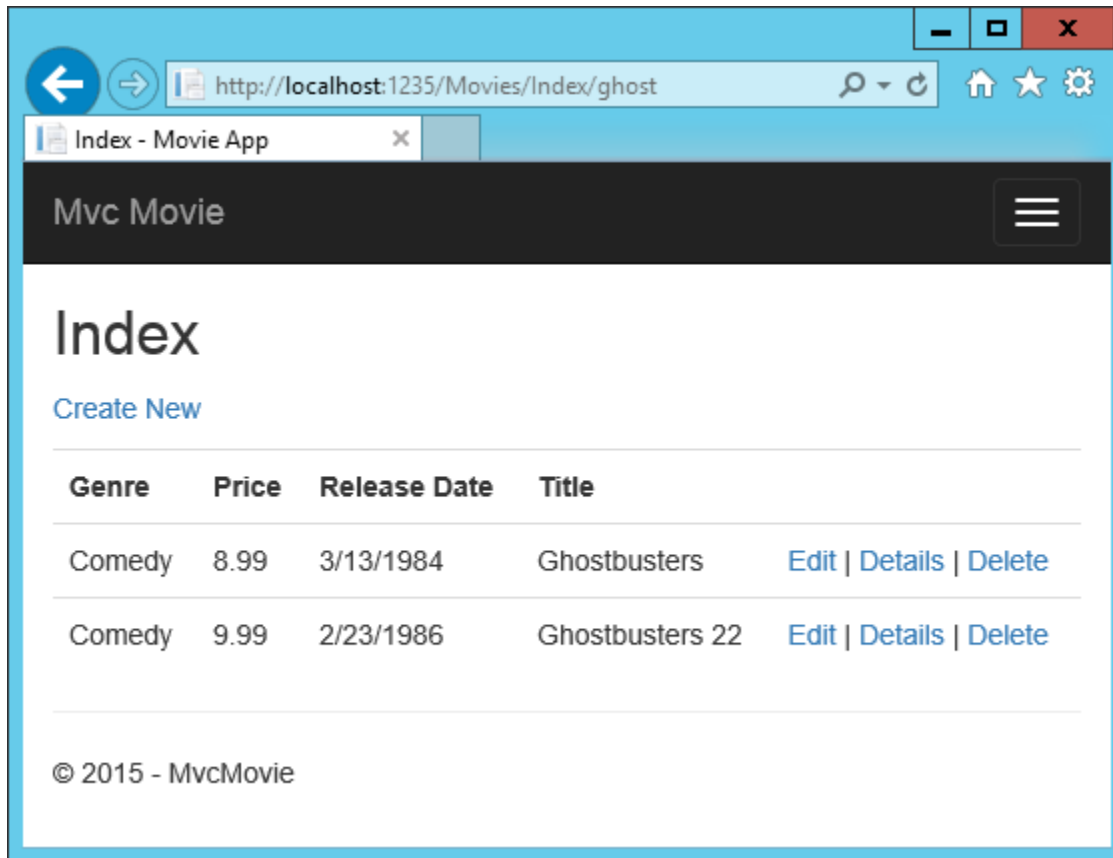
The previous `Index` method:

```
1 public IActionResult Index(string searchString)
2 {
3     var movies = from m in _context.Movie
4                   select m;
5
6     if (!String.IsNullOrEmpty(searchString))
7     {
8         movies = movies.Where(s => s.Title.Contains(searchString));
9     }
10
11     return View(movies);
12 }
```

The updated `Index` method:

```
1 public IActionResult Index(string id)
2 {
3     var movies = from m in _context.Movie
4                   select m;
5
6     if (!String.IsNullOrEmpty(id))
7     {
8         movies = movies.Where(s => s.Title.Contains(id));
9     }
10
11     return View(movies);
12 }
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound ID parameter, change it back so that it takes a parameter named `searchString`:

```
public IActionResult Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

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

    return View(movies);
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

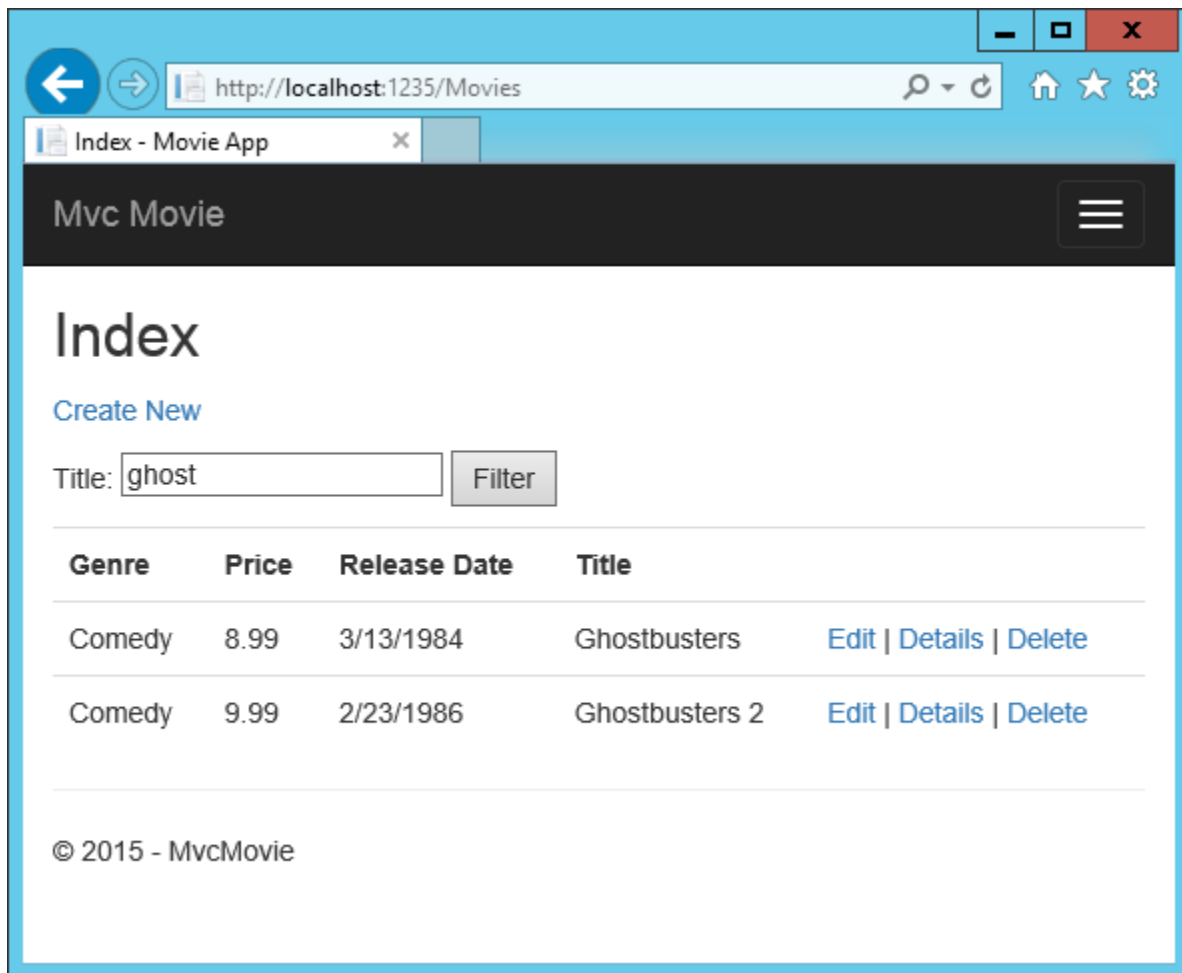
```
1 @model IEnumerable<MvcMovie.Models.Movie>
2
3 @{
4     ViewData["Title"] = "Index";
5 }
6
7 <h2>Index</h2>
8
```

```

9 <p>
10     <a asp-action="Create">Create New</a>
11 </p>
12
13 <form asp-controller="Movies" asp-action="Index">
14     <p>
15         Title: <input type="text" name="SearchString">
16         <input type="submit" value="Filter" />
17     </p>
18 </form>
19
20 <table class="table">
21     <tr>

```

The HTML `<form>` tag is super-charged by the [Form Tag Helper](#), so when you submit the form, the filter string is posted to the `Index` action of the `movies` controller. Save your changes and then test the filter.



There's no `[HttpPost]` overload of the `Index` method. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost]` `Index` method.

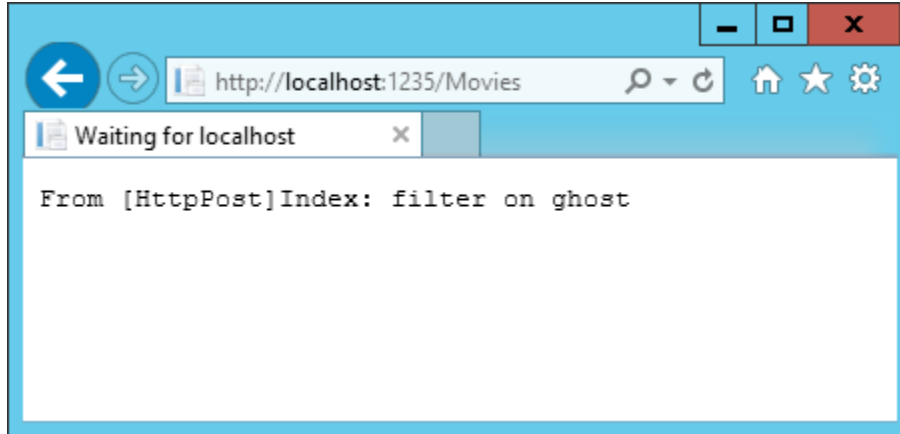
```

[HttpPost]
public string Index(FormCollection fc, string searchString)
{

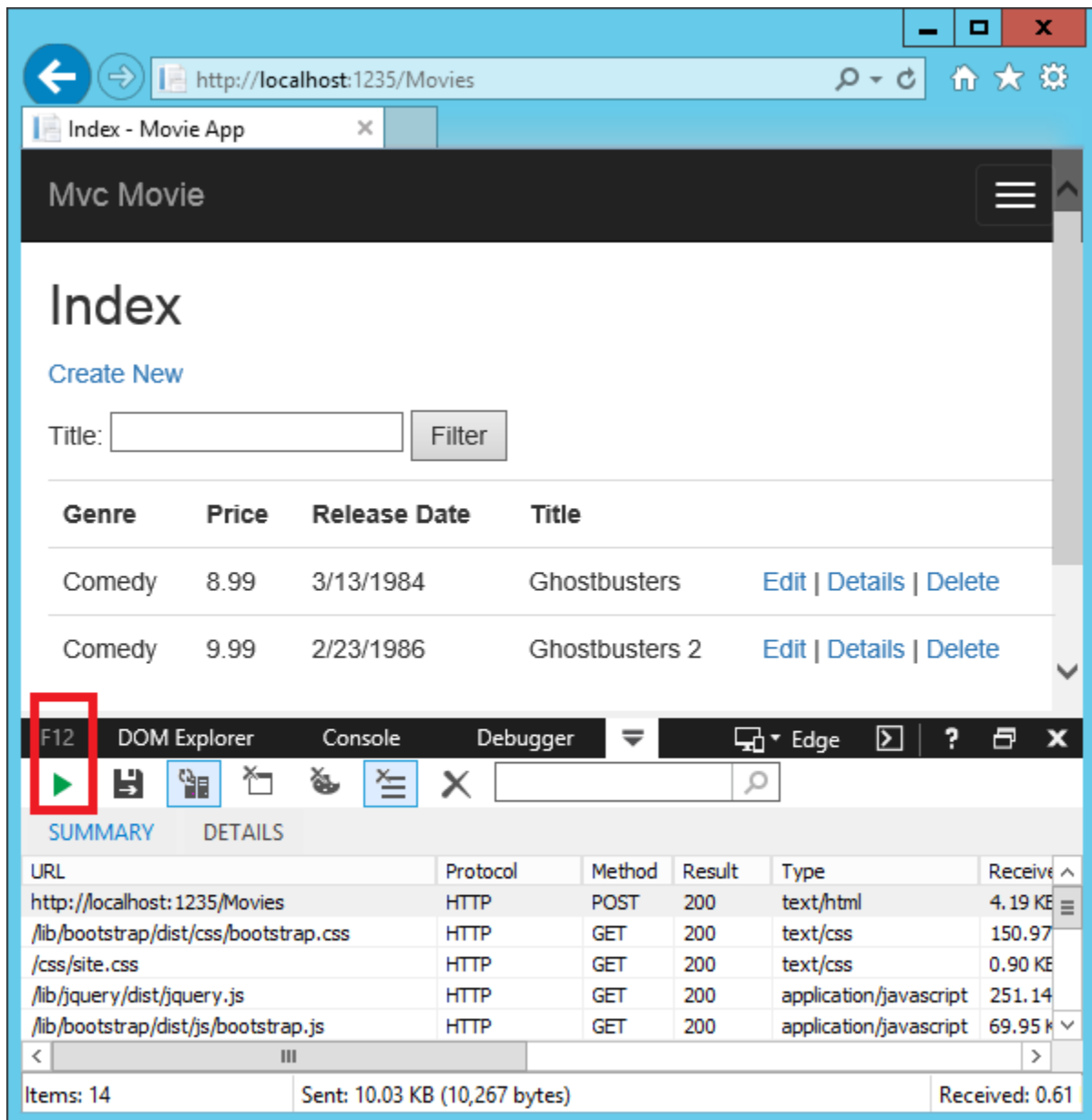
```

```
return "From [HttpPost]Index: filter on " + searchString;  
}
```

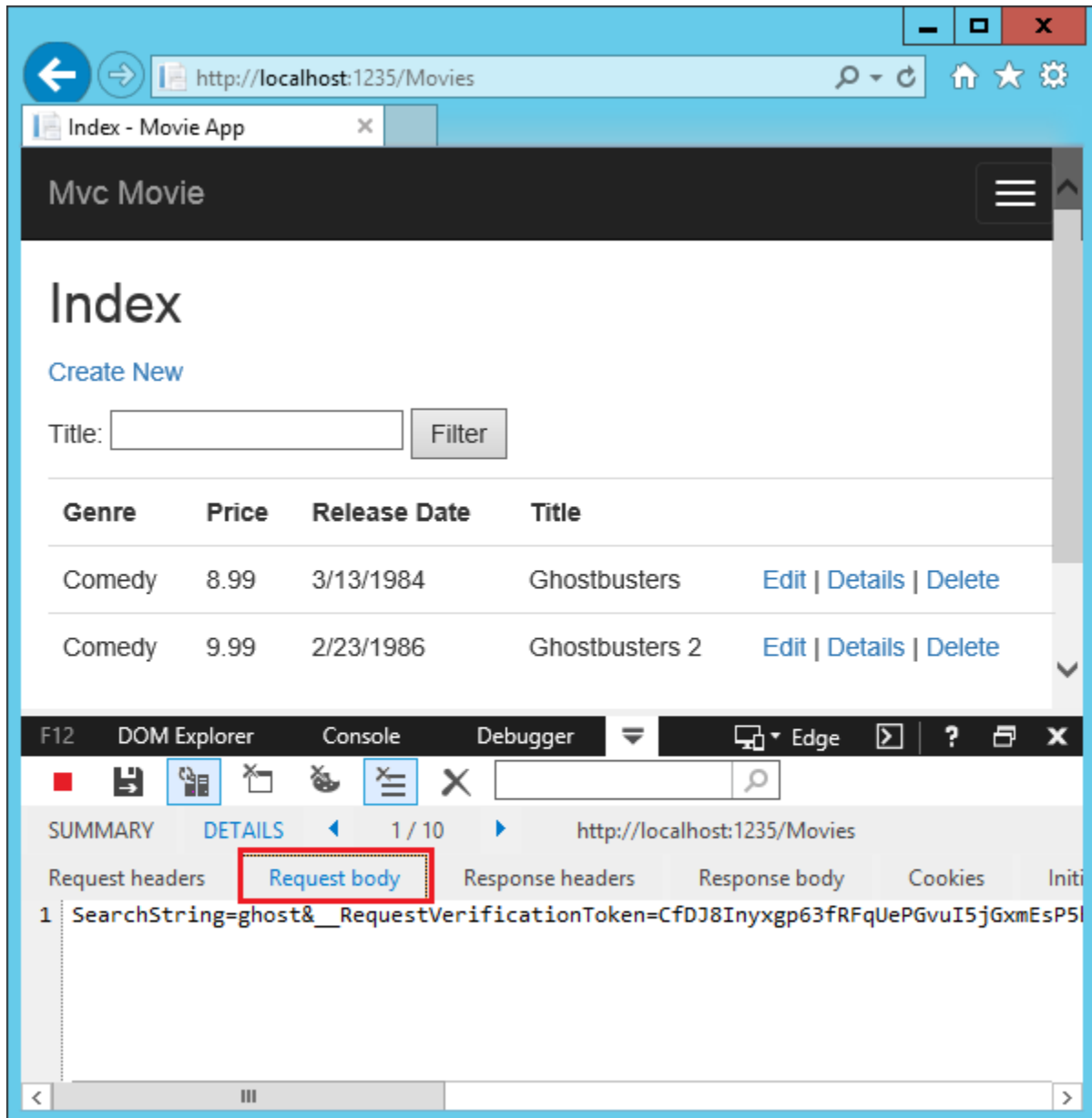
If you did, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (`localhost:xxxxx/Movies/Index`) – there's no search information in the URL itself. Right now, the search string information is sent to the server as a form field value. You can verify that with the [F12 Developer tools](#) or the excellent [Fiddler tool](#). Start the [F12 tool](#) and tap the **Enable network traffic capturing** icon.



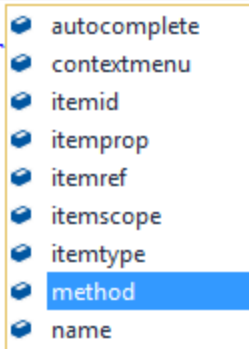
Double tap the `http://localhost:1235/Movies` HTTP POST 200 line and then tap **Request body**.



You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the [Form Tag Helper](#) generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

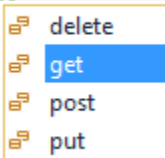
Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. We'll fix this by specifying the request should be `HTTP GET`. Notice how `intelliSense` helps us update the markup.

```
<form asp-controller="Movies" asp-action="Index" m>
  <p>
    Title: <input type="text" name="SearchStr" />
    <input type="submit" value="Filter" />
  </p>
</form>
```



- autocomplete
- contextmenu
- itemid
- itemprop
- itemref
- itemscope
- itemtype
- method**
- name

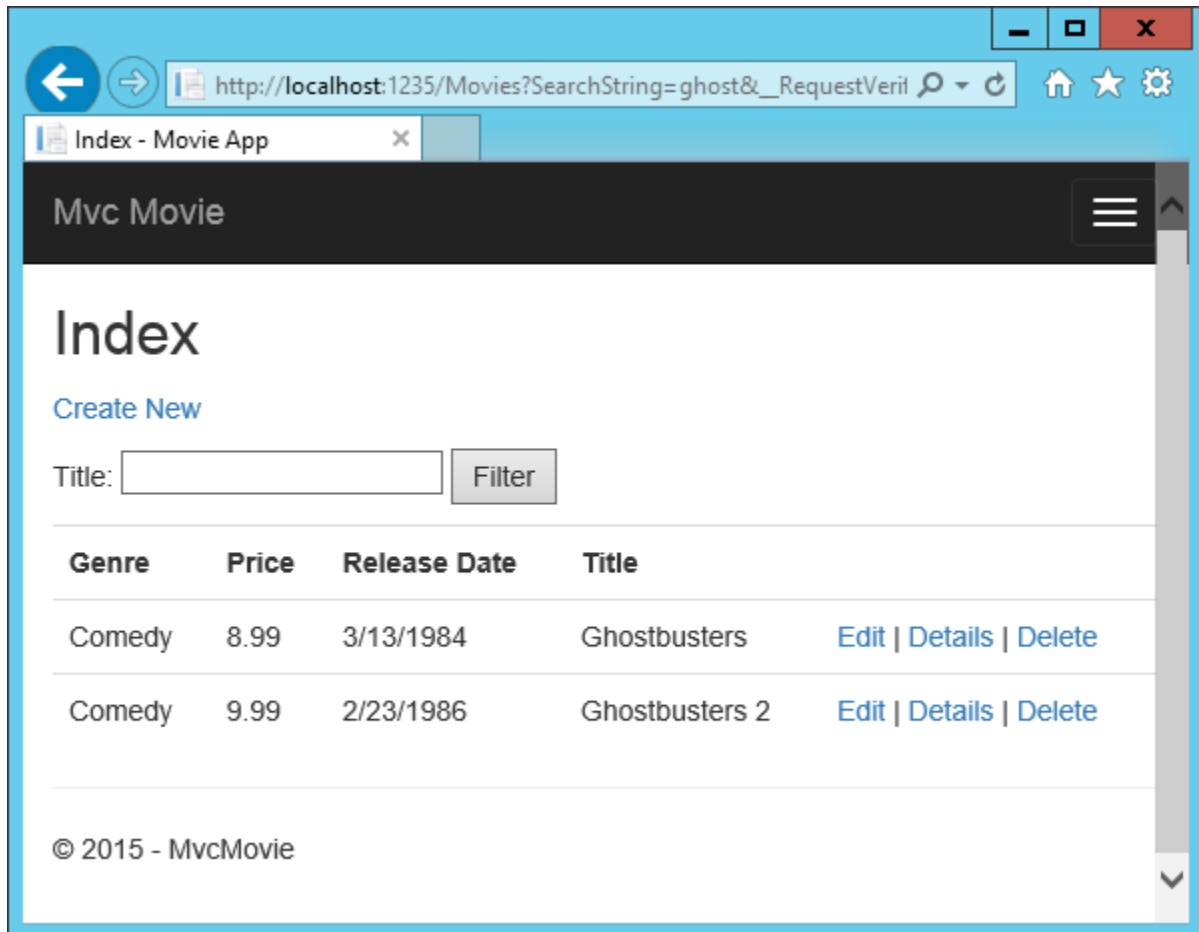
```
<form asp-controller="Movies" asp-action="Index" method="get">
  <p>
    Title: <input type="text" name="SearchString" />
    <input type="submit" value="Filter" />
  </p>
</form>
```



- delete
- get**
- post
- put

Notice the distinctive font in the `<form>` tag. That distinctive font indicates the tag is supported by [Tag Helpers](#).

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet` `Index` action method, even if you have a `HttpPost` `Index` method.



The XSRF token and any other posted form elements will also be added to the URL.

Adding Search by Genre

Replace the `Index` method with the following code:

```
chGenre

public IActionResult Index(string movieGenre, string searchString)
{
    var GenreQry = from m in _context.Movie
                   orderby m.Genre
                   select m.Genre;

    var GenreList = new List<string>();
    GenreList.AddRange(GenreQry.Distinct());
    ViewData["movieGenre"] = new SelectList(GenreList);

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

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

```
if (!string.IsNullOrEmpty(movieGenre))
{
    movies = movies.Where(x => x.Genre == movieGenre);
}

return View(movies);
}
```

This version of the `Index` method takes a `movieGenre` parameter. The first few lines of code create a `List` object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

```
var GenreQry = from m in _context.Movie
               orderby m.Genre
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in a `ViewData` dictionary. Storing category data (such as a movie genre's) as a `SelectList` object in a `ViewData` dictionary, then accessing the category data in a dropdown list box is a typical approach for MVC apps.

The following code shows how to check the `movieGenre` parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

```
if (!string.IsNullOrEmpty(movieGenre))
{
    movies = movies.Where(x => x.Genre == movieGenre);
}
```

As stated previously, the query is not run on the data base until the movie list is iterated over (which happens in the View, after the `Index` action method returns).

Adding search by genre to the Index view

Add an `Html.DropDownList` helper to the `Views/Movies/Index.cshtml` file. The completed markup is shown below:

```
1 <form asp-controller="Movies" asp-action="Index" method="get">
2     <p>
3         Genre: @Html.DropDownList("movieGenre", "All")
4         Title: <input type="text" name="SearchString">
5         <input type="submit" value="Filter" />
6     </p>
7 </form>
```

Note: The next version of this tutorial will replace the `Html.DropDownList` helper with the [Select Tag Helper](#).

Test the app by searching by genre, by movie title, and by both.

2.1.8 Adding a New Field

By [Rick Anderson](#)

In this section you'll use [Entity Framework Code First Migrations](#) to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a *Rating* property:

```

1 public class Movie
2 {
3     public int ID { get; set; }
4     public string Title { get; set; }
5
6     [Display(Name = "Release Date")]
7     [DataType(DataType.Date)]
8     public DateTime ReleaseDate { get; set; }
9     public string Genre { get; set; }
10    public decimal Price { get; set; }
11    public string Rating { get; set; }
12 }

```

Build the app (Ctrl+Shift+B).

Because you've added a new field to the *Movie* class, you also need to update the binding white list so this new property will be included. Update the *[Bind]* attribute for *Create* and *Edit* action methods to include the *Rating* property:

```
[Bind("ID,Title,ReleaseDate,Genre,Price,Rating")]
```

You also need to update the view templates in order to display, create and edit the new *Rating* property in the browser view.

Edit the */Views/Movies/Index.cshtml* file and add a *Rating* field:

```

1 <table class="table">
2     <tr>
3         <th>
4             @Html.DisplayNameFor(model => model.Genre)
5         </th>
6         <th>
7             @Html.DisplayNameFor(model => model.Price)
8         </th>
9         <th>
10            @Html.DisplayNameFor(model => model.ReleaseDate)
11        </th>
12        <th>
13            @Html.DisplayNameFor(model => model.Title)
14        </th>
15        <th>
16            @Html.DisplayNameFor(model => model.Rating)
17        </th>
18        <th></th>
19    </tr>
20
21    @foreach (var item in Model) {
22        <tr>
23            <td>
24                @Html.DisplayFor(modelItem => item.Genre)

```

```

25     </td>
26     <td>
27         @Html.DisplayFor(modelItem => item.Price)
28     </td>
29     <td>
30         @Html.DisplayFor(modelItem => item.ReleaseDate)
31     </td>
32     <td>
33         @Html.DisplayFor(modelItem => item.Title)
34     </td>
35     <td>
36         @Html.DisplayFor(modelItem => item.Rating)
37     </td>
38     <td>


```

Update the `/Views/Movies/Create.cshtml` with a Rating field. You can copy/paste the previous “form group” and let IntelliSense help you update the fields. IntelliSense works with [Tag Helpers](#).

```

    </div>
    <div class="form-group">
        <label asp-for="Title" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Title" class="form-control" />
            <span asp-validation-for="Title" class="text-danger" />
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Rating" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Rating" class="form-control" />
            <span asp-validation-for="Rating" class="text-danger" />
        </div>
    </div>
    </div>
</form>

```



The changed are highlighted below:

```

1  <form asp-action="Create">
2      <div class="form-horizontal">
3          <h4>Movie</h4>
4          <hr />
5          <div asp-validation-summary="ValidationSummary.ModelOnly" class="text-danger"></div>
6          <div class="form-group">
7              <label asp-for="Genre" class="col-md-2 control-label"></label>
8              <div class="col-md-10">
9                  <input asp-for="Genre" class="form-control" />
10                 <span asp-validation-for="Genre" class="text-danger" />
11             </div>
12         </div>
13         @*Markup removed for brevity.*@
14         <div class="form-group">

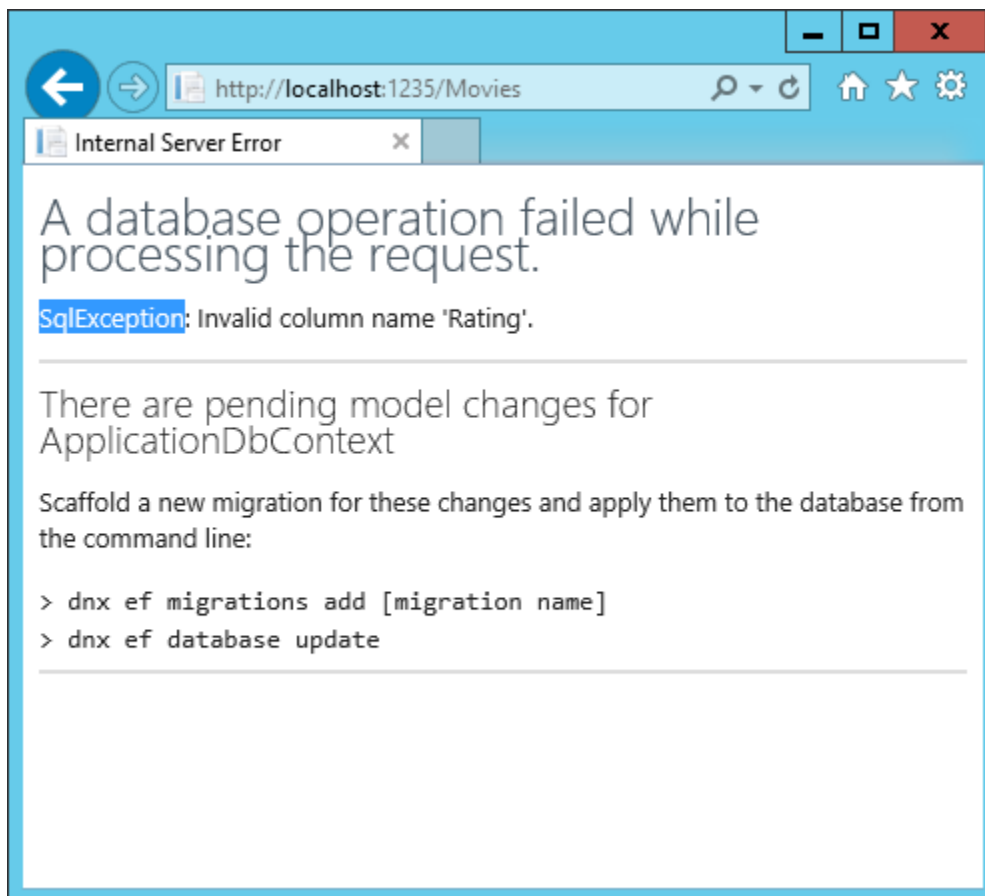
```

```

15     <label asp-for="Rating" class="col-md-2 control-label"></label>
16     <div class="col-md-10">
17         <input asp-for="Rating" class="form-control" />
18         <span asp-validation-for="Rating" class="text-danger" />
19     </div>
20 </div>
21 <div class="form-group">
22     <div class="col-md-offset-2 col-md-10">
23         <input type="submit" value="Create" class="btn btn-default" />
24     </div>
25 </div>
26 </div>
27 </form>

```

The app won't work until we update the DB to include the new field. If you run it now, you'll get the following `SqlException`:



You're seeing this error because the updated `Movie` model class in the application is now different than the schema of the `Movie` table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application.

2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, we'll use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each new `Movie`.

```
1      new Movie
2      {
3          Title = "Ghostbusters ",
4          ReleaseDate = DateTime.Parse("1984-3-13"),
5          Genre = "Comedy",
6          Rating = "G",
7          Price = 8.99M
8      },
```

Build the solution then open a command prompt. Enter the following commands:

```
dnx ef migrations add Rating
dnx ef database update
```

The `migrations add` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model. The name “Rating” is arbitrary and is used to name the migration file. It’s helpful to use a meaningful name for the migration step.

If you delete all the records in the DB, the initialize will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from SSOX.

Run the app and verify you can create/edit/display movies with a `Rating` field. You should also add the `Rating` field to the `Edit`, `Details`, and `Delete` view templates.

2.1.9 Adding Validation

By [Rick Anderson](#)

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is **DRY** (“Don’t Repeat Yourself”). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the application.

Let’s look at how you can take advantage of this validation support in the movie application.

Adding Validation Rules to the Movie Model

You'll begin by adding some validation logic to the `Movie` class.

Open the `Movie.cs` file. Notice the `System.ComponentModel.DataAnnotations` namespace does not contain `Microsoft.AspNet.Mvc.DataAnnotations`. `Microsoft.AspNet.Mvc.DataAnnotations` provides a built-in set of validation attributes that you can apply declaratively to any class or property. (It also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.)

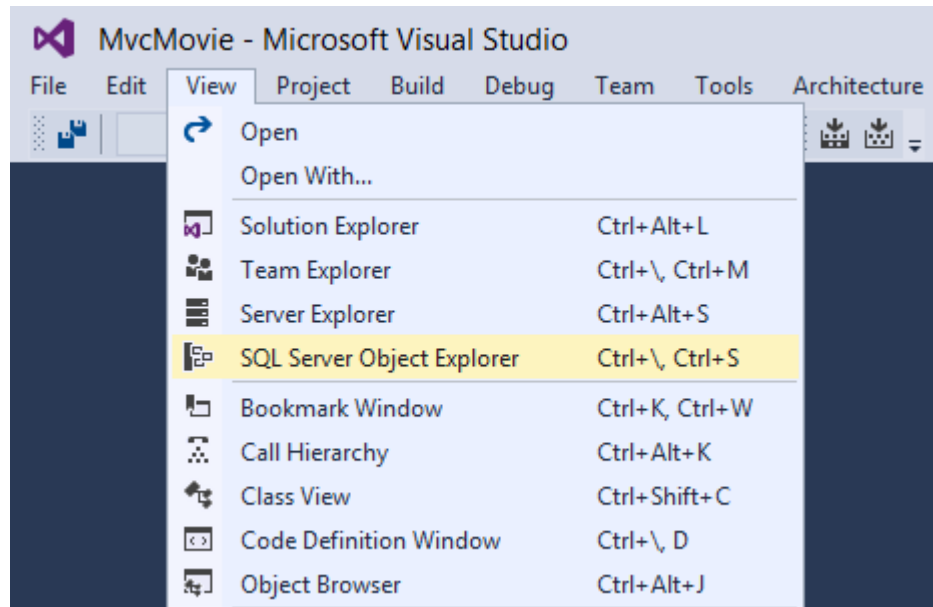
Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```

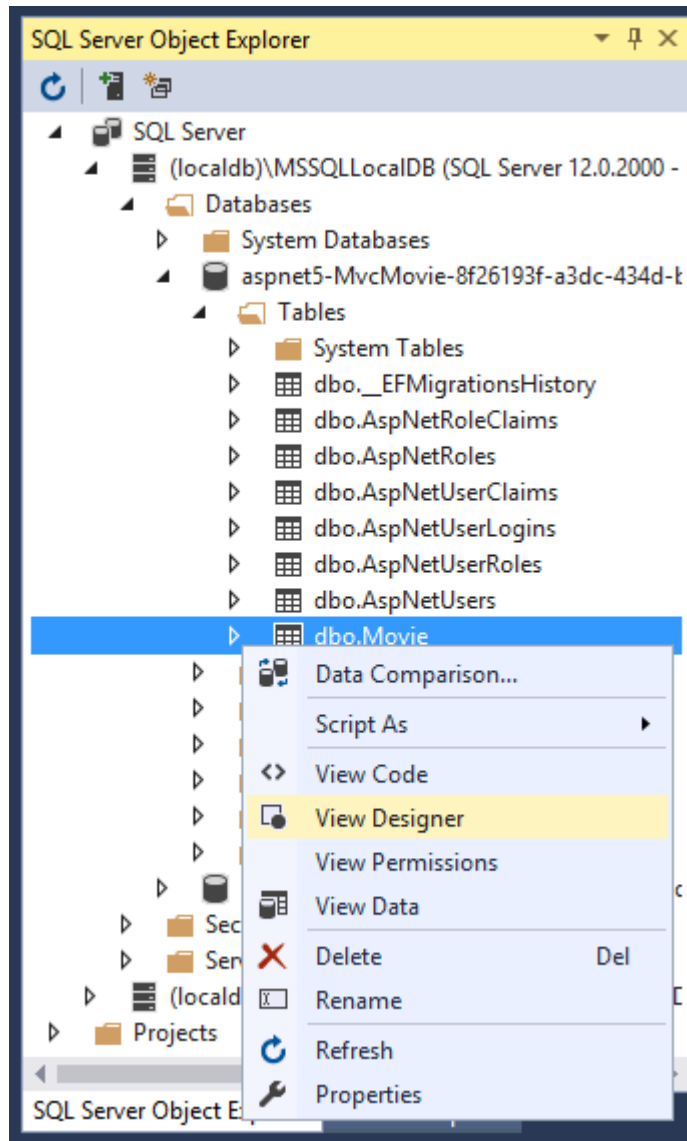
1 public class Movie
2 {
3     public int ID { get; set; }
4
5     [StringLength(60, MinimumLength = 3)]
6     public string Title { get; set; }
7
8     [Display(Name = "Release Date")]
9     [DataType(DataType.Date)]
10    public DateTime ReleaseDate { get; set; }
11
12    [RegularExpression(@"^[A-Z]+[a-zA-Z ' '-\s]*$")]
13    [Required]
14    [StringLength(30)]
15    public string Genre { get; set; }
16
17    [Range(1, 100)]
18    [DataType(DataType.Currency)]
19    public decimal Price { get; set; }
20
21    [RegularExpression(@"^[A-Z]+[a-zA-Z ' '-\s]*$")]
22    [StringLength(5)]
23    public string Rating { get; set; }
24 }
```

The `StringLength` attribute sets the maximum length of the string, and it sets this limitation on the database, therefore the database schema will change.

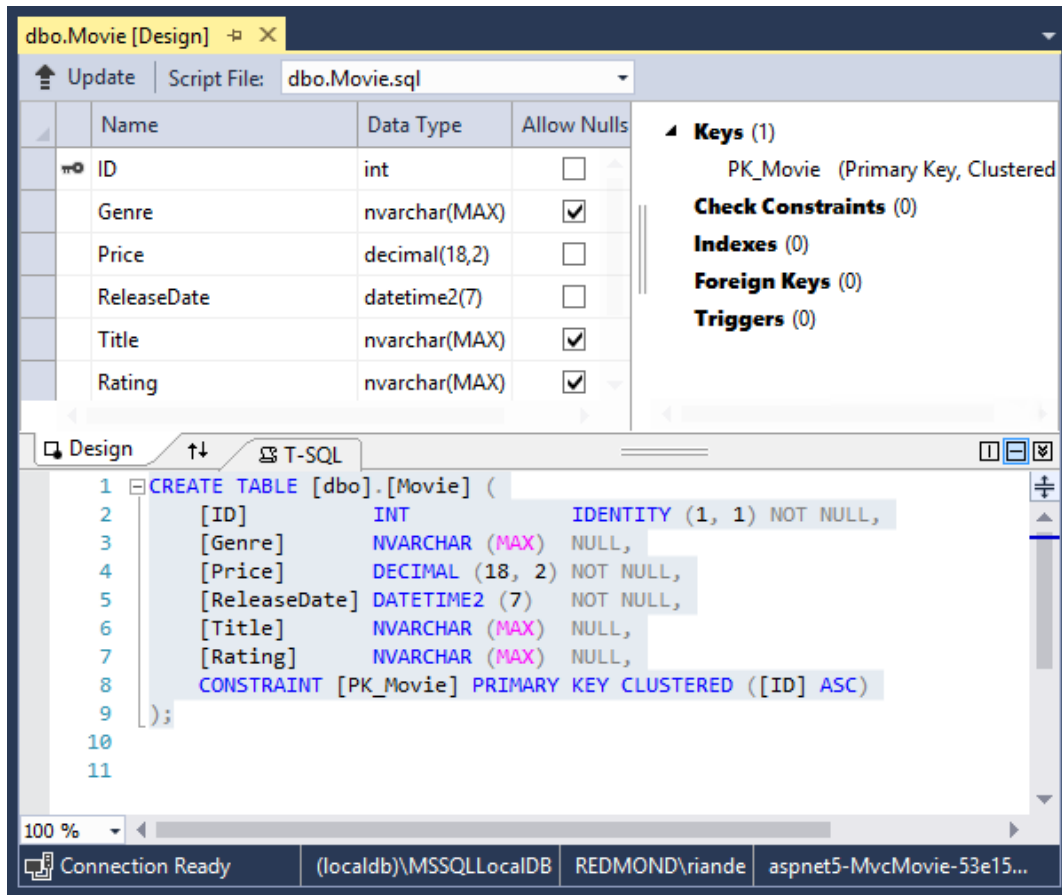
- From the **View** menu, open **SQL Server Object Explorer (SSOX)**.



- Right click on the `Movie` table > **View Designer**



The following image shows the table design and the T-SQL that can generate the table.



In the image above, you can see all the string fields are set to `NVARCHAR (MAX)`.

Build the project, open a command window and enter the following commands:

```

dnx ef migrations add DataAnnotations
dnx ef database update

```

Examine the Movie schema:

Name	Data Type	Allow Nulls	Default
ID	int	<input type="checkbox"/>	
Genre	nvarchar(30)	<input type="checkbox"/>	
Price	decimal(18,2)	<input type="checkbox"/>	
ReleaseDate	datetime2(7)	<input type="checkbox"/>	
Title	nvarchar(MAX)	<input checked="" type="checkbox"/>	
Rating	nvarchar(MAX)	<input checked="" type="checkbox"/>	
		<input type="checkbox"/>	

The string fields show the new length limits and Genre is no longer nullable.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` and `MinimumLength` attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The `RegularExpression` attribute is used to limit what characters can be input. In the code above, `Genre` and `Rating` must use only letters (white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw a `DbUpdateException` exception when the `SaveChanges` method is called, because several required `Movie` property values are missing.

```
Movie movie = new Movie();
movie.Title = "Gone with the Wind";
_context.Movie.Add(movie);
_context.SaveChanges();           // <= Will throw server side validation exception
```

The code above throws the following exception:

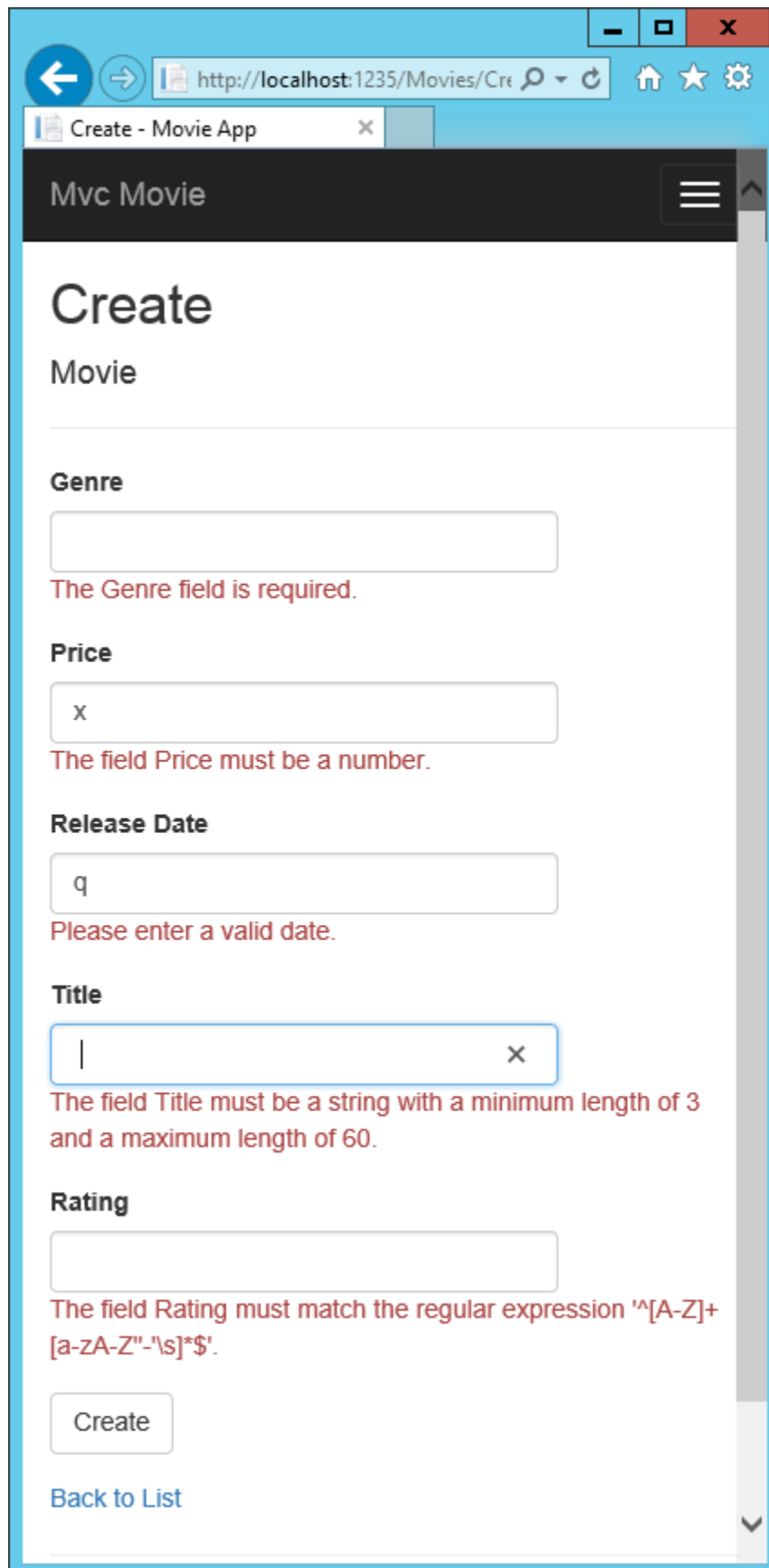
```
A database operation failed while processing the request.
DbUpdateException: An error occurred while updating the entries.
See the inner exception for details.
SqlException: Cannot insert the value NULL into column 'Genre', table 'aspnet5-MvcMovie-dbo.Movie';
Scolumn does not allow nulls. INSERT fails.
```

Having validation rules automatically enforced by ASP.NET helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in MVC

Run the app and navigate to the `Movies` controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



The screenshot shows a web browser window with the address bar at `http://localhost:1235/Movies/Crt`. The browser has a single tab titled 'Create - Movie App'. The page has a dark header with 'Mvc Movie' and a hamburger menu icon. The main content area is titled 'Create Movie' and contains a form with the following fields and errors:

- Genre**: An empty text input field. Below it, a red error message reads: 'The Genre field is required.'
- Price**: A text input field containing the character 'x'. Below it, a red error message reads: 'The field Price must be a number.'
- Release Date**: A text input field containing the character 'q'. Below it, a red error message reads: 'Please enter a valid date.'
- Title**: A text input field with a blue border and a clear button (X) on the right. Below it, a red error message reads: 'The field Title must be a string with a minimum length of 3 and a maximum length of 60.'
- Rating**: An empty text input field. Below it, a red error message reads: 'The field Rating must match the regular expression `^[A-Z]+[a-zA-Z'"'\s]*$`.'

At the bottom of the form, there is a 'Create' button and a 'Back to List' link.

Note: You may not be able to enter decimal points or commas in the `Price` field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See [Additional resources](#) for more information. For now, just enter whole numbers like 10.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the HTTP `Post` method, by using the [Fiddler tool](#), or the [F12 Developer tools](#).

How Validation Occurs in the Create View and Create Action Method

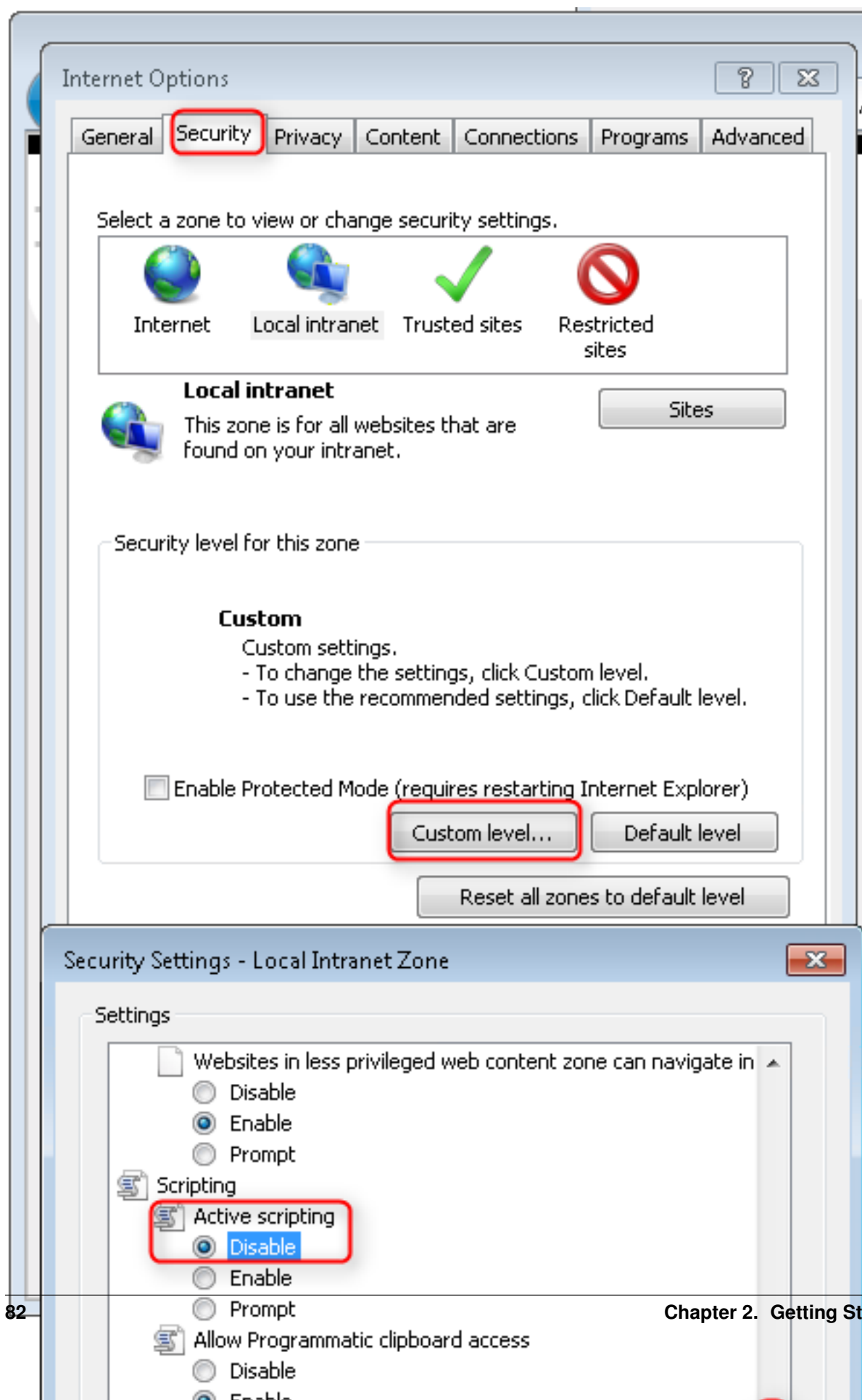
You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows the two `Create` methods.

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

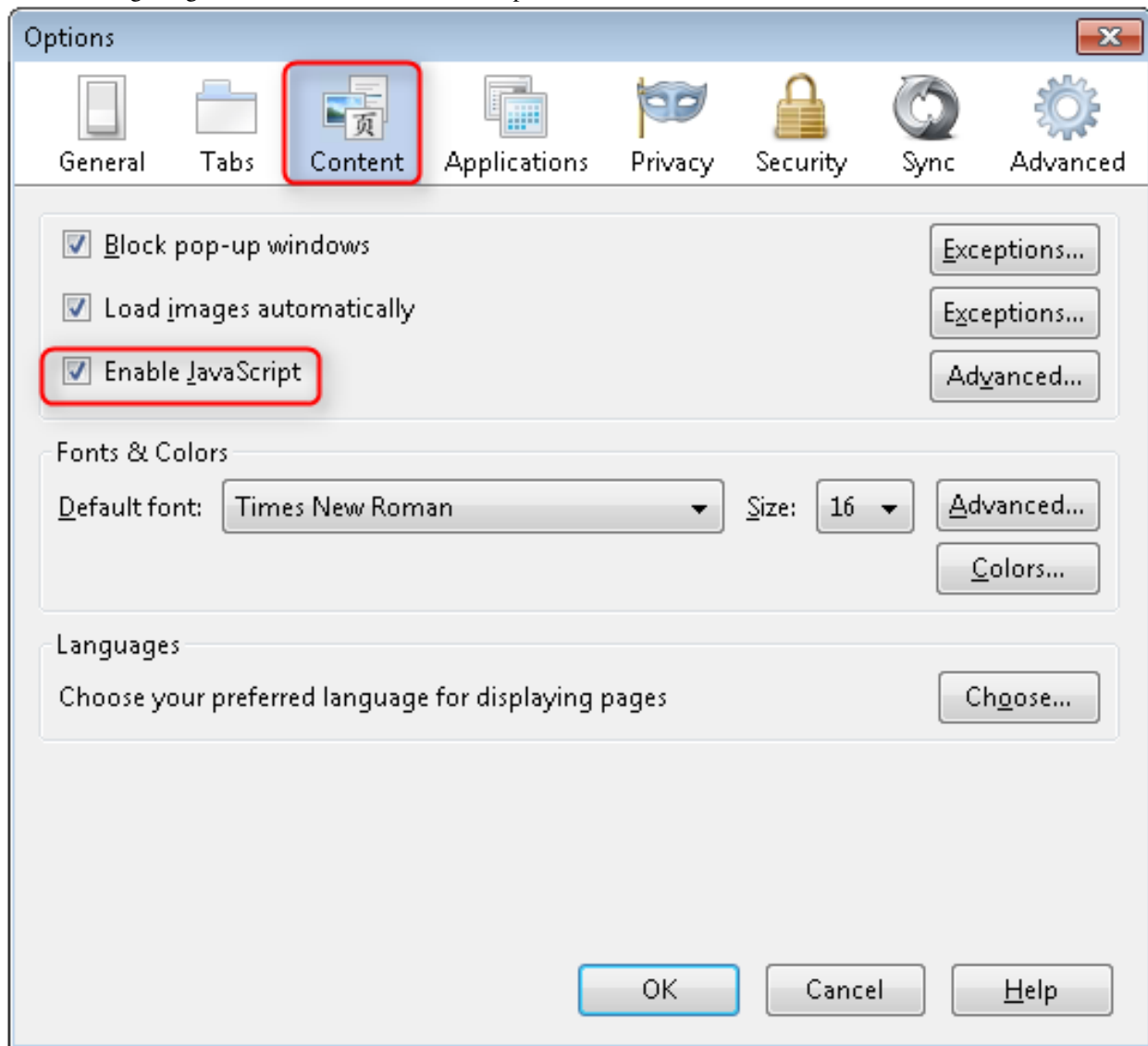
// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Create([Bind("ID,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Movie.Add(movie);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial `Create` form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `HttpPost` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form is not posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method calling `ModelState.IsValid` to check whether the movie has any validation errors.

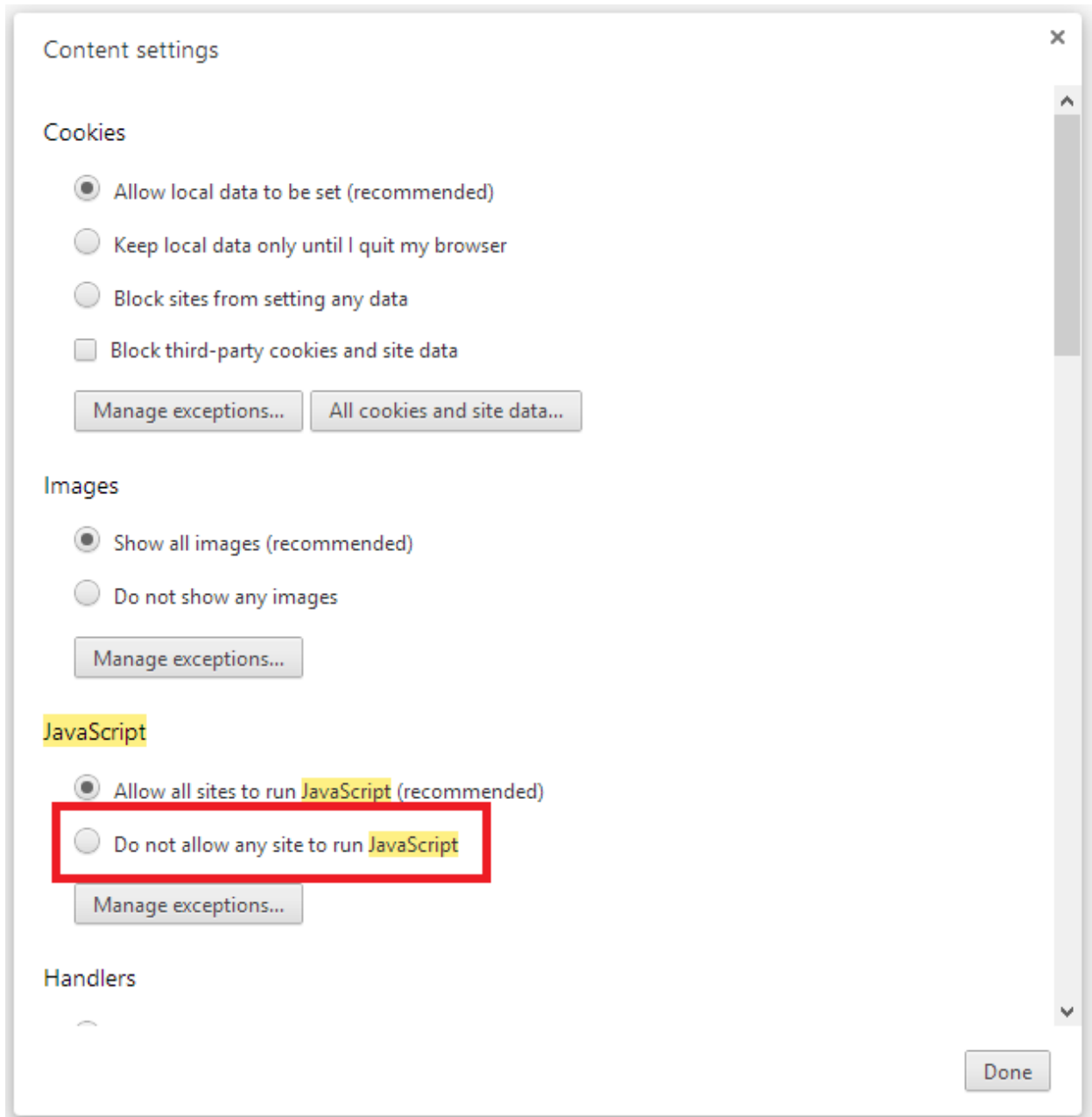
You can set a break point in the `[HttpPost]` `Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.



The following image shows how to disable JavaScript in the FireFox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.


```
// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public IActionResult Create([Bind("ID,Title,ReleaseDate,Genre")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Movie.Add(movie);
        _context.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

Below is portion of the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

```
1 <form asp-action="Create">
2   <div class="form-horizontal">
3     <h4>Movie</h4>
4     <hr />
5     <div asp-validation-summary="ValidationSummary.ModelOnly" class="text-danger"></div>
6     <div class="form-group">
7       <label asp-for="Genre" class="col-md-2 control-label"></label>
8       <div class="col-md-10">
9         <input asp-for="Genre" class="form-control" />
10        <span asp-validation-for="Genre" class="text-danger" />
11      </div>
12    </div>
13    @*Markup removed for brevity.*@
14    <div class="form-group">
15      <label asp-for="Rating" class="col-md-2 control-label"></label>
16      <div class="col-md-10">
17        <input asp-for="Rating" class="form-control" />
18        <span asp-validation-for="Rating" class="text-danger" />
19      </div>
20    </div>
21    <div class="form-group">
22      <div class="col-md-offset-2 col-md-10">
23        <input type="submit" value="Create" class="btn btn-default" />
24      </div>
25    </div>
26  </div>
27 </form>
```

The [Input Tag Helper](#) consumes the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](#) displays a validation message.

What's really nice about this approach is that neither the controller nor the *Create* view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the *Movie* class. These same validation rules are automatically applied to the *Edit* view and any other views templates you might create that edit your model.

If you want to change the validation logic later, you can do so in exactly one place by adding validation attributes to the model (in this example, the *Movie* class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that that you'll be fully

honoring the DRY principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the *Movie* class. The *System.ComponentModel.DataAnnotations* namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a *DataType* enumeration value to the release date and to the price fields. The following code shows the *ReleaseDate* and *Price* properties with the appropriate *DataType* attribute.

```
1 [DataType(DataType.Date)]
2 public DateTime ReleaseDate { get; set; }
3
4 [DataType(DataType.Currency)]
5 public decimal Price { get; set; }
```

The *DataType* attributes only provide hints for the view engine to format the data (and supply attributes such as `<a>` for URL's and `` for email. You can use the *RegularExpression* attribute to validate the format of the data. The *DataType* attribute is used to specify a data type that is more specific than the database intrinsic type, they are not validation attributes. In this case we only want to keep track of the date, not the time. The *DataType* Enumeration provides for many data types, such as *Date*, *Time*, *PhoneNumber*, *Currency*, *EmailAddress* and more. The *DataType* attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for *DataType.EmailAddress*, and a date selector can be provided for *DataType.Date* in browsers that support HTML5. The *DataType* attributes emits HTML 5 data- (pronounced data dash) attributes that HTML 5 browsers can understand. The *DataType* attributes do **not** provide any validation.

DataType.Date does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's *CultureInfo*.

The *DisplayFormat* attribute is used to explicitly specify the date format:

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

The *ApplyFormatInEditMode* setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably do not want the currency symbol in the text box for editing.)

You can use the *DisplayFormat* attribute by itself, but it's generally a good idea to use the *DataType* attribute alone. The *DataType* attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with *DisplayFormat*:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your *locale*
- The *DataType* attribute can enable MVC to choose the right field template to render the data (the *DisplayFormat* if used by itself uses the string template). For more information, see Brad Wilson's [ASP.NET MVC 2 Templates](#). (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

Note: jQuery validation does not work with the *Range* attribute and *DateTime*. For example, the following code will always display a client side validation error, even when the date is in the specified range:

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

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
1 public class Movie
2 {
3     public int ID { get; set; }
4
5     [StringLength(60, MinimumLength = 3)]
6     public string Title { get; set; }
7
8     [Display(Name = "Release Date"), DataType(DataType.Date)]
9     public DateTime ReleaseDate { get; set; }
10
11     [RegularExpression(@"^[A-Z]+[a-zA-Z' '-\s]*$"), Required, StringLength(30)]
12     public string Genre { get; set; }
13
14     [Range(1, 100), DataType(DataType.Currency)]
15     public decimal Price { get; set; }
16
17     [RegularExpression(@"^[A-Z]+[a-zA-Z' '-\s]*$"), StringLength(5)]
18     public string Rating { get; set; }
19 }
```

In the next part of the series, we'll review the application and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)

2.1.10 Examining the Details and Delete methods

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

2.2 Building Your First Web API with MVC 6

By [Mike Wasson](#) and [Rick Anderson](#)

HTTP is not just for serving up web pages. It's also a powerful platform for building APIs that expose services and data. HTTP is simple, flexible, and ubiquitous. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop apps.

In this tutorial, you'll build a simple web API for managing a list of "to-do" items. You won't build any UI in this tutorial.

Previous versions of ASP.NET included the Web API framework for creating web APIs. In ASP.NET 5, this functionality has been merged into the MVC 6 framework. Unifying the two frameworks makes it simpler to build apps that include both UI (HTML) and APIs, because now they share the same code base and pipeline.

Note: If you are porting an existing Web API app to MVC 6, see [Migrating From ASP.NET Web API 2 to MVC 6](#)

In this article:

- [Overview](#)
- [Install Fiddler](#)
- [Create the project](#)
- [Add a model class](#)
- [Add a repository class](#)
- [Register the repository](#)
- [Add a controller](#)
- [Getting to-do items](#)
- [Use Fiddler to call the API](#)
- [Implement the other CRUD operations](#)
- [Next steps](#)

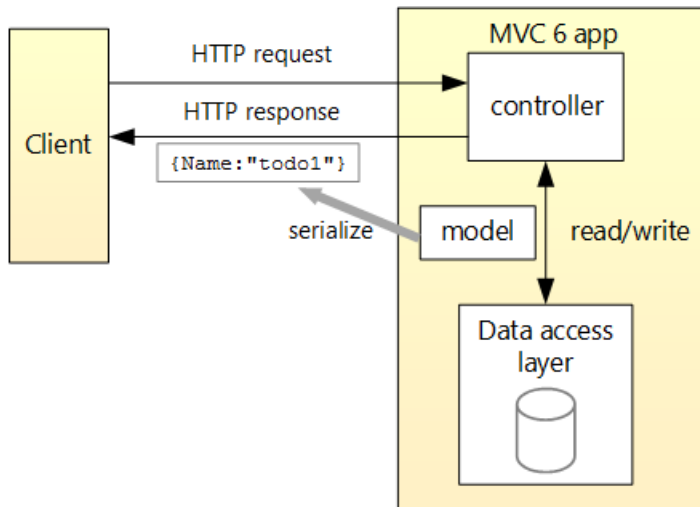
You can browse the source code for the sample app on [GitHub](#).

2.2.1 Overview

Here is the API that you'll create:

API	Description	Request body	Response body
GET /api/todo	Get all to-do items	None	Array of to-do items
GET /api/todo/{id}	Get an item by ID	None	To-do item
POST /api/todo	Add a new item	To-do item	To-do item
PUT /api/todo/{id}	Update an existing item	To-do item	None
DELETE /api/todo/{id}	Delete an item.	None	None

The following diagram show the basic design of the app.



- The client is whatever consumes the web API (browser, mobile app, and so forth). We aren't writing a client in this tutorial.
- A *model* is an object that represents the data in your application. In this case, the only model is a to-do item. Models are represented as simple C# classes (POCOs).
- A *controller* is an object that handles HTTP requests and creates the HTTP response. This app will have a single controller.
- To keep the tutorial simple and focused on MVC 6, the app doesn't use a database. Instead, it just keeps to-do items in memory. But we'll still include a (trivial) data access layer, to illustrate the separation between the web API and the data layer. For a tutorial that uses a database, see [Building your first MVC 6 application](#).

2.2.2 Install Fiddler

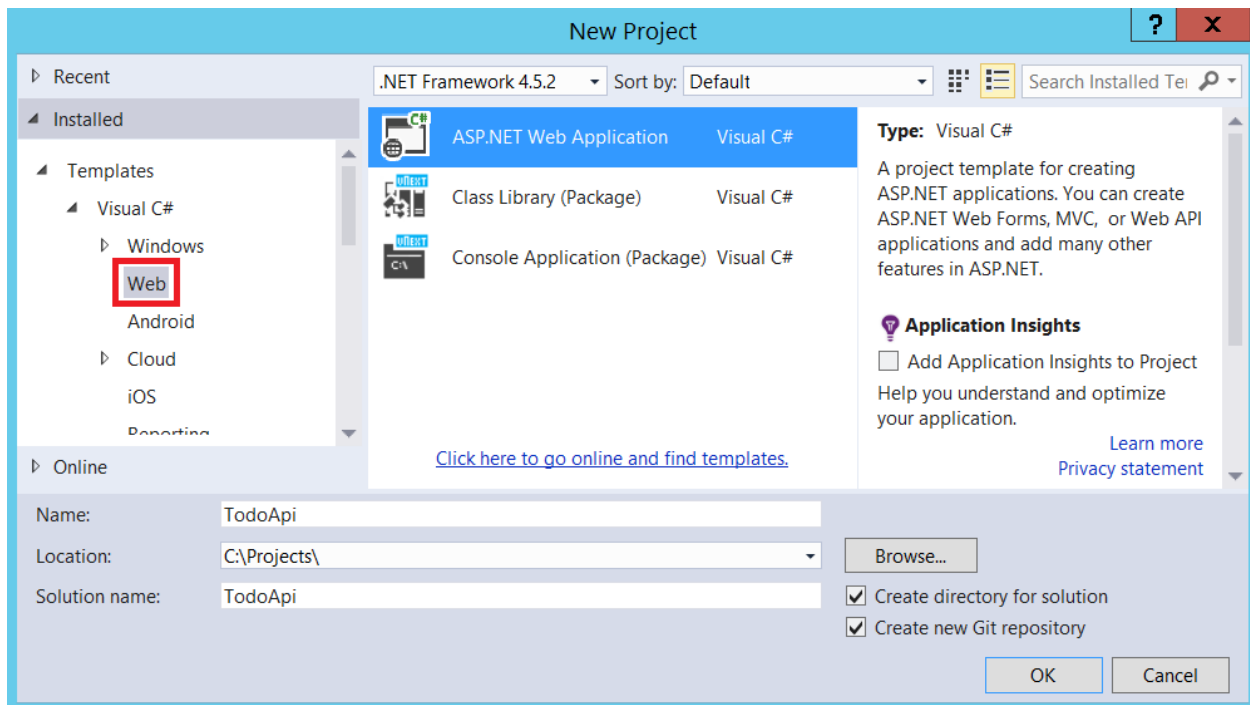
This step is optional but recommended.

Because we're not building a client, we need a way to call the API. In this tutorial, I'll show that by using [Fiddler](#). Fiddler is a web debugging tool that lets you compose HTTP requests and view the raw HTTP responses. Fiddler lets you make direct HTTP requests to the API as we develop the app.

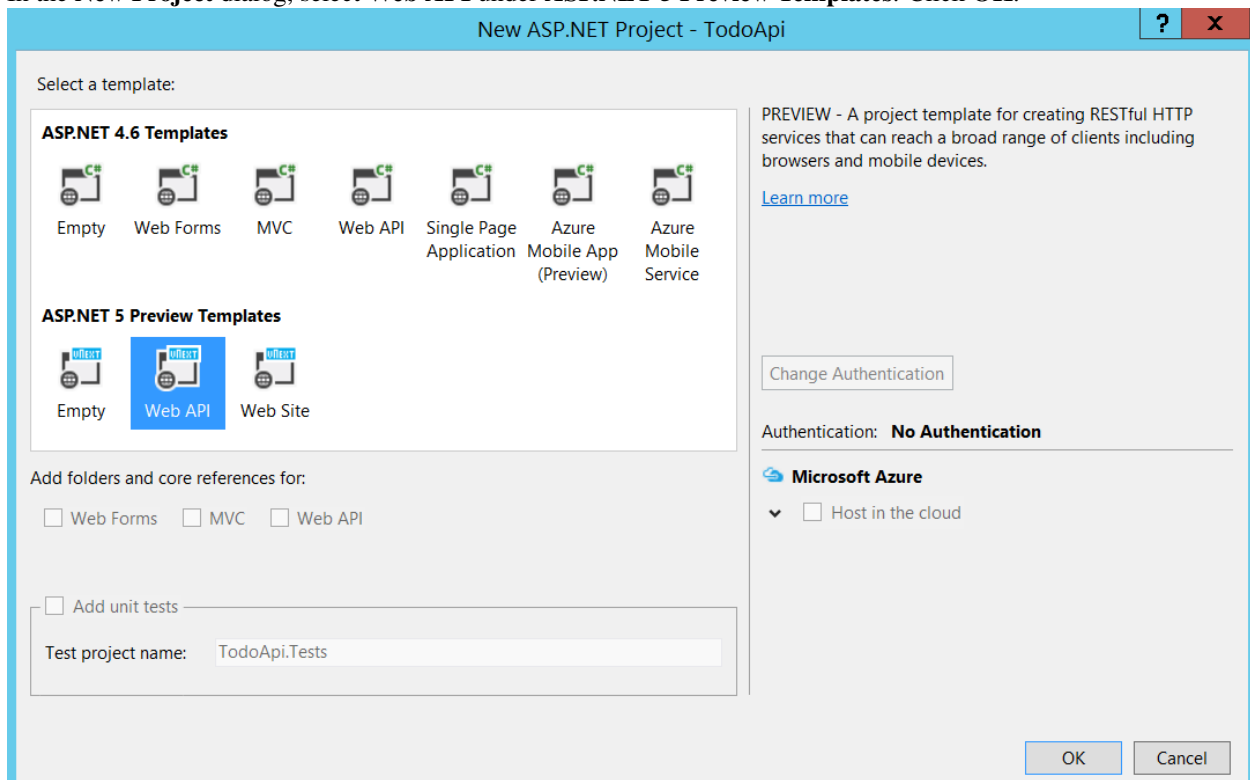
2.2.3 Create the project

Start Visual Studio 2015. From the **File** menu, select **New > Project**.

Select the **ASP.NET Web Application** project template. Name the project `ToDoApi` and click **OK**.



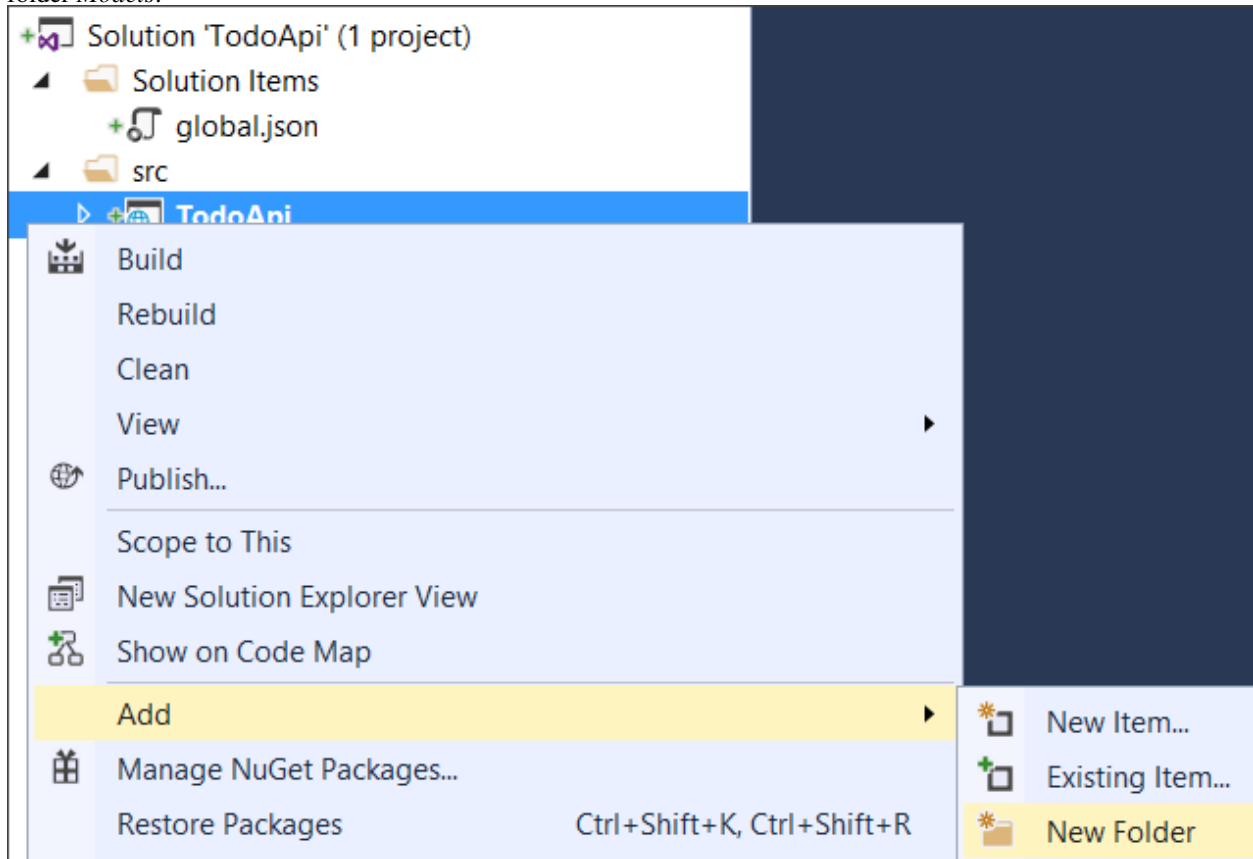
In the **New Project** dialog, select **Web API** under **ASP.NET 5 Preview Templates**. Click **OK**.



2.2.4 Add a model class

A model is an object that represents the data in your application. In this case, the only model is a to-do item.

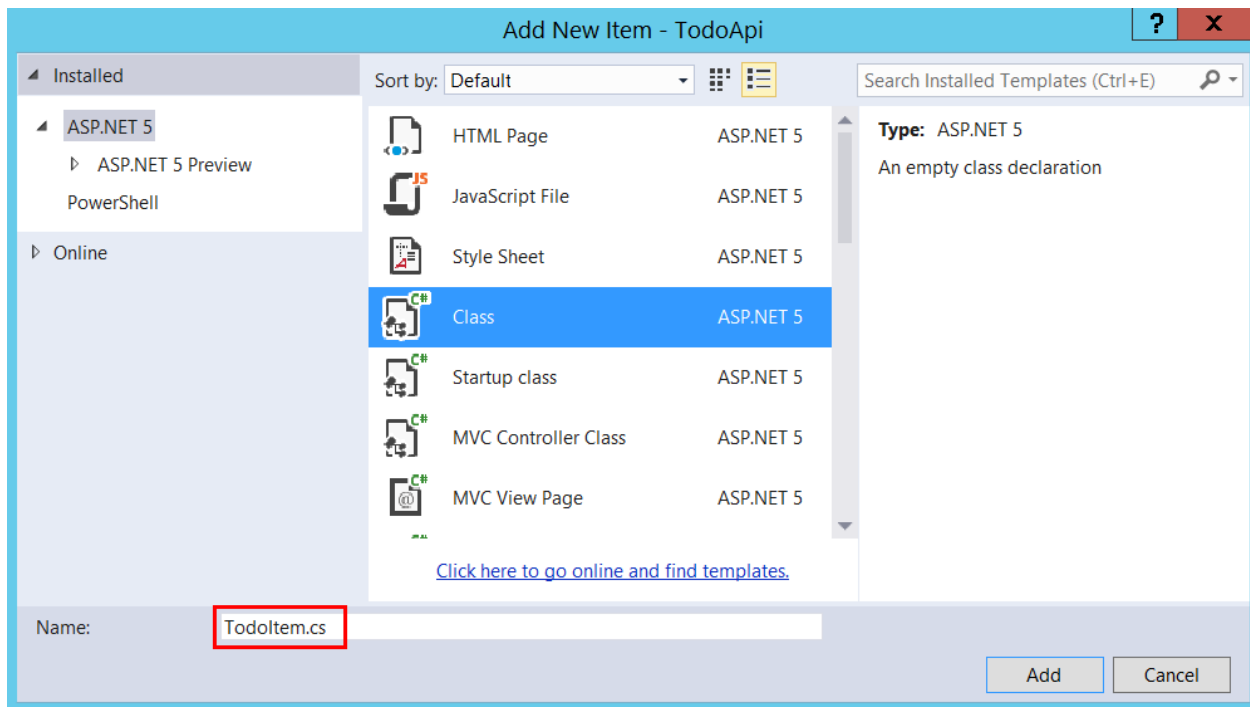
Add a folder named “Models”. In Solution Explorer, right-click the project. Select **Add > New Folder**. Name the folder *Models*.



Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

Next, add a `TodoItem` class. Right-click the *Models* folder and select **Add > New Item**.

In the **Add New Item** dialog, select the **Class** template. Name the class `TodoItem` and click **OK**.



Replace the generated code with:

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

2.2.5 Add a repository class

A *repository* is an object that encapsulates the data layer, and contains logic for retrieving data and mapping it to an entity model. Even though the example app doesn't use a database, it's useful to see how you can inject a repository into your controllers. Create the repository code in the *Models* folder.

Start by defining a repository interface named *ITodoRepository*. Use the class template (**Add New Item > Class**).

```
using System.Collections.Generic;

namespace TodoApi.Models
{
    public interface ITodoRepository
    {
        void Add(TodoItem item);
        IEnumerable<TodoItem> GetAll();
        TodoItem Find(string key);
        TodoItem Remove(string key);
        void Update(TodoItem item);
    }
}
```


This interface defines basic CRUD operations. In practice, you might have domain-specific methods.

Next, add a `TodoRepository` class that implements `ITodoRepository`:

```
using System;
using System.Collections.Generic;
using System.Collections.Concurrent;

namespace TodoApi.Models
{
    public class TodoRepository : ITodoRepository
    {
        static ConcurrentDictionary<string, TodoItem> _todos = new ConcurrentDictionary<string, TodoItem>();

        public TodoRepository()
        {
            Add(new TodoItem { Name = "Item1" });
        }

        public IEnumerable<TodoItem> GetAll()
        {
            return _todos.Values;
        }

        public void Add(TodoItem item)
        {
            item.Key = Guid.NewGuid().ToString();
            _todos[item.Key] = item;
        }

        public TodoItem Find(string key)
        {
            TodoItem item;
            _todos.TryGetValue(key, out item);
            return item;
        }

        public TodoItem Remove(string key)
        {
            TodoItem item;
            _todos.TryGetValue(key, out item);
            _todos.TryRemove(key, out item);
            return item;
        }

        public void Update(TodoItem item)
        {
            _todos[item.Key] = item;
        }
    }
}
```

Build the app to verify you don't have any errors.

2.2.6 Register the repository

By defining a repository interface, we can decouple the repository class from the MVC controller that uses it. Instead of newing up a `TodoRepository` inside the controller, we will inject an `ITodoRepository`, using the ASP.NET

5 dependency injection (DI) container.

This approach makes it easier to unit test your controllers. Unit tests should inject a mock or stub version of `ITodoRepository`. That way, the test narrowly targets the controller logic and not the data access layer.

In order to inject the repository into the controller, we need to register it with the DI container. Open the *Startup.cs* file. Add the following using directive:

```
using TodoApi.Models;
```

In the `ConfigureServices` method, add the highlighted code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    // Add our repository type
    services.AddSingleton<ITodoRepository, TodoRepository>();
}
```

2.2.7 Add a controller

In Solution Explorer, right-click the *Controllers* folder. Select **Add > New Item**. In the **Add New Item** dialog, select the **Web API Controller Class** template. Name the class `TodoController`.

Replace the generated code with the following:

```
using System.Collections.Generic;
using Microsoft.AspNet.Mvc;
using TodoApi.Models;

namespace SimpleApi.Controllers
{
    [Route("api/[controller]")]
    public class TodoController : Controller
    {
        [FromServices]
        public ITodoRepository TodoItems { get; set; }
    }
}
```

This defines an empty controller class. In the next sections, we'll add methods to implement the API. The `[FromServices]` attribute tells MVC to inject the `ITodoRepository` that we registered in the *Startup* class.

Delete the *ValuesController.cs* file from the *Controllers* folder. The project template adds it as an example controller, but we don't need it.

2.2.8 Getting to-do items

To get to-do items, add the following methods to the `TodoController` class.

```
[HttpGet]
public IEnumerable<TodoItem> GetAll()
{
    return TodoItems.GetAll();
}

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
```

```
{
    var item = TodoItems.Find(id);
    if (item == null)
    {
        return HttpNotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

Here is an example HTTP response for the `GetAll` method:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/10.0
Date: Thu, 18 Jun 2015 20:51:10 GMT
Content-Length: 82

[{"Key": "4f67d7c5-a2a9-4aae-b030-16003dd829ae", "Name": "Item1", "IsComplete": false}]
```

Later in the tutorial I'll show how you can view the HTTP response using the Fiddler tool.

Routing and URL paths

The `[HttpGet]` attribute specifies that these are HTTP GET methods. The URL path for each method is constructed as follows:

- Take the template string in the controller's route attribute, `[Route("api/[controller]")]`
- Replace “[Controller]” with the name of the controller, which is the controller class name minus the “Controller” suffix. For this sample the name of the controller is “todo” (case insensitive). For this sample, the controller class name is `TodoController` and the root name is “todo”. ASP.NET MVC is not case sensitive.
- If the `[HttpGet]` attribute also has a template string, append that to the path. This sample doesn't use a template string.

For the `GetById` method, “{id}” is a placeholder variable. In the actual HTTP request, the client will use the ID of the `todo` item. At runtime, when MVC invokes `GetById`, it assigns the value of “{id}” in the URL the method's `id` parameter.

Open the `src\TodoApi\Properties\launchSettings.json` file and replace the `launchUrl` value to use the `todo` controller. That change will cause IIS Express to call the `todo` controller when the project is started.

```
{
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/todo",
      "environmentVariables": {
        "ASPNET_ENV": "Development"
      }
    }
  }
}
```

To learn more about request routing in MVC 6, see [Routing to Controller Actions](#).

Return values

The `GetAll` method returns a CLR object. MVC automatically serializes the object to [JSON](#) and writes the JSON into the body of the response message. The response code for this method is 200, assuming there are no unhandled exceptions. (Unhandled exceptions are translated into 5xx errors.)

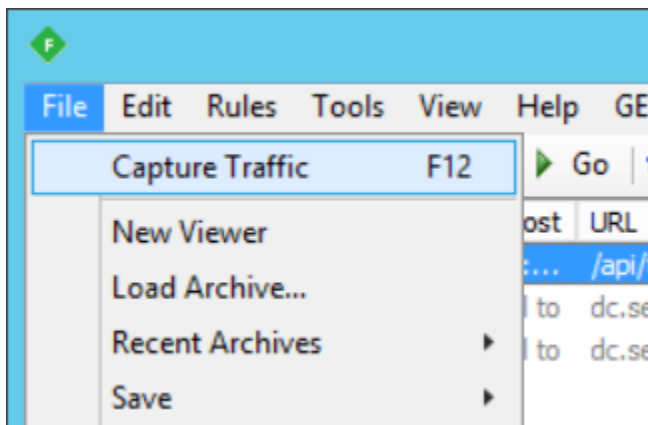
In contrast, the `GetById` method returns the more general `ActionResult` type, which represents a generic result type. That's because `GetById` has two different return types:

- If no item matches the requested ID, the method returns a 404 error. This is done by returning `HttpNotFound`.
- Otherwise, the method returns 200 with a JSON response body. This is done by returning an [ObjectResult](#).

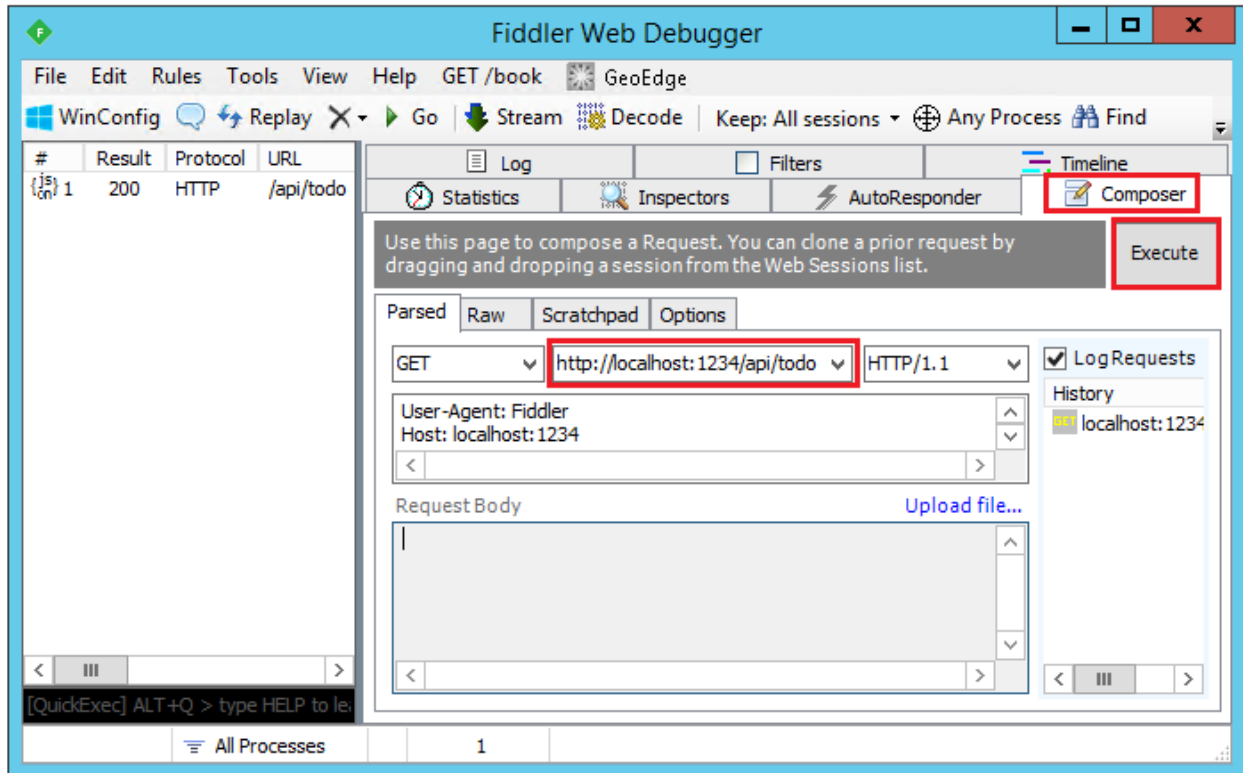
2.2.9 Use Fiddler to call the API

This step is optional, but it's useful to see the raw HTTP responses from the web API. In Visual Studio, press `^F5` to launch the app. Visual Studio launches a browser and navigates to `http://localhost:port/api/todo`, where *port* is a randomly chosen port number. If you're using Chrome, Edge or Firefox, the *todo* data will be displayed. If you're using IE, IE will prompt to you open or save the *todo.json* file.

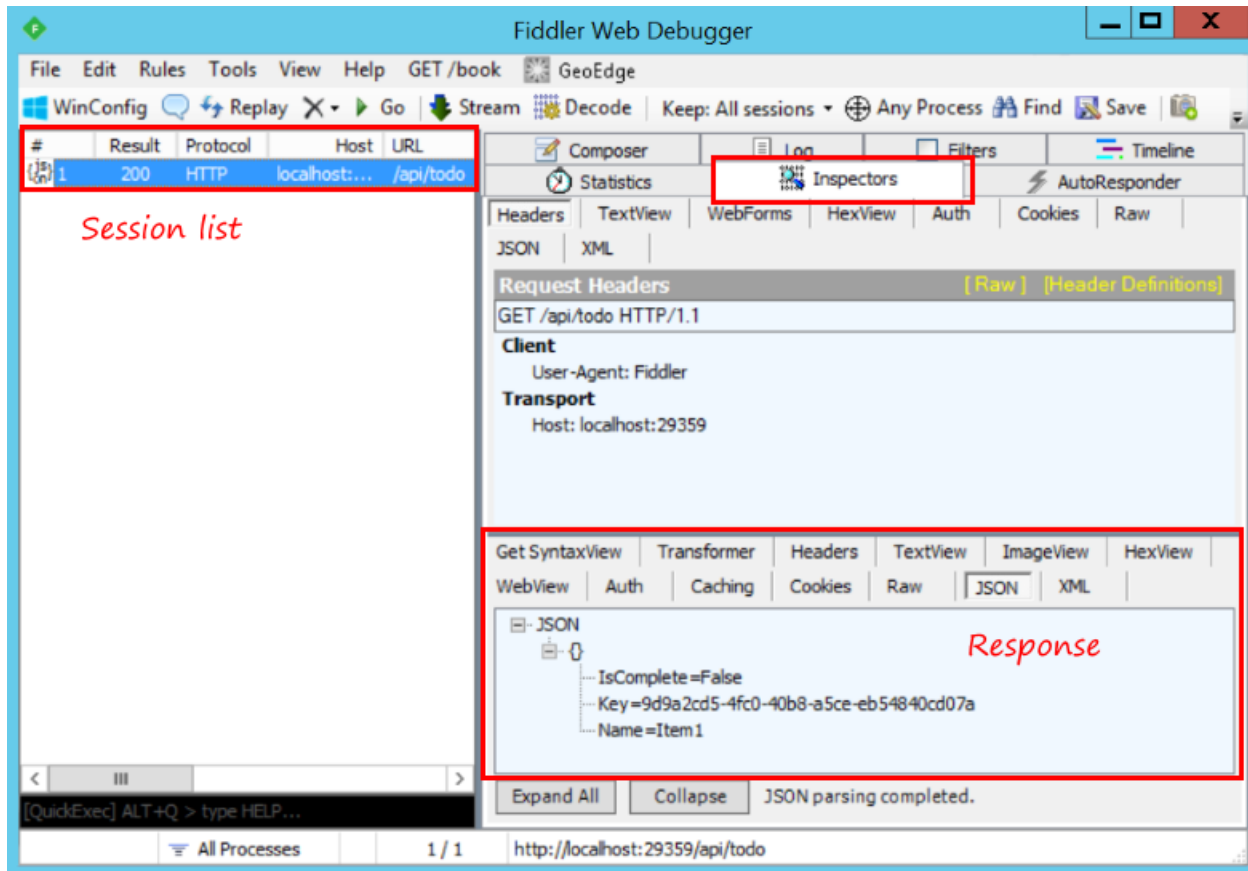
Launch Fiddler. From the **File** menu, uncheck the **Capture Traffic** option. This turns off capturing HTTP traffic.



Select the **Composer** page. In the **Parsed** tab, type `http://localhost:port/api/todo`, where *port* is the port number. Click **Execute** to send the request.



The result appears in the sessions list. The response code should be 200. Use the **Inspectors** tab to view the content of the response, including the response body.



2.2.10 Implement the other CRUD operations

The last step is to add Create, Update, and Delete methods to the controller. These methods are variations on a theme, so I'll just show the code and highlight the main differences.

Create

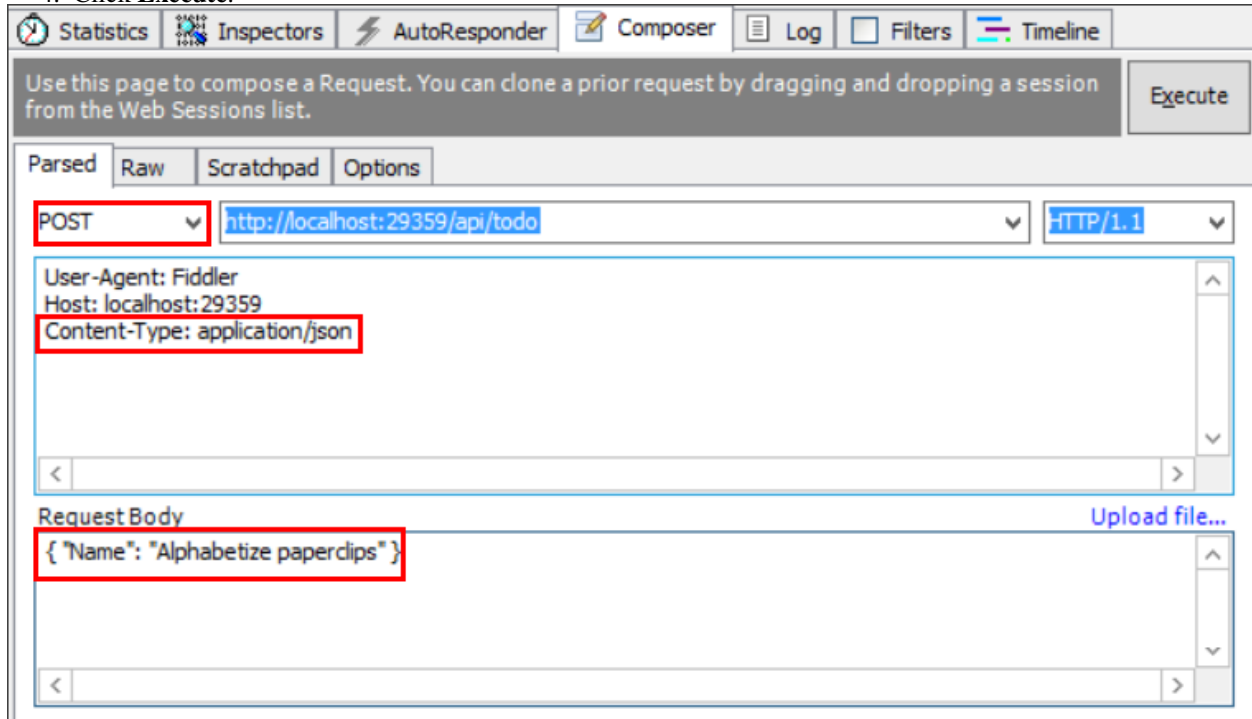
```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return HttpBadRequest();
    }
    TodoItems.Add(item);
    return CreatedAtRoute("GetTodo", new { controller = "Todo", id = item.Key }, item);
}
```

This is an HTTP POST method, indicated by the `[HttpPost]` attribute. The `[FromBody]` attribute tells MVC to get the value of the to-do item from the body of the HTTP request.

The `CreatedAtRoute` method returns a 201 response, which is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a Location header to the response. The Location header specifies the URI of the newly created to-do item. See [10.2.2 201 Created](#).

We can use Fiddler to send a Create request:

1. In the **Composer** page, select POST from the drop-down.
2. In the request headers text box, add a `Content-Type` header with the value `application/json`. Fiddler automatically adds the `Content-Length` header.
3. In the request body text box, enter the following: `{"Name": "<your to-do item>"}`
4. Click **Execute**.



Here is an example HTTP session. Use the **Raw** tab to see the session data in this format.

Request:

```
POST http://localhost:29359/api/todo HTTP/1.1
User-Agent: Fiddler
Host: localhost:29359
Content-Type: application/json
Content-Length: 33

{"Name": "Alphabetize paperclips"}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: http://localhost:29359/api/ToDo/8fa2154d-f862-41f8-a5e5-a9a3faba0233
Server: Microsoft-IIS/10.0
Date: Thu, 18 Jun 2015 20:51:55 GMT
Content-Length: 97

{"Key": "8fa2154d-f862-41f8-a5e5-a9a3faba0233", "Name": "Alphabetize paperclips", "IsComplete": false}
```

Update

```
[HttpPut("{id}")]
public IActionResult Update(string id, [FromBody] TodoItem item)
{
    if (item == null || item.Key != id)
    {
        return BadRequest();
    }

    var todo = TodoItems.Find(id);
    if (todo == null)
    {
        return HttpNotFound();
    }

    TodoItems.Update(item);
    return new NoContentResult();
}
```

Update is similar to Create, but uses HTTP PUT. The response is 204 (No Content). According to the HTTP spec, a PUT request requires the client to send the entire updated entity, not just the deltas. To support partial updates, use HTTP PATCH.

The screenshot shows the Fiddler web debugging tool interface. At the top, there are tabs for 'Parsed', 'Raw', 'Scratchpad', and 'Options'. Below these, the 'PUT' method is selected, and the URL is 'http://localhost:1234/api/todo/707e881d-ca69-40ef-b6a1-3e2fbfa'. The HTTP version is set to 'HTTP/1.1'. The 'User-Agent' is 'Fiddler', 'Host' is 'localhost:1234', 'Content-Type' is 'application/json', and 'Content-Length' is '79'. The 'Request Body' is shown as a JSON object: `{\"Key\": \"707e881d-ca69-40ef-b6a1-3e2fbfa31ba6\", \"Name\": \"Item1\", \"IsComplete\": true}`. There is an 'Upload file...' link next to the request body field.

Delete

```
[HttpDelete("{id}")]
public void Delete(string id)
{
    TodoItems.Remove(id);
}
```


The void return type returns a 204 (No Content) response. That means the client receives a 204 even if the item has already been deleted, or never existed. There are two ways to think about a request to delete a non-existent resource:

- “Delete” means “delete an existing item”, and the item doesn’t exist, so return 404.
- “Delete” means “ensure the item is not in the collection.” The item is already not in the collection, so return a 204.

Either approach is reasonable. If you return 404, the client will need to handle that case.

The screenshot shows a web client interface with four tabs: "Parsed", "Raw", "Scratchpad", and "Options". The "Parsed" tab is active. At the top, there are three dropdown menus: the first is set to "DELETE", the second to "http://localhost:1234/api/todo/707e881d-ca69-40ef-b6a1-3e2fbfa", and the third to "HTTP/1.1". Below these, a text area contains the headers: "User-Agent: Fiddler", "Host: localhost:1234", and "Content-Type: application/json". To the right of this text area are up and down arrow buttons. Below the headers is a "Request Body" section with a large, empty text area and a "Upload file..." link to its right. At the bottom of the "Request Body" section are left and right arrow buttons.

2.2.11 Next steps

To learn about creating a backend for a native mobile app, see [Creating Backend Services for Native Mobile Applications](#).

For information about deploying your API, see [Publishing and Deployment](#).

3.1 Music Store Tutorial

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

3.2 Creating Backend Services for Native Mobile Applications

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

4.1 Model Binding

By Rachel Appel

In this article

- *Introduction to model binding*
- *How model binding works*
- *Customize model binding behavior with attributes*
- *Binding formatted data from the request body*

4.1.1 Introduction to model binding

Model binding in MVC maps data from HTTP requests to action method parameters. The parameters may be simple types such as strings, integers, or floats, or they may be complex types. This is a great feature of MVC because mapping incoming data to a counterpart is an often repeated scenario, regardless of size or complexity of the data. MVC solves this problem by abstracting binding away so developers don't have to keep rewriting a slightly different version of that same code in every app. Writing your own text to type converter code is tedious, and error prone.

4.1.2 How model binding works

When MVC receives an HTTP request, it routes it to a specific action method of a controller. It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action method's parameters. For example, consider the following URL:

http://contoso.com/movies/edit/2

Since the route template looks like this, `{controller=Home}/{action=Index}/{id?}`, `movies/edit/2` routes to the `Movies` controller, and its `Edit` action method. It also accepts an optional parameter called `id`. The code for the action method should look something like this:

```
1 public IActionResult Edit(int? id)
```

Note: The strings in the URL route are not case sensitive.

MVC will try to bind request data to the action parameters by name. MVC will look for values for each parameter using the parameter name and the names of its public settable properties. In the above example, the only action

parameter is named `id`, which MVC binds to the value with the same name in the route values. In addition to route values MVC will bind data from various parts of the request and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:

1. `Form values`: These are form values that go in the HTTP request using the POST method. (including jQuery POST requests).
2. `Route values`: The set of route values provided by [routing](#).
3. `Query strings`: The query string part of the URI.

Note: Form values, route data, and query strings are all stored as name-value pairs.

Since model binding asked for a key named `id` and there is nothing named `id` in the form values, it moved on to the route values looking for that key. In our example, it's a match. Binding happens, and the value is converted to the integer 2. The same request using `Edit(string id)` would convert to the string "2".

So far the example uses simple types. In MVC simple types are any .NET primitive type or type with a string type converter. If the action method's parameter were a class such as the `Movie` type, which contains both simple and complex types as properties, MVC's model binding will still handle it nicely. It uses reflection and recursion to traverse the properties of complex types looking for matches. Model binding looks for the pattern `parameter_name.property_name` to bind values to properties. If it doesn't find matching values of this form, it will attempt to bind using just the property name. For those types such as `Collection` types, model binding looks for matches to `parameter_name[index]` or just `[index]`. Model binding treats `Dictionary` types similarly, asking for `parameter_name[key]` or just `[key]`, as long as they keys are simple types. Keys that are supported match the field names HTML and tag helpers generated for the same model type. This enables round-tripping values so that the form fields remain filled with the user's input for their convenience, for example, when bound data from a create or edit did not pass validation.

In order for binding to happen, members of classes must be public, writable properties containing a default public constructor. Public fields are not bound. Properties of type `IEnumerable` or array must be settable and will be populated with an array. Collection properties of type `ICollection<T>` can be read-only.

When a parameter is bound, model binding stops looking for values with that name and it moves on to bind the next parameter. If binding fails, MVC does not throw an error. You can query for model state errors by checking the `ModelState.IsValid` property.

Note: Each entry in the controller's `ModelState` property is a `ModelStateEntry` containing an `Errors` property. It's rarely necessary to query this collection yourself. Use `ModelState.IsValid` instead.

Additionally, there are some special data types that MVC must consider when performing model binding:

- `IFormFile, IEnumerable<IFormFile>`: One or more uploaded files that are part of the HTTP request.
- `CancellationToken`: Used to cancel activity in asynchronous controllers.

These types can be bound to action parameters or to properties on a class type.

Once model binding is complete, [validation](#) occurs. Default model binding works great for the vast majority of development scenarios. It is also extensible so if you have unique needs you can customize the built-in behavior.

4.1.3 Customize model binding behavior with attributes

MVC contains several attributes that you can use to direct its default model binding behavior to a different source. For example, you can specify whether binding is required for a property, or if it should never happen at all by using the `[BindRequired]` or `[BindNever]` attributes. Alternatively, you can override the default data source, and specify the model binder's data source. Below is a list of model binding attributes:

- `[BindRequired]`: This attribute adds a model state error if binding cannot occur.

- `[BindNever]`: Tells the model binder to never bind to this parameter.
- `[FromHeader]`, `[FromQuery]`, `[FromRoute]`, `[FromForm]`: Use these to specify the exact binding source you want to apply.
- `[FromServices]`: This attribute uses [dependency injection](#) to bind parameters from services.
- `[FromBody]`: Use the configured formatters to bind data from the request body. The formatter is selected based on content type of the request.
- `[ModelBinder]`: Used to override the default model binder, binding source and name.

Attributes are very helpful tools when you need to override the default behavior of model binding.

4.1.4 Binding formatted data from the request body

Request data can come in a variety of formats including JSON, XML and many others. When you use the `[FromBody]` attribute to indicate that you want to bind a parameter to data in the request body, MVC uses a configured set of formatters to handle the request data based on its content type. By default MVC includes a `JsonInputFormatter` class for handling JSON data, but you can add additional formatters for handling XML and other custom formats.

Note: The `JsonInputFormatter` is the default formatter and it is based off of [Json.NET](#).

ASP.NET selects input formatters based on the [Content-Type](#) header and the type of the parameter, unless there is an attribute applied to it specifying otherwise. If you'd like to use XML or another format you must configure it in the *Startup.cs* file, but you may first have to obtain a reference to `Microsoft.AspNet.Mvc.Formatters.Xml` using NuGet. Your startup code should look something like this:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddMvc()
4         .AddXmlSerializerFormatters();
5 }
```

Code in the *Startup.cs* file contains a `ConfigureServices` method with a `services` argument you can use to build up services for your ASP.NET app. In the sample, we are adding an XML formatter as a service that MVC will provide for this app. The `options` argument passed into the `AddMvc` method allows you to add and manage filters, formatters, and other system options from MVC upon app startup. Then apply the `Consumes` attribute to controller classes or action methods to work with the format you want.

4.2 Model Validation

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

4.3 Formatting

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

4.4 Custom Formatters

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.1 Razor Syntax

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.2 Dynamic vs Strongly Typed Views

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

Learn more about [Dynamic vs Strongly Typed Views](#).

5.3 HTML Helpers

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.4 Tag Helpers

5.4.1 Introduction to Tag Helpers

By Rick Anderson

- *What are Tag Helpers?*
- *What Tag Helpers provide*
- *Managing Tag Helper scope*
- *IntelliSense support for Tag Helpers*
- *Tag Helpers compared to HTML Helpers*
- *Tag Helpers compared to Web Server Controls*
- *Customizing the Tag Helper element font*
- *Additional Resources*

What are Tag Helpers?

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with [HTML Helpers](#), Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. [Tag Helpers compared to HTML Helpers](#) explains the differences in more detail.

What Tag Helpers provide

An HTML-friendly development experience For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. [Tag Helpers compared to HTML Helpers](#) explains the differences in more detail. [IntelliSense support for Tag Helpers](#) explains the IntelliSense environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only a

For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element. For example, the `<input>` element used in many of the views in the *Views/Account* folder contains the `asp-for` attribute, which extracts the name of the specified model property into the rendered HTML. The following Razor markup:

```
<label asp-for="Email"></label>
```

Generates the following HTML:

```
<label for="Email">Email</label>
```

The `asp-for` attribute is made available by the `For` property in the `LabelTagHelper`. See [Authoring Tag Helpers](#) for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the `!”` opt-out character.

@addTagHelper makes Tag Helpers available

If you create a new ASP.NET 5 web app named *AuthoringTagHelpers* (with no authentication), the following *Views/_ViewImports.cshtml* file will be added to your project:

```
@using AuthoringTagHelpers
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is *Views/_ViewImports.cshtml*, which by default is inherited by all view files in the *Views* folder and sub-directories; making Tag Helpers available. The code above uses the wildcard syntax (“*”) to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNet.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or sub-directory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using “*” for all Tag Helpers), and the second parameter “*Microsoft.AspNet.Mvc.TagHelpers*” specifies the assembly containing the Tag Helpers. *Microsoft.AspNet.Mvc.TagHelpers* is the assembly for the built-in ASP.NET 5 Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

```
@using AuthoringTagHelpers
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
@addTagHelper "*", AuthoringTagHelpers"
```

If your project contains an `EmailTagHelper` with the default namespace (*AuthoringTagHelpers.TagHelpers.EmailTagHelper*), you can provide the fully qualified name (FQN) of the Tag Helper:

```
@using AuthoringTagHelpers
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
@addTagHelper "AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers"
```

To add a Tag Helper to a view using an FQN, you first add the FQN (*AuthoringTagHelpers.TagHelpers.EmailTagHelper*), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the “*” wildcard syntax. The wildcard syntax allows you to insert the wildcard character “*” as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper`:

```
@addTagHelper "AuthoringTagHelpers.TagHelpers.E*", AuthoringTagHelpers"
@addTagHelper "AuthoringTagHelpers.TagHelpers.Email*", AuthoringTagHelpers"
```

As mentioned previously, adding the `@addTagHelper` directive to the `Views/_ViewImports.cshtml` file makes the Tag Helper available to all view files in the `Views` directory and sub-directories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a `Views/Folder/_ViewImports.cshtml` file removes the specified Tag Helper from all of the views in `Folder`.

Controlling Tag Helper scope with the `_ViewImports.cshtml` file

You can add a `_ViewImports.cshtml` to any view folder, and the view engine adds the directives from that `_ViewImports.cshtml` file to those contained in the `Views/_ViewImports.cshtml` file. If you added an empty `Views/Home/_ViewImports.cshtml` file for the `Home` views, there would be no change because the `_ViewImports.cshtml` file is additive. Any `@addTagHelper` directives you add to the `Views/Home/_ViewImports.cshtml` file (that are not in the default `Views/_ViewImports.cshtml` file) would expose those Tag Helpers to views only in the `Home` folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character (`'!''`). For example, Email validation is disabled in the `` with the Tag Helper opt-out character:

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. In the code image below, the Tag Helper prefix is set to `"th:"`, so only those elements using the prefix `"th:"` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `<input>` element does not.

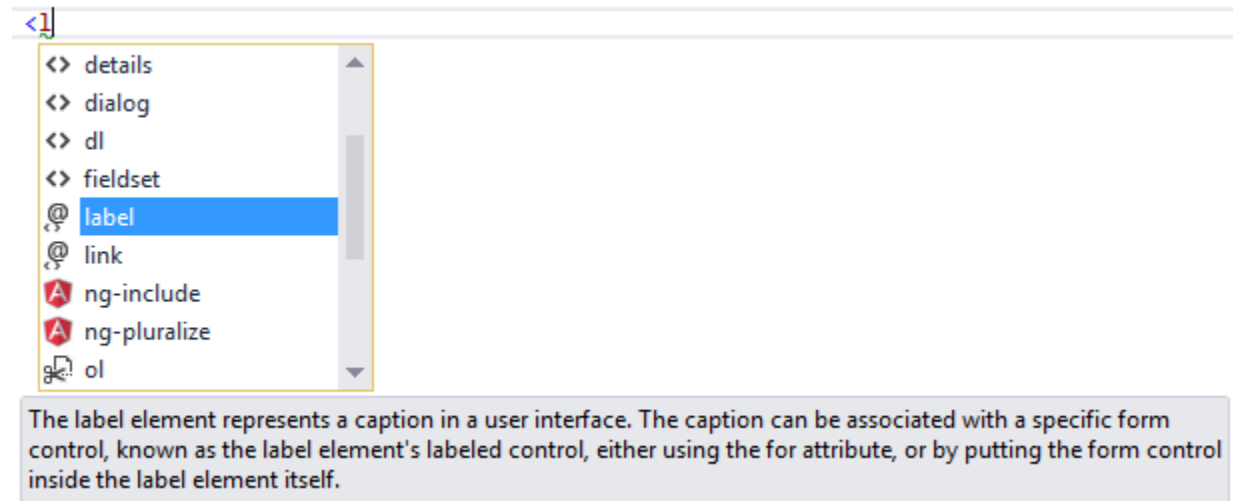
```
@tagHelperPrefix "th:"
<div class="form-group">
  <th:label asp-for="Password" class="col-md-2 control-label"></th:label>
  <div class="col-md-10">
    <input asp-for="Password" class="form-control" />
    <th:span asp-validation-for="Password" class="text-danger"></th:span>
  </div>
</div>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

IntelliSense support for Tag Helpers

When you create a new ASP.NET web app in Visual Studio, it adds “Microsoft.AspNet.Tooling.Razor” to the *project.json* file. This is the package that adds Tag Helper tooling.

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



Not only do you get HTML help, but the icon (the “@” symbol with “<>” under it).

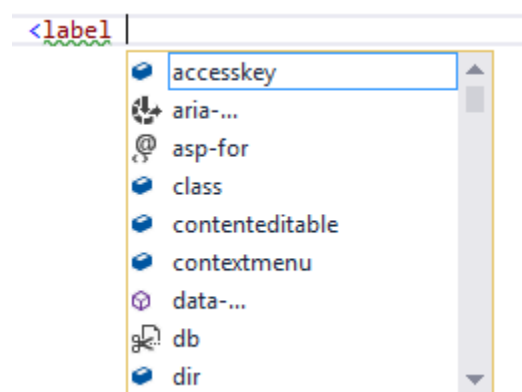


identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the “<>” icon.

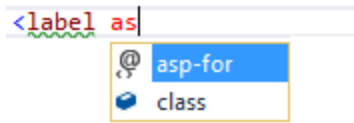
A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.

```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:



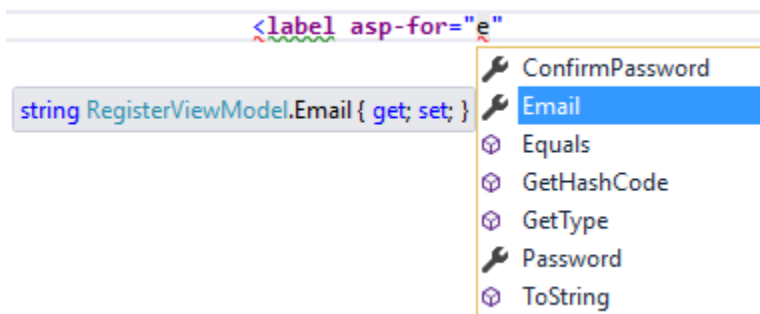
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:



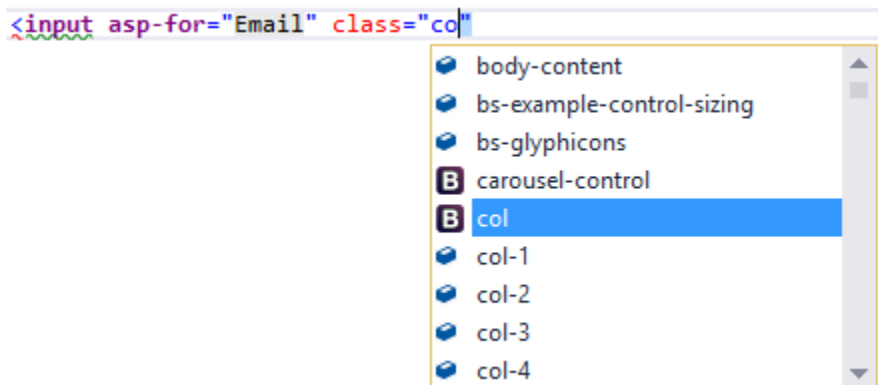
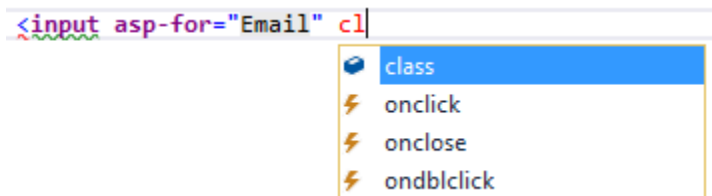
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio “Blue” or “Light” color theme, the font is bold purple. If you’re using the “Dark” theme the font is bold teal. The images in this document were taken using the default theme.

`<label asp-for`

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the `default`) inside the double quotes (“”), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I’m editing the `Register` view, so the `RegisterViewModel` is available.



IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:



Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while [HTML Helpers](#) are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class “caption”:

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

The at (@) symbol tells Razor this is the start of code. The next two parameters (“FirstName” and “First Name:”) are strings, so [IntelliSense](#) can’t help. The last argument:

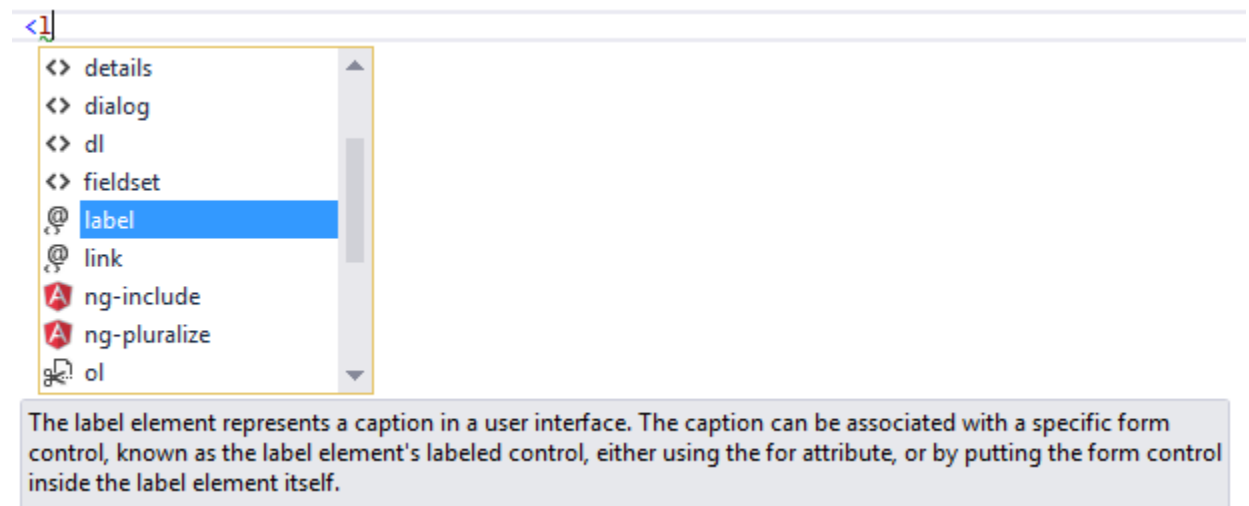
```
new {@class="caption"}
```

Is an anonymous object used to represent attributes. Because **class** is a reserved keyword in C#, you use the @ symbol to force C# to interpret “@class=” as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

```
<label class="caption" asp-for="FirstName"></label>
```

With the Tag Helper version, as soon as you enter <l in the Visual Studio editor, IntelliSense displays matching elements:



IntelliSense helps you write the entire line. The `LabelTagHelper` also defaults to setting the content of the `asp-for` attribute value (“FirstName”) to “First Name”; It converts camel-cased properties to a sentence composed of the property name with a space where each new upper-case letter occurs. In the following markup:

```
<label class="caption" asp-for="FirstName"></label>
```

generates:

```
<label class="caption" for="FirstName">First Name</label>
```

The camel-cased to sentence-cased content is not used if you add content to the `<label>`. For example:

```
<label class="caption" asp-for="FirstName">Name First</label>
```

generates:

```
<label class="caption" for="FirstName">Name First</label>
```

The following code image shows the Form portion of the *Views/Account/Register.cshtml* Razor view generated from the legacy ASP.NET 4.5.x MVC template included with Visual Studio 2015.

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizontal" })
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:


```

<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>

```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

```

<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>

```

Each of the “asp-” attributes has a value of “Email”, but “Email” is not a string. In this context, “Email” is the C# model expression property for the `RegisterViewModel`.

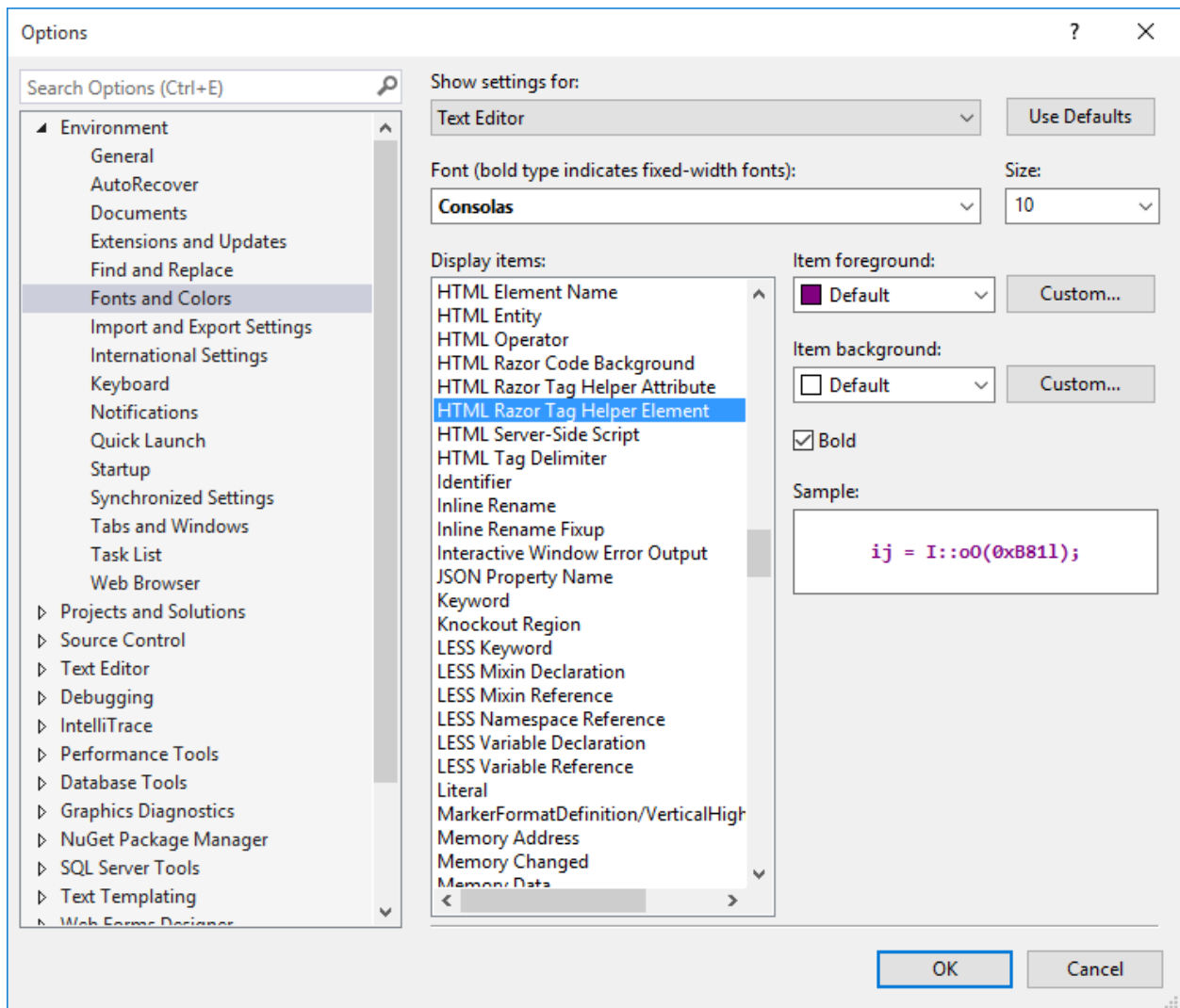
The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML Helpers approach. *IntelliSense support for Tag Helpers* goes into detail on working with Tag Helpers in the Visual Studio editor.

Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET [Web Server controls](#) are declared and invoked on a page.
- [Web Server controls](#) have a non-trivial lifecycle that can make developing and debugging difficult.
- Web Server controls allow you to add functionality to the client Document Object Model (DOM) elements by using a client control. Tag Helpers have no DOM.
- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.
- Multiple Tag Helpers can act on the same element (see [Avoiding Tag Helper conflicts](#)) while you typically can't compose Web Server controls.
- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.
- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.
- Web Server controls use [System.ComponentModel](#) to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools > Options > Environment > Fonts and Colors**:



Additional Resources

- [TagHelperSamples on GitHub](#) contains Tag Helper samples for working with [Bootstrap](#).
- [Channel 9 video on advanced Tag Helpers](#). This is a great video on more advanced features. It's a couple of versions out-of-date but the comments contain a list of changes to the current version. The updated code can be found [here](#).

5.4.2 Authoring Tag Helpers

By Rick Anderson

- *Getting started with Tag Helpers*
- *Starting the email Tag Helper*
- *A working email Tag Helper*
- *The bold Tag Helper*
- *Web site information Tag Helper*

- *Condition Tag Helper*
- *Avoiding Tag Helper conflicts*
- *Inspecting and retrieving child content*
- *Wrap up and next steps*
- *Additional Resources*

You can browse the source code for the sample app used in this document on [GitHub](#).

Getting started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. [Introduction to Tag Helpers](#) describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method. We will introduce the `TagHelper` methods and properties as we use them in this tutorial.

1. Create a new ASP.NET MVC 6 project called **AuthoringTagHelpers**. You won't need authentication for this project.
2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it is a reasonable convention. Now let's get started writing some simple tag helpers.

Starting the email Tag Helper

In this section we will write a tag helper that updates an email tag. For example:

```
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

1. Add the following `EmailTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.Runtime.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

Notes:

- Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **EmailTagHelper** is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.

- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides the rich methods and properties we will examine in this tutorial.
- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.
- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.
- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.
- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```
public class Email : TagHelper
```

2. To make the `EmailTagHelper` class available to all our Razor views, we will add the `addTagHelper` directive to the `Views/_ViewImports.cshtml` file:

```
@using AuthoringTagHelpers
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
@addTagHelper "*", AuthoringTagHelpers"
```

The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (we are using "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET 5 MVC 6 tag helpers using the wildcard syntax (those helpers are discussed in [Introduction to Tag Helpers](#).) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

```
@using AuthoringTagHelpers
@addTagHelper "*", Microsoft.AspNet.Mvc.TagHelpers"
@addTagHelper "AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers"
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*Authoring-TagHelpers*). Most developers will prefer to use the wildcard syntax. [Introduction to Tag Helpers](#) goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

3. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

4. Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

Note: Like [HTML tags and attributes](#), tags, class names and attributes in Razor, and C# are not case-sensitive.

A working email Tag Helper

In this section, we will update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update our tag helper to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // Pascal case gets translated into lower-kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes["href"] = "mailto:" + address;
        output.Content.SetContent(address);
    }
}
```

Notes:

- Pascal-cased class and property names for tag helpers are translated into their [lower kebab case](#). Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.
- The last line sets the completed content for our minimally functional tag helper.
- The following line shows the syntax for adding attributes:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes["href"] = "mailto:" + address;
    output.Content.SetContent(address);
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

3. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
```

```

<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email mail-to="Support"></email><br />
    <strong>Marketing:</strong><email mail-to="Marketing"></email>
</address>

```

4. Run the app and verify that it generates the correct links.

Note: If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a start tag (`<email mail-to="Rick">`) you must decorate the class with the following:

```
[TargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
```

With a self-closing email tag helper, the output would be ``. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that is self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

An asynchronous email helper

In this section we'll write an asynchronous email helper.

1. Replace the `EmailTagHelper` class with the following code:

```

public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a"; // Replaces <email> with <a> tag
        var content = await context.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes["href"] = "mailto:" + target;
        output.Content.SetContent(target);
    }
}

```

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.
- We use the `output` parameter to get contents of the HTML element.

2. Make the following change to the `Views/Home/Contact.cshtml` file so the tag helper can get the target email.

```

@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

```

```
<address>
  One Microsoft Way<br />
  Redmond, WA 98052<br />
  <abbr title="Phone">P:</abbr>
  425.555.0100
</address>

<address>
  <strong>Support:</strong><email>Support</email><br />
  <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. Run the app and verify that it generates valid email links.

The bold Tag Helper

1. Add the following BoldTagHelper class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.Runtime.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [TargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

Notes:

- The [HtmlTargetElement] attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named “bold” will match, and the Process override method in the class will run. In our sample, the Process method removes the “bold” attribute and surrounds the containing markup with .
- Because we don’t want to replace the existing tag content, we must write the opening tag with the PreContent.SetHtmlContent method and the closing tag with the PostContent.SetHtmlContent method.

2. Modify the *About.cshtml* view to contain a bold attribute value. The completed code is shown below.

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```


3. Run the app. You can use your favorite browser to inspect the source and verify that the markup has changed as promised.

The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of “bold”. The `<bold>` element was not modified by the tag helper.

4. Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **BoldTagHelper** to `<bold>` tags.
5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

```
[TargetElement("bold")]
[TargetElement(Attributes = "bold")]
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named “bold” with an attribute named “bold” (`<bold bold />`) to match.

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

For a good example of a bootstrap progress bar that targets a tag and an attribute, see [Creating custom MVC 6 Tag Helpers](#).

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the **BoldTagHelper** to target `<MyBold>` tags, you would use the following attribute:

```
[HtmlTargetElement("MyBold")]
```

Web site information Tag Helper

1. Add a *Models* folder.
2. Add the following *WebsiteContext* class to the *Models* folder:

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. Add the following *WebsiteInformationTagHelper* class to the *TagHelpers* folder.

```
using System;
using Microsoft.AspNetCore.Razor.Runtime.TagHelpers;
using AuthoringTagHelpers.Models;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
```

```

        public WebsiteContext Info { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "section";
            output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li></ul>");
            output.TagMode = TagMode.StartTagAndEndTag;
        }
    }
}

```

Notes:

- As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into **lower kebab case**. Therefore, to use the `WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.
- We are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

```
[HtmlTargetElement("WebsiteInformation")]
```

The lower kebab case tag `<website-information />` would not match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

```
[HtmlTargetElement("Website-Information")]
```

- Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a `section` element (which is not self-closing and we are writing content inside the `section` element). Therefore, we need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)
- The `$` (dollar sign) in the following line uses an **interpolated string**:

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

5. Add the following markup to the `About.cshhtml` view. The highlighted markup displays the web site information.

```

@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1790,

```

```
Approved = true,
TagsToShow = 131 }" />
```

Note: In the Razor markup shown below:

```
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1790,
    Approved = true,
    TagsToShow = 131 }" />
```

Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

6. Run the app, and navigate to the About view to see the web site information.

Note:

- You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:

```
<website-information info="new WebsiteContext {
    Version = new Version(1, 3),
    CopyrightYear = 1790,
    Approved = true,
    TagsToShow = 131 }" >
</website-information>
```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

1. Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.Runtime.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [TargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
```

```
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <Website-InfoMation info=Model />
    <div condition="Model.Approved">
        <p>
            This website has <strong surround="em"> @Model.Approved </strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. Replace the Index method in the Home controller with the following code:

```
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

4. Run the app and browse to the home page. The markup in the conditional div will not be rendered. Append the query string `?approved=true` to the URL (for example, <http://localhost:1235/Home/Index?approved=true>). The approved is set to true and the conditional markup will be displayed.

Note: We use the `nameof` operator to specify the attribute to target rather than specifying a string as we did with the bold tag helper:

```
[TargetElement(Attributes = nameof(Condition))]
// [TargetElement(Attributes = "condition")]
public class ConditionTagHelper : TagHelper
{
    public bool Condition { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        if (!Condition)
        {
            output.SuppressOutput();
        }
    }
}
```

The `nameof` operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoiding Tag Helper conflicts

In this section, we will write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and we may refactor them in the future, we'll keep them in the same file.

1. Add the following `AutoLinker` class to the `TagHelpers` folder.

```
[TargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\"b(?:https?://)(\\S+)\\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version
    }
}
```

Notes: The `AutoLinkerHttpTagHelper` class targets `p` elements and uses `Regex` to create the anchor.

2. Add the following markup to the end of the `Views/Home/Contact.cshtml` file:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. Run the app and verify that the tag helper renders the anchor correctly.
4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert `www` text to an anchor tag that also contains the original `www` text. The updated code is highlighted below:

```
[TargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\"b(?:https?://)(\\S+)\\b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version
    }
}
```

```
[TargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent.GetContent(),
            @"\"b(www\\.) (\\S+) \\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>")); // www version
    }
}
```

5. Run the app. Notice the www text is rendered as a link but the HTTP text is not. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. Later in the tutorial we'll see how to control the order that tag helpers run in. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. We'll fix that with the following code:

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\"b(?:https?://) (\\S+) \\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>")); // http link version
    }
}

[TargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\"b(www\\.) (\\S+) \\b",
            "<a target=\"_blank\" href=\"http://$0\">$0</a>")); // www version
    }
}
```

Note: In the first edition of the auto-linking tag helpers, we got the content of the target with the following code:

```
var childContent = await output.GetChildContentAsync();
```

That is, we call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. We fixed that problem with the following code:

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

7. Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links will not be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get { return int.MinValue; }
    }
}
```

The above code will guarantee that the WWW tag helper runs before the HTTP tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspecting and retrieving child content

The tag-helpers provide several properties to retrieve content.

- The result of `GetChildContentAsync` can be appended to `output.Content`.
- You can inspect the result of `GetChildContentAsync` with `GetContent`.
- If you modify `output.Content`, the `TagHelper` body will not be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
            childContent,
            @"\"b(?:https?://) (\S+)\"b",
            "<a target=\"_blank\" href=\"$0\">$0</a>")); // http link version
    }
}
```

- Multiple calls to `GetChildContentAsync` will return the same value and will not re-execute the `TagHelper` body unless you pass in a false parameter indicating not use the cached result.

Wrap up and next steps

This tutorial was an introduction to authoring tag helpers and *the code samples* should not be considered a guide to best practices. For example, a real app would probably use a more elegant regular expression to replace both HTTP and WWW links in one expression. The ASP.NET 5 MVC 6 tag helpers provide the best examples of well-written tag helpers.

Additional Resources

- [TagHelperSamples on GitHub](#) contains tag helper samples for working with [Bootstrap](#).
- [Channel 9 video on advanced tag helpers](#). This is a great video on more advanced features. It's a couple versions out of date but the comments contain a list of changes to the current version and the updated code can be found [here](#).

5.4.3 Advanced Tag Helpers

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.5 Partial Views

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.6 Injecting Services Into Views

By [Steve Smith](#)

ASP.NET MVC 6 supports [dependency injection](#) into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. You should try to maintain [separation of concerns](#) between your controllers and views. Most of the data your views display should be passed in from the controller.

Sections:

- *A Simple Example*
- *Populating Lookup Data*
- *Overriding Services*

[View sample files](#)

5.6.1 A Simple Example

You can inject a service into a view using the `@inject` directive. You can think of `@inject` as adding a property to your view, and populating the property using DI.

The syntax for `@inject`: `@inject <type> <name>`

An example of `@inject` in action:

```

1 @using System.Threading.Tasks
2 @using ViewInjectSample.Model
3 @using ViewInjectSample.Model.Services
4 @model IEnumerable<ToDoItem>
5 @inject StatisticsService StatsService
6 <!DOCTYPE html>
7 <html>
8 <head>
9     <title>To Do Items</title>
10 </head>
11 <body>
12     <div>
13         <h1>To Do Items</h1>
14         <ul>
15             <li>Total Items: @await StatsService.GetCount()</li>
16             <li>Completed: @await StatsService.GetCompletedCount()</li>
17             <li>Avg. Priority: @await StatsService.GetAveragePriority()</li>
18         </ul>
19         <table>
20             <tr>
21                 <th>Name</th>
22                 <th>Priority</th>
23                 <th>Is Done?</th>
24             </tr>
25             @foreach (var item in Model)
26             {
27                 <tr>
28                     <td>@item.Name</td>
29                     <td>@item.Priority</td>
30                     <td>@item.IsDone</td>
31                 </tr>
32             }
33         </table>
34     </div>
35 </body>
36 </html>

```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in `Startup.cs`:

```

1 // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
2 public void ConfigureServices(IServiceCollection services)
3 {
4     services.AddMvc();
5
6     services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
7     services.AddTransient<StatisticsService>();
8     services.AddTransient<ProfileOptionsService>();

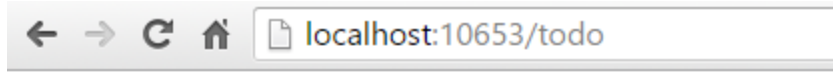
```

The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

```
1 using System.Linq;
2 using System.Threading.Tasks;
3 using ViewInjectSample.Interfaces;
4
5 namespace ViewInjectSample.Model.Services
6 {
7     public class StatisticsService
8     {
9         private readonly IToDoItemRepository _todoItemRepository;
10
11         public StatisticsService(IToDoItemRepository todoItemRepository)
12         {
13             _todoItemRepository = todoItemRepository;
14         }
15
16         public async Task<int> GetCount()
17         {
18             return await Task.FromResult(_todoItemRepository.List().Count());
19         }
20
21         public async Task<int> GetCompletedCount()
22         {
23             return await Task.FromResult(
24                 _todoItemRepository.List().Count(x => x.IsDone));
25         }
26
27         public async Task<double> GetAveragePriority()
28         {
29             if (_todoItemRepository.List().Count() == 0)
30             {
31                 return 0.0;
32             }
33
34             return await Task.FromResult(
35                 _todoItemRepository.List().Average(x => x.Priority));
36         }
37     }
38 }
```

The sample repository uses an in-memory collection. The implementation shown above (which operates on all of the data in memory) is not recommended for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:



To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

5.6.2 Populating Lookup Data

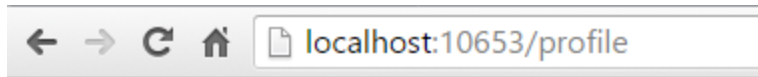
View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard MVC approach would require the controller to request data access services for each of these sets of options, and then populate a model or `ViewBag` with each set of options to be bound.

An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller, moving this view element construction logic into the view itself. The controller action to display a profile editing form only needs to pass the form the profile instance:

```

1  using Microsoft.AspNet.Mvc;
2  using ViewInjectSample.Model;
3
4  namespace ViewInjectSample.Controllers
5  {
6      public class ProfileController : Controller
7      {
8          [Route("Profile")]
9          public IActionResult Index()
10         {
11             // TODO: look up profile based on logged-in user
12             var profile = new Profile()
13             {
14                 Name = "Steve",
15                 FavColor = "Blue",
16                 Gender = "Male",
17                 State = new State("Ohio", "OH")
18             };
19             return View(profile);
20         }
21     }
22 
```

The HTML form used to update these preferences includes dropdown lists for three of the properties:



Update Profile

Name:

Gender:

State:

Fav. Color:

These lists are populated by a service that has been injected into the view:

```

1  @using System.Threading.Tasks
2  @using ViewInjectSample.Model.Services
3  @model ViewInjectSample.Model.Profile
4  @inject ProfileOptionsService Options
5  <!DOCTYPE html>
6  <html>
7  <head>
8      <title>Update Profile</title>
9  </head>
10 <body>
11 <div>
12     <h1>Update Profile</h1>
13     Name: @Html.TextBoxFor(m => m.Name)
14     <br/>
15     Gender: @Html.DropDownList("Gender",
16         Options.ListGenders().Select(g =>
17             new SelectListItem() { Text = g, Value = g }))
18     <br/>
19
20     State: @Html.DropDownListFor(m => m.State.Code,
21         Options.ListStates().Select(s =>
22             new SelectListItem() { Text = s.Name, Value = s.Code}))
23     <br />
24
25     Fav. Color: @Html.DropDownList("FavColor",
26         Options.ListColors().Select(c =>
27             new SelectListItem() { Text = c, Value = c }))
28 </div>
29 </body>
30 </html>

```

The ProfileOptionsService is a UI-level service designed to provide just the data needed for this form:

```

1  using System.Collections.Generic;
2
3  namespace ViewInjectSample.Model.Services
4  {
5      public class ProfileOptionsService
6      {
7          public List<string> ListGenders()

```

```

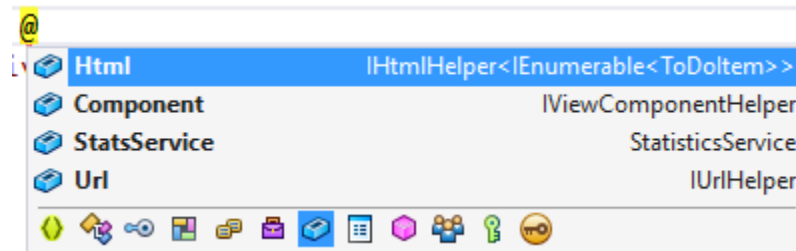
8      {
9          // keeping this simple
10         return new List<string>() { "Female", "Male" };
11     }
12
13     public List<State> ListStates()
14     {
15         // a few states from USA
16         return new List<State>()
17         {
18             new State("Alabama", "AL"),
19             new State("Alaska", "AK"),
20             new State("Ohio", "OH")
21         };
22     }
23
24     public List<string> ListColors()
25     {
26         return new List<string>() { "Blue", "Green", "Red", "Yellow" };
27     }
28 }
29

```

Tip: Don't forget to register types you will request through dependency injection in the `ConfigureServices` method in `Startup.cs`.

5.6.3 Overriding Services

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



As you can see, the default fields include `Html`, `Component`, and `Url` (as well as the `StatsService` that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using `@inject`:

```

1  @using System.Threading.Tasks
2  @using ViewInjectSample.Helpers
3  @inject MyHtmlHelper Html
4  <!DOCTYPE html>
5  <html>
6  <head>
7      <title>My Helper</title>
8  </head>
9  <body>
10     <div>
11         Test: @Html.Value

```

```
12     </div>
13 </body>
14 </html>
```

If you want to extend existing services, you can simply use this technique while inheriting from or wrapping the existing implementation with your own.

5.6.4 See Also

- Simon Timms Blog: [Getting Lookup Data Into Your View](#)

5.7 View Components

By [Rick Anderson](#)

In this article:

- *Introducing view components*
- *Examine the `ViewComponent` class*
- *Examine the view component view*
- *Add `InvokeAsync` to the priority view component*
- *Specifying a view name*

5.7.1 Introducing view components

New to ASP.NET MVC 6, view components are similar to partial views, but they are much more powerful. View components include the same separation-of-concerns and testability benefits found between a controller and view. A view component is responsible for rendering a chunk rather than a whole response. You can use view components to solve any problem that you feel is too complex with a partial, such as:

- Dynamic navigation menus
- Tag cloud (where it queries the database)
- Login panel
- Shopping cart
- Recently published articles
- Sidebar content on a typical blog

One use of a view component could be to create a login panel that would be displayed on every page with the following functionality:

- If the user is not logged in, a login panel is rendered.
- If the user is logged in, links to log out and manage account are rendered.
- If the user is in the admin role, an admin panel is rendered.

You can also create a view component that gets and renders data depending on the user's claims. You can add this view component view to the layout page and have it get and render user-specific data throughout the whole application. View components don't use model binding, and only depend on the data you provide when calling into it.

A view component consists of two parts, the class (typically derived from `ViewComponent`) and the Razor view which calls methods in the view component class. Like controllers, a view component can be a POCO, but most users will want to take advantage of the methods and properties available by deriving from `ViewComponent`.

A view component class can be created by any of the following:

- Deriving from *ViewComponent*.
- Decorating the class with the `[ViewComponent]` attribute, or deriving from a class with the `[ViewComponent]` attribute.
- Creating a class where the name ends with the suffix *ViewComponent*.

Like controllers, view components must be public, non-nested, non-abstract classes.

5.7.2 Examine the `ViewComponent` class

- Examine the `src\ToDoList\ViewComponents\PriorityListViewComponent.cs` file:

```

1  using System.Linq;
2  using Microsoft.AspNetCore.Mvc;
3  using ToDoList.Models;
4
5  namespace ToDoList.ViewComponents
6  {
7      public class PriorityListViewComponent : ViewComponent
8      {
9          private readonly ApplicationDbContext db;
10
11          public PriorityListViewComponent(ApplicationDbContext context)
12          {
13              db = context;
14          }
15
16          public IViewComponentResult Invoke(int maxPriority)
17          {
18              var items = db.TODOItems.Where(x => x.IsDone == false &&
19                  x.Priority <= maxPriority);
20
21              return View(items);
22          }
23      }
24  }
```

Notes on the code:

- View component classes can be contained in **any** folder in the project.
- Because the class name `PriorityListViewComponent` ends with the suffix **ViewComponent**, the runtime will use the string “PriorityList” when referencing the class component from a view. I’ll explain that in more detail later.
- The `[ViewComponent]` attribute can change the name used to reference a view component. For example, we could have named the class `XYZ`, and applied the `ViewComponent` attribute:

```

1  [ViewComponent(Name = "PriorityList")]
2  public class XYZ : ViewComponent
```

- The `[ViewComponent]` attribute above tells the view component selector to use the name `PriorityList` when looking for the views associated with the component, and to use the string “PriorityList” when referencing

the class component from a view. I'll explain that in more detail later.

- The component uses constructor injection to make the data context available.
- `Invoke` exposes a method which can be called from a view, and it can take an arbitrary number of arguments. An asynchronous version, `InvokeAsync`, is available. We'll see `InvokeAsync` and multiple arguments later in the tutorial. In the code above, the `Invoke` method returns the set of *ToDoItems* that are not completed and have priority greater than or equal to `maxPriority`.

5.7.3 Examine the view component view

1. Examine the contents of the *Views\Todo\Components*. This folder **must** be named *Components*.

Note: View Component views are more typically added to the *Views\Shared\Components* folder, because view components are typically not controller specific.

2. Examine the *Views\Todo\Components\PriorityList* folder. This folder name must match the name of the view component (name of the class minus the suffix if we followed convention and used the *ViewComponent* suffix in the class name). If you used the `ViewComponent` attribute, the folder name would need to match the attribute designation.
3. Examine the *Views\Todo\Components\PriorityList\Default.cshtml* Razor view.

```
1 @model IEnumerable<ToDoList.Models.ToDoItem>
2
3 <h3>Priority Items</h3>
4 <ul>
5     @foreach (var todo in Model)
6     {
7         <li>@todo.Title</li>
8     }
9 </ul>
```

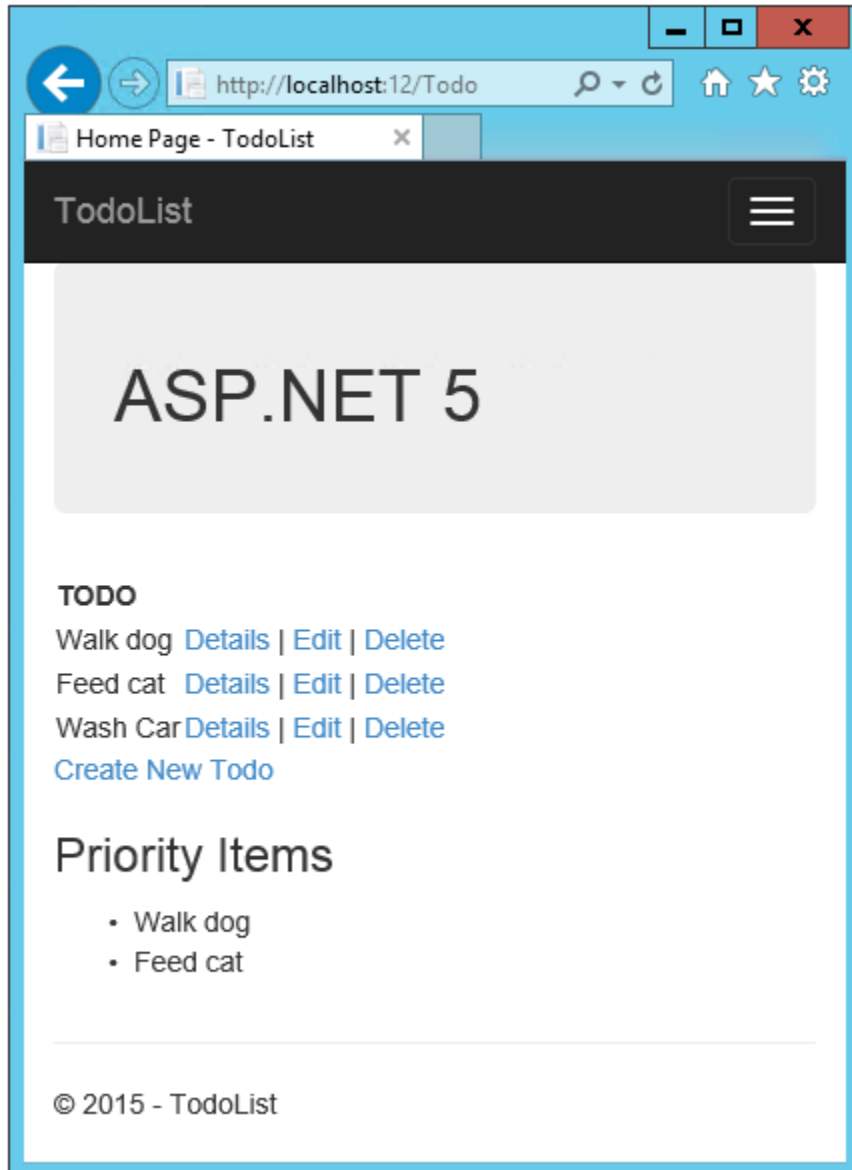
The Razor view takes a list of `ToDoItems` and displays them. If the view component `invoke` method doesn't pass the name of the view (as in our sample), *Default* is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view.

4. Add a `div` containing a call to the priority list component to the bottom of the *views\todo\index.cshtml* file:

```
1 @* Markup removed for brevity *@
2 <div>@Html.ActionLink("Create New Todo", "Create", "Todo") </div>
3 <div>
4     <div class="col-md-4">
5         @Component.Invoke("PriorityList", 1)
6     </div>
7 </div>
```

The markup `@Component.Invoke` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. In this case, we are passing "1" as the priority we want to filter on. `Invoke` and `InvokeAsync` can take an arbitrary number of arguments.

The following image shows the priority items: (make sure you have at least one priority 1 item that is not completed)



5.7.4 Add InvokeAsync to the priority view component

Update the priority view component class with the following code:

Note: `IQueryable` renders the sample synchronous, not asynchronous. This is a simple example of how you could call asynchronous methods.

```
1 using System.Threading.Tasks;
2
3 public class PriorityListViewComponent : ViewComponent
4 {
5     private readonly ApplicationDbContext db;
6
7     public PriorityListViewComponent(ApplicationDbContext context)
8     {
```

```

9      db = context;
10     }
11
12     // Synchronous Invoke removed.
13
14     public async Task<IViewComponentResult> InvokeAsync(int maxPriority, bool isDone)
15     {
16         var items = await GetItemsAsync(maxPriority, isDone);
17         return View(items);
18     }
19
20     private Task<IQueryable<ToDoItem>> GetItemsAsync(int maxPriority, bool isDone)
21     {
22         return Task.FromResult(GetItems(maxPriority, isDone));
23     }
24     private IQueryable<ToDoItem> GetItems(int maxPriority, bool isDone)
25     {
26         var items = db.ToDoItems.Where(x => x.IsDone == isDone &&
27             x.Priority <= maxPriority);
28
29         string msg = "Priority <= " + maxPriority.ToString() +
30             " && isDone == " + isDone.ToString();
31         ViewBag.PriorityMessage = msg;
32
33         return items;
34     }
35 }

```

Update the view component Razor view (*ToDoList\src\ToDoList\Views\ToDo\Components\PriorityList\Default.cshtml*) to show the priority message :

```

1 @model IEnumerable<ToDoList.Models.ToDoItem>
2
3 <h4>@ViewBag.PriorityMessage</h4>
4 <ul>
5     @foreach (var todo in Model)
6     {
7         <li>@todo.Title</li>
8     }
9 </ul>

```

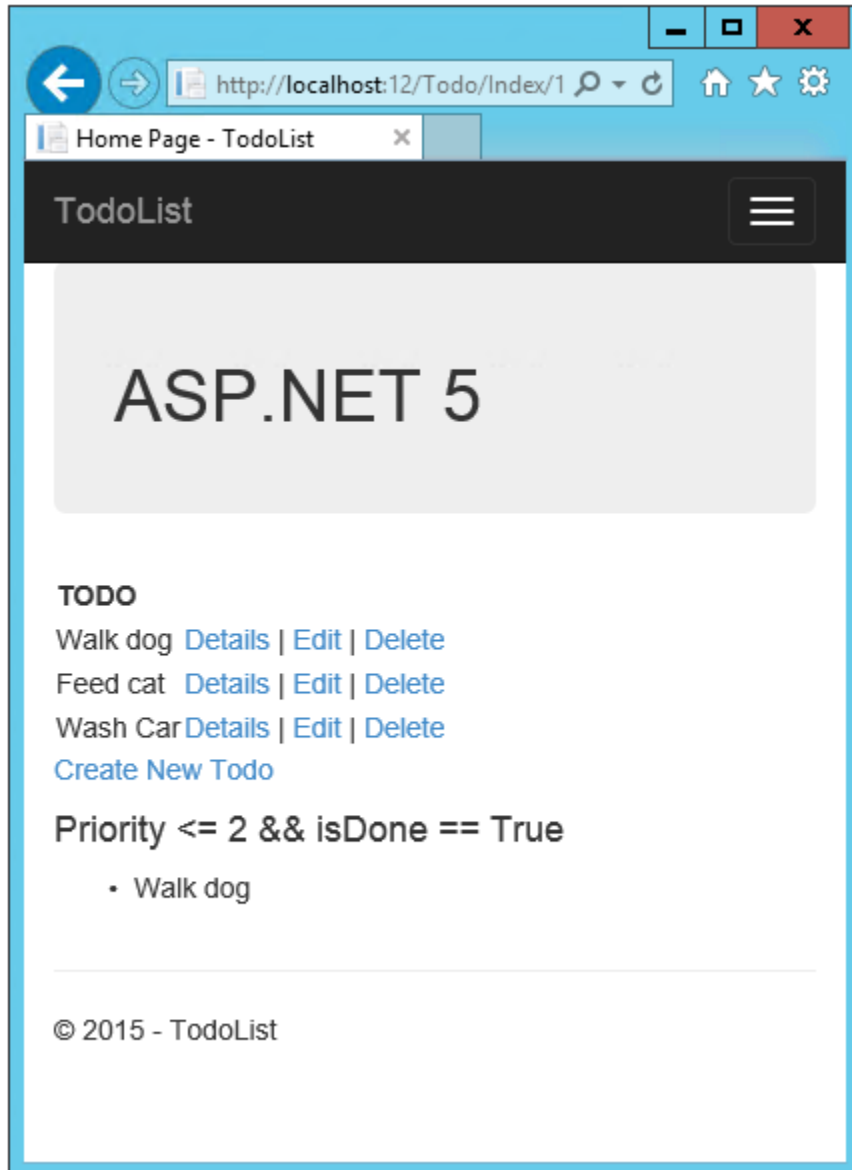
Finally, update the *views\todo\index.cshtml* view:

```

1 @* Markup removed for brevity. *@
2 <div class="col-md-4">
3     @await Component.InvokeAsync("PriorityList", 2, true)
4 </div>

```

The following image reflects the changes we made to the priority view component and Index view:



5.7.5 Specifying a view name

A complex view component might need to specify a non-default view under some conditions. The following shows how to specify the “PVC” view from the `InvokeAsync` method: Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

```

1 public async Task<IViewComponentResult> InvokeAsync(int maxPriority, bool isDone)
2 {
3     string MyView = "Default";
4     // If asking for all completed tasks, render with the "PVC" view.
5     if (maxPriority > 3 && isDone == true)
6     {
7         MyView = "PVC";
8     }
9     var items = await GetItemsAsync(maxPriority, isDone);

```

```
10     return View(MyView, items);  
11 }
```

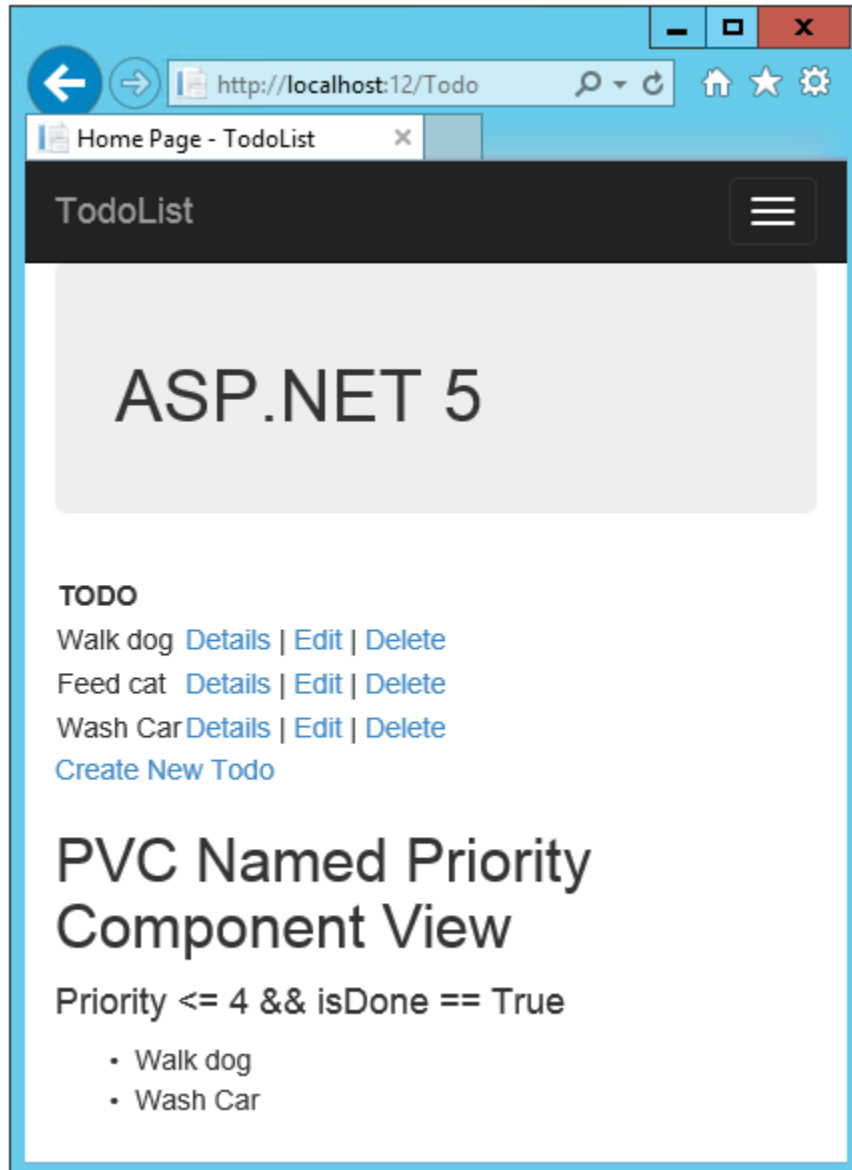
Examine the *Views\Todo\Components\PriorityList\PVC.cshtml* view. I changed the PVC view to verify it's being used:

```
1 @model IEnumerable<TodoList.Models.TODOItem>  
2  
3 <h2> PVC Named Priority Component View</h2>  
4 <h4>@ViewBag.PriorityMessage</h4>  
5 <ul>  
6     @foreach (var todo in Model)  
7     {  
8         <li>@todo.Title</li>  
9     }  
10 </ul>
```

Finally, update *Views\TodoIndex.cshtml*

```
1 @await Component.InvokeAsync("PriorityList", 4, true)
```

Run the app and click on the PVC link (or navigate to `localhost:<port>/Todo/IndexFinal`). Refresh the page to see the PVC view.



5.8 Creating a Custom View Engine

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

5.9 Building Mobile Specific Views

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

Controllers

6.1 Controllers, Actions, and Action Results

By [Steve Smith](#)

Actions and action results are a fundamental part of how developers build apps using ASP.NET MVC.

Sections:

- [What is a Controller](#)
- [Defining Actions](#)

[View or download sample from GitHub.](#)

6.1.1 What is a Controller

In ASP.NET MVC, a *Controller* is used to define and group a set of actions. An *action* (or *action method*) is a method on a controller that handles incoming requests. Controllers provide a logical means of grouping similar actions together, allowing common sets of rules (e.g. routing, caching, authorization) to be applied collectively. Incoming requests are mapped to actions through [routing](#).

In ASP.NET 5, a controller can be any instantiable class that ends in “Controller” or inherits from a class that ends with “Controller”. Controllers should follow the [Explicit Dependencies Principle](#) and request any dependencies their actions require through their constructor using [dependency injection](#).

By convention, controller classes:

- are located in the root-level “Controllers” folder
- inherit from `Microsoft.AspNet.Mvc.Controller`

These two conventions are not required.

Within the Model-View-Controller pattern, a Controller is responsible for the initial processing of the request and instantiation of the Model. Generally, business decisions should be performed within the Model.

Note: The Model should be a *Plain Old CLR Object (POCO)*, not a `DbContext` or database-related type.

The controller takes the result of the model’s processing (if any), returns the proper view along with the associated view data. Learn more: [Overview of ASP.NET MVC](#) and [Getting started with ASP.NET MVC 6](#).

Tip: The Controller is a *UI level* abstraction. Its responsibility is to ensure incoming request data is valid and to

choose which view (or result for an API) should be returned. In well-factored apps it will not directly include data access or business logic, but instead will delegate to services handling these responsibilities.

6.1.2 Defining Actions

Any public method on a controller type is an action. Parameters on actions are bound to request data and validated using [model binding](#).

Warning: Action methods that accept parameters should verify the `ModelState.IsValid` property is true.

Action methods should contain logic for mapping an incoming request to a business concern. Business concerns should typically be represented as services that your controller accesses through [dependency injection](#). Actions then map the result of the business action to an application state.

Actions can return anything, but frequently will return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*; the action result *does the responding*.

Controller Helper Methods

Although not required, most developers will want to have their controllers inherit from the base `Controller` class. Doing so provides controllers with access to many properties and helpful methods, including the following helper methods designed to assist in returning various responses:

View Returns a view that uses a model to render HTML. Example: `return View(customer);`

HTTP Status Code Return an HTTP status code. Example: `return BadRequest();`

Formatted Response Return `Json` or similar to format an object in a specific manner. Example: `return Json(customer);`

Content negotiated response Instead of returning an object directly, an action can return a content negotiated response (using `Ok`, `Created`, `CreatedAtRoute` or `CreatedAtAction`). Examples: `return Ok();` or `return CreatedAtRoute("routename", values, newobject);`

Redirect Returns a redirect to another action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction` or `RedirectToRoute`). Example: `return RedirectToAction("Complete", new {id = 123});`

In addition to the methods above, an action can also simply return an object. In this case, the object will be formatted based on the client's request. Learn more about [Formatting](#)

Cross-Cutting Concerns

In most apps, many actions will share parts of their workflow. For instance, most of an app might be available only to authenticated users, or might benefit from caching. When you want to perform some logic before or after an action method runs, you can use a *filter*. You can help keep your actions from growing too large by using [Filters](#) to handle these cross-cutting concerns. This can help eliminate duplication within your actions, allowing them to follow the [Don't Repeat Yourself \(DRY\) principle](#).

In the case of authorization and authentication, you can apply the `Authorize` attribute to any actions that require it. Adding it to a controller will apply it to all actions within that controller. Adding this attribute will ensure the appropriate filter is applied to any request for this action. Some attributes can be applied at both controller and action levels to provide granular control over filter behavior. Learn more: [Filters](#) and [Authorization Filters](#).

Other examples of cross-cutting concerns in MVC apps may include:

- [Error Handling](#)
- [Response Caching](#)

Note: Many cross-cutting concerns can be handled using filters in MVC apps. Another option to keep in mind that is available to any ASP.NET app is [custom middleware](#).

6.2 Routing to Controller Actions

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

6.3 Error Handling

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

6.4 Filters

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

6.5 Dependency Injection and Controllers

By [Steve Smith](#)

ASP.NET MVC 6 controllers should request their dependencies explicitly via their constructors. In some instances, individual controller actions may require a service, and it may not make sense to request at the controller level. In this case, you can also choose to inject a service as a parameter on the action method.

In this article:

- [Dependency Injection](#)
- [Constructor Injection](#)
- [Action Injection with FromServices](#)
- [Accessing Settings from a Controller](#)

[View or download sample from GitHub.](#)

6.5.1 Dependency Injection

Dependency injection is a technique that follows the [Dependency Inversion Principle](#), allowing for applications to be composed of loosely coupled modules. ASP.NET 5, which ASP.NET MVC 6 is built on, has built-in support for [dependency injection](#) ([learn more](#)), and expects applications built for ASP.NET 5 to implement this technique (rather than static access or direct instantiation).

Tip: It's important that you have a good understanding of how ASP.NET 5 implements Dependency Injection (DI). If you haven't already done so, please read [dependency injection in ASP.NET 5 Fundamentals](#).

6.5.2 Constructor Injection

ASP.NET 5's built-in support for constructor-based dependency injection extends to ASP.NET MVC 6 controllers. By simply adding a service type to your controller as a constructor parameter, ASP.NET will attempt to resolve that type using its built in service container. Services are typically, but not always, defined using interfaces. For example, if your application has business logic that depends on the current time, you can inject a service that retrieves the time (rather than hard-coding it), which would allow your tests to pass in implementations that use a set time.

```
1 using System;
2
3 namespace ControllerDI.Interfaces
4 {
5     public interface IDateTime
6     {
7         DateTime Now { get; }
8     }
9 }
```

Implementing an interface like this one so that it uses the system clock at runtime is trivial:

```
1 using System;
2 using ControllerDI.Interfaces;
3
4 namespace ControllerDI.Services
5 {
6     public class SystemDateTime : IDateTime
7     {
8         public DateTime Now
9         {
10             get { return DateTime.Now; }
11         }
12     }
13 }
```

```

12     }
13 }

```

With this in place, we can use the service in our controller. In this case, we have added some logic to the HomeController Index method to display a greeting to the user based on the time of day.

```

1  using ControllerDI.Interfaces;
2  using Microsoft.AspNet.Mvc;
3
4  namespace ControllerDI.Controllers
5  {
6      public class HomeController : Controller
7      {
8          private readonly IDateTime _dateTime;
9
10         public HomeController(IDateTime dateTime)
11         {
12             _dateTime = dateTime;
13         }
14
15         public IActionResult Index()
16         {
17             var serverTime = _dateTime.Now;
18             if (serverTime.Hour < 12)
19             {
20                 ViewData["Message"] = "It's morning here - Good Morning!";
21             }
22             else if (serverTime.Hour < 17)
23             {
24                 ViewData["Message"] = "It's afternoon here - Good Afternoon!";
25             }
26             else
27             {
28                 ViewData["Message"] = "It's evening here - Good Evening!";
29             }
30             return View();
31         }
32     }
33 }

```

If we run the application now, we will most likely encounter an error:

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type 'ControllerDI.Interfaces.IDateTime' while attempting to activate 'ControllerDI.Controllers.HomeController'. Microsoft.Extensions.DependencyInjection.ActivatorUtilities.GetService(IServiceProvider sp, Type type, Type requiredBy, Boolean isDefaultParameterRequired)

This error occurs when we have not configured a service in the ConfigureServices method in our Startup class. To specify that requests for IDateTime should be resolved using an instance of SystemDateTime, add the highlighted line in the listing below to your ConfigureServices method:

```

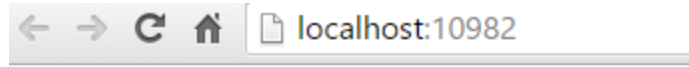
1  public void ConfigureServices(IServiceCollection services)
2  {
3      services.AddMvc();
4
5      // Add application services.
6      services.AddTransient<IDateTime, SystemDateTime>();

```

7 }

Note: This particular service could be implemented using any of several different lifetime options (`Transient`, `Scoped`, or `Singleton`). Be sure you understand how each of these scope options will affect the behavior of your service. [Learn more](#).

Once the service has been configured, running the application and navigating to the home page should display the time-based message as expected:



A Message From The Server

It's afternoon here - Good Afternoon!

Tip: To see how [explicitly requesting dependencies](#) in controllers makes code easier to test, learn more about [unit testing ASP.NET 5 applications](#).

ASP.NET 5's built-in dependency injection supports having only a single constructor for classes requesting services. If you have more than one constructor, you may get an exception stating:

An unhandled exception occurred while processing the request.

InvalidOperationException: Multiple constructors accepting all given argument types have been found in type 'ControllerDI.Controllers.HomeController'. There should only be one applicable constructor. Microsoft.Extensions.DependencyInjection.ActivatorUtilities.FindApplicableConstructor(Type instanceType, Type[] argumentTypes, ConstructorInfo& matchingConstructor, Nullable'1[]& parameterMap)

As the error message states, you can correct this problem having just a single constructor. You can also [replace the default dependency injection support with a third party implementation](#), many of which support multiple constructors.

6.5.3 Action Injection with FromServices

Sometimes you don't need a service for more than one action within your controller. In this case, it may make sense to inject the service as a parameter to the action method. This is done by marking the parameter with the attribute `[FromServices]` as shown here:

```
1 public IActionResult About([FromServices] IDateTime dateTime)
2 {
3     ViewData["Message"] = "Currently on the server the time is " + dateTime.Now;
4
5     return View();
6 }
```

6.5.4 Accessing Settings from a Controller

It's fairly common that you may want to access some application or configuration settings from a controller. Doing so should use the Options pattern described in [configuration](#). To request configuration settings via dependency injection from your controller, you shouldn't request a configuration or settings type directly. Instead, request an

`IOptions<T>` instance, where `T` is the configuration class you need. To work with the options pattern, you need to create a class that represents the options, such as this one:

```
1 namespace ControllerDI.Model
2 {
3     public class SampleWebSettings
4     {
5         public string Title { get; set; }
6         public int Updates { get; set; }
7     }
8 }
```

Then you need to configure the application to use the options model and add your configuration class to the services collection in `ConfigureServices`:

```
1 public Startup()
2 {
3     var builder = new ConfigurationBuilder()
4         .AddJsonFile("samplewebsettings.json");
5     Configuration = builder.Build();
6 }
7
8 public IConfigurationRoot Configuration { get; set; }
9
10 // This method gets called by the runtime. Use this method to add services to the container.
11 // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
12 public void ConfigureServices(IServiceCollection services)
13 {
14     // Required to use the Options<T> pattern
15     services.AddOptions();
16
17     // Add settings from configuration
18     services.Configure<SampleWebSettings>(Configuration);
19
20     // Uncomment to add settings from code
21     //services.Configure<SampleWebSettings>(settings =>
22     //{
23     //     settings.Updates = 17;
24     //});
25
26     services.AddMvc();
27
28     // Add application services.
29     services.AddTransient<IDateTime, SystemDateTime>();
30 }
```

Note: In the above listing, we are configuring the application to read the settings from a JSON-formatted file. You can also configure the settings entirely in code, as is shown in the commented code above. [Learn more about ASP.NET Configuration options](#)

Once you've specified a strongly-typed configuration object (in this case, `SampleWebSettings`) and added it to the services collection, you can request it from any Controller or Action method by requesting an instance of `IOptions<T>` (in this case, `IOptions<SampleWebSettings>`). The following code shows how one would request the settings from a controller:

```
1 public class SettingsController : Controller
2 {
3     private readonly SampleWebSettings _settings;
```

```
4      public SettingsController(IOptions<SampleWebSettings> settingsOptions )
5      {
6          _settings = settingsOptions.Value;
7      }
8
9
10     public IActionResult Index()
11     {
12         ViewData["Title"] = _settings.Title;
13         ViewData["Updates"] = _settings.Updates;
14         return View();
15     }
16 }
```

Following the Options pattern allows settings and configuration to be decoupled from one another, and ensures the controller is following [separation of concerns](#), since it doesn't need to know how or where to find the settings information. It also makes the controller easier to [unit test](#), since there is no [static cling](#) or direct instantiation of settings classes within the controller class.

6.6 Testing Controller Logic

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

6.7 Areas

By *Tom Archer*

[Areas](#) provide a way to separate a large MVC application into semantically-related groups of models, views, and controllers. Let's take a look at an example to illustrate how Areas are created and used. Let's say you have a store app that has two distinct groupings of controllers and views: Products and Services.

Instead of having all of the controllers located under the Controllers parent directory, and all the views located under the Views parent directory, you could use Areas to group your views and controllers according to the area (or logical grouping) with which they're associated.

- Project name
 - Areas
 - * Products
 - Controllers
 - HomeController.cs
 - Views
 - Home

```

    · Index.cshtml
* Services
    · Controllers
    · HomeController.cs
    · Views
    · Home
    · Index.cshtml

```

Looking at the preceding directory hierarchy example, there are a few guidelines to keep in mind when defining areas:

- A directory called *Areas* must exist as a child directory of the project.
- The *Areas* directory contains a subdirectory for each of your project's areas (*Products* and *Services*, in this example).
- Your controllers should be located as follows: `/Areas/[area]/Controllers/[controller].cs`
- Your views should be located as follows: `/Areas/[area]/Views/[controller]/[action].cshtml`

Note that if you have a view that is shared across controllers, it can be located in either of the following locations:

- `/Areas/[area]/Views/Shared/[action].cshtml`
- `/Views/Shared/[action].cshtml`

Once you've defined the folder hierarchy, you need to tell MVC that each controller is associated with an area. You do that by decorating the controller name with the `[Area]` attribute.

```

...
namespace MyStore.Areas.Products.Controllers
{
    [Area("Products")]
    public class HomeController : Controller
    {
        // GET: /<controller>/
        public IActionResult Index()
        {
            return View();
        }
    }
}

```

The final step is to set up a route definition that works with your newly created areas. The [Routing to Controller Actions](#) article goes into detail about how to create route definitions, including using conventional routes versus attribute routes. In this example, we'll use a conventional route. To do so, simply open the `Startup.cs` file and modify it by adding the highlighted route definition below.

```

...
app.UseMvc(routes =>
{
    routes.MapRoute(name: "areaRoute",
        template: "{area:exists}/{controller=Home}/{action=Index}");

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

```

Now, when the user browses to *http://<yourApp>/products*, the `Index` action method of the `HomeController` in the `Products` area will be invoked.

6.7.1 Linking between areas

To link between areas, you simply specify the area in which the controller is defined. If the controller is not a part of an area, use an empty string.

The following snippet shows how to link to a controller action that is defined within an area named *Products*.

```
@Html.ActionLink("See Products Home Page", "Index", "Home", new { area = "Products" }, null)
```

To link to a controller action that is not part of an area, simply specify an empty string for the area.

```
@Html.ActionLink("Go to Home Page", "Index", "Home", new { area = "" }, null)
```

6.7.2 Summary

Areas are a very useful tool for grouping semantically-related controllers and actions under a common parent folder. In this article, you learned how to set up your folder hierarchy to support Areas, how to specify the `[Area]` attribute to denote a controller as belonging to a specified area, and how to define your routes with areas.

6.8 Working with the Application Model

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

7.1 Response Caching

By Steve Smith

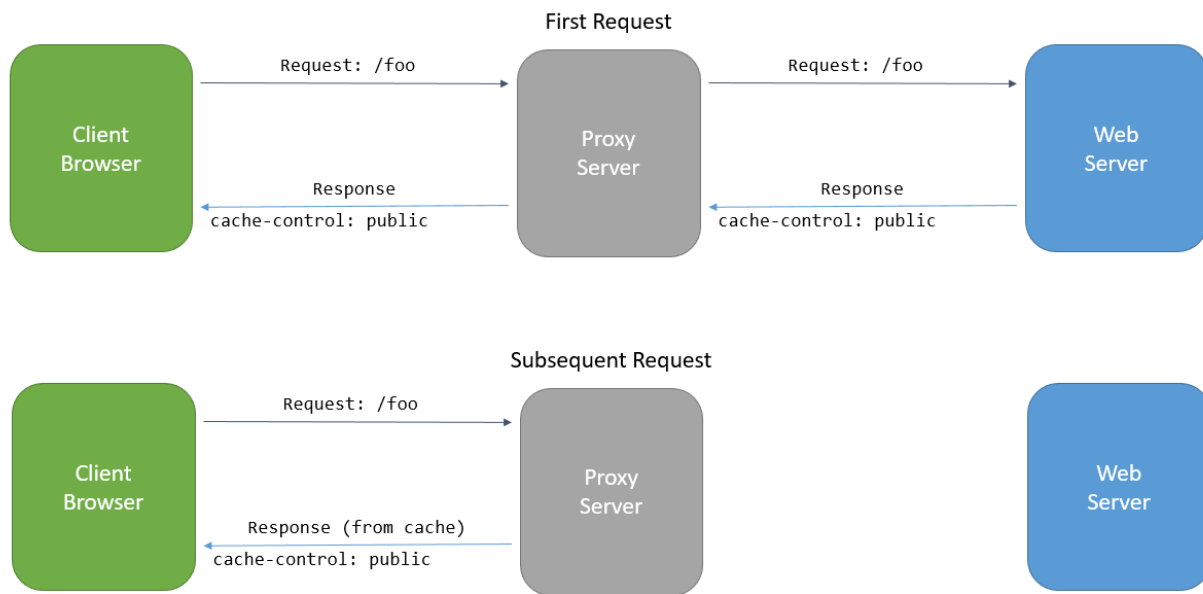
In this article:

- *What is Response Caching*
- `ResponseCache` Attribute

[View or download sample from GitHub.](#)

7.1.1 What is Response Caching

Response caching refers to specifying cache-related headers on HTTP responses made by ASP.NET MVC actions. These headers specify how you want client and intermediate (proxy) machines to cache responses to certain requests (if at all). This can reduce the number of requests a client or proxy makes to the web server, since future requests for the same action may be served from the client or proxy's cache. In this case, the request is never made to the web server.



The primary HTTP header used for caching is `Cache-Control`. The [HTTP 1.1 specification](#) details many options for this directive. Three common directives are:

public Indicates that the response may be cached.

private Indicates the response is intended for a single user and **must not** be cached by a shared cache. The response could still be cached in a private cache (for instance, by the user's browser).

no-cache Indicates the response **must not** be used by a cache to satisfy any subsequent request (without successful revalidation with the origin server).

Note: **Response caching does not cache responses on the web server.** It differs from [output caching](#), which would cache responses in memory on the server in earlier versions of ASP.NET and ASP.NET MVC. Output caching middleware is planned to be added to ASP.NET MVC 6 in a future release.

Additional HTTP headers used for caching include `Pragma` and `Vary`, which are described below. Learn more about [Caching in HTTP from the specification](#).

7.1.2 ResponseCache Attribute

The [ResponseCache Attribute](#) is used to specify how a controller action's headers should be set to control its cache behavior. The attribute has the following properties, all of which are optional unless otherwise noted.

Duration int The maximum duration (in seconds) the response should be cached. **Required** unless `NoStore` is `true`.

Location ResponseCacheLocation The location where the response may be cached. May be `Any`, `None`, or `Client`. Default is `Any`.

NoStore bool Determines whether the value should be stored or not, and overrides other property values. When `true`, `Duration` is ignored and `Location` is ignored for values other than `None`.

VaryByHeader string When set, a `vary` response header will be written with the response.

CacheProfileName string When set, determines the name of the cache profile to use.

Order int The order of the filter (from [IOrderedFilter](#)).

The `ResponseCacheAttribute` is used to configure and create (via [IFilterFactory](#)) a [ResponseCacheFilter](#) which performs the work of writing the appropriate HTTP headers to the response. The filter will first remove any existing headers for `Vary`, `Cache-Control`, and `Pragma`, and then will write out the appropriate headers based on the properties set in the `ResponseCacheAttribute`.

The Vary Header

This header is only written when the `VaryByHeader` property is set, in which case it is set to that property's value.

NoStore and Location.None

`NoStore` is a special property that overrides most of the other properties. When this property is `true`, the `Cache-Control` header will be set to "no-store". Additionally, if `Location` is set to `None`, then `Cache-Control` will be set to "no-store, no-cache" and `Pragma` is likewise set to `no-cache`. (If `NoStore` is `false` and `Location` is `None`, then both `Cache-Control` and `Pragma` will be set to `no-cache`).

A good scenario in which to set `NoStore` to `true` is error pages. It's unlikely you would want to respond to a user's request with the error response a different user previously generated, and such responses may include stack traces and other sensitive information that shouldn't be stored on intermediate servers. For example:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public IActionResult Error()
{
    return View();
}
```

This will result in the following headers:

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

Location and Duration

To enable caching, `Duration` must be set to a positive value and `Location` must be either `Any` (the default) or `Client`. In this case, the `Cache-Control` header will be set to the location value followed by the “max-age” of the response.

Note: `Location`’s options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted previously, setting `Location` to `None` will set both `Cache-Control` and `Pragma` headers to `no-cache`.

Below is an example showing the headers produced by setting `Duration` and leaving the default `Location` value.

```
[ResponseCache(Duration=60)]
```

Produces the following headers:

```
Cache-Control: public,max-age=60
```

Cache Profiles

Instead of duplicating `ResponseCache` settings on many controller action attributes, cache profiles can be configured as options when setting up MVC in the `ConfigureServices` method in `Startup`. Values found in a referenced cache profile will be used as the defaults by the `ResponseCache` attribute, and will be overridden by any properties specified on the attribute.

Setting up a cache profile:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddMvc(options =>
4     {
5         options.CacheProfiles.Add("Default",
6             new CacheProfile()
7             {
8                 Duration=60
9             });
10        options.CacheProfiles.Add("Never",
11            new CacheProfile()
12            {
13                Location = ResponseCacheLocation.None,
14                NoStore = true
15            });
16    });
17 }
```

```
16     });  
17 }
```

Referencing a cache profile:

```
[ResponseCache(CacheProfileName="Default")]
```

Tip: The `ResponseCache` attribute can be applied both to actions (methods) as well as controllers (classes). Method-level attributes will override the settings specified in class-level attributes.

In the following example, a class-level attribute specifies a duration of 30 while a method-level attributes references a cache profile with a duration set to 60.

```
1 [ResponseCache(Duration=30)]  
2 public class HomeController : Controller  
3 {  
4     [ResponseCache(CacheProfileName = "Default")]  
5     public IActionResult Index()  
6     {  
7         return View();  
8     }  
}
```

The resulting header:

```
Cache-Control: public,max-age=60
```

Security

8.1 Authorization Filters

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

8.2 Enforcing SSL

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

8.3 Anti-Request Forgery

Note: We are currently working on this topic.

We welcome your input to help shape the scope and approach. You can track the status and provide input on this [issue](#) at GitHub.

If you would like to review early drafts and outlines of this topic, please leave a note with your contact information in the [issue](#).

Learn more about how you can [contribute](#) on GitHub.

8.4 Specifying a CORS Policy

By [Mike Wasson](#)

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the *same-origin policy*, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites make cross-origin requests to your web app.

[Cross Origin Resource Sharing](#) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. This topic shows how to enable CORS in your ASP.NET MVC 6 application. (For background on CORS, see [How CORS works](#).)

8.4.1 Add the CORS package

In your `project.json` file, add the following:

```
"dependencies": {  
  "Microsoft.AspNet.Cors": "6.0.0-beta8"  
},
```

8.4.2 Configure CORS

To configure CORS, call `AddCors` in the `ConfigureServices` method of your `Startup` class, as shown here:

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc();  
    services.AddCors(options =>  
    {  
        // Define one or more CORS policies  
        options.AddPolicy("AllowSpecificOrigin",  
            builder =>  
            {  
                builder.WithOrigins("http://example.com");  
            });  
    });  
}
```

This example defines a CORS policy named “AllowSpecificOrigin” that allows cross-origin requests from “http://example.com” and no other origins. The lambda takes a `CorsPolicyBuilder` object. To learn more about the various CORS policy settings, see [CORS policy options](#).

8.4.3 Apply CORS Policies

The next step is to apply the policies. You can apply a CORS policy per action, per controller, or globally for all controllers in your application.

Per action

Add the `[EnableCors]` attribute to the action. Specify the policy name.

```
public class HomeController : Controller
{
    [EnableCors("AllowSpecificOrigin")]
    public IActionResult Index()
    {
        return View();
    }
}
```

Per controller

Add the `[EnableCors]` attribute to the controller class. Specify the policy name.

```
[EnableCors("AllowSpecificOrigin")]
public class HomeController : Controller
{
}
```

Globally

Add the `CorsAuthorizationFilterFactory` filter to the global filter collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.Configure<MvcOptions>(options =>
    {
        options.Filters.Add(new CorsAuthorizationFilterFactory("AllowSpecificOrigin"));
    });
}
```

The precedence order is: Action, controller, global. Action-level policies take precedence over controller-level policies, and controller-level policies take precedence over global policies.

Disable CORS

To disable CORS for a controller or action, use the `[DisableCors]` attribute.

```
[DisableCors]
public IActionResult About()
{
    return View();
}
```

Migration

9.1 Migrating From ASP.NET MVC 5 to MVC 6

By [Rick Anderson](#), [Daniel Roth](#), [Steve Smith](#), and [Scott Addie](#)

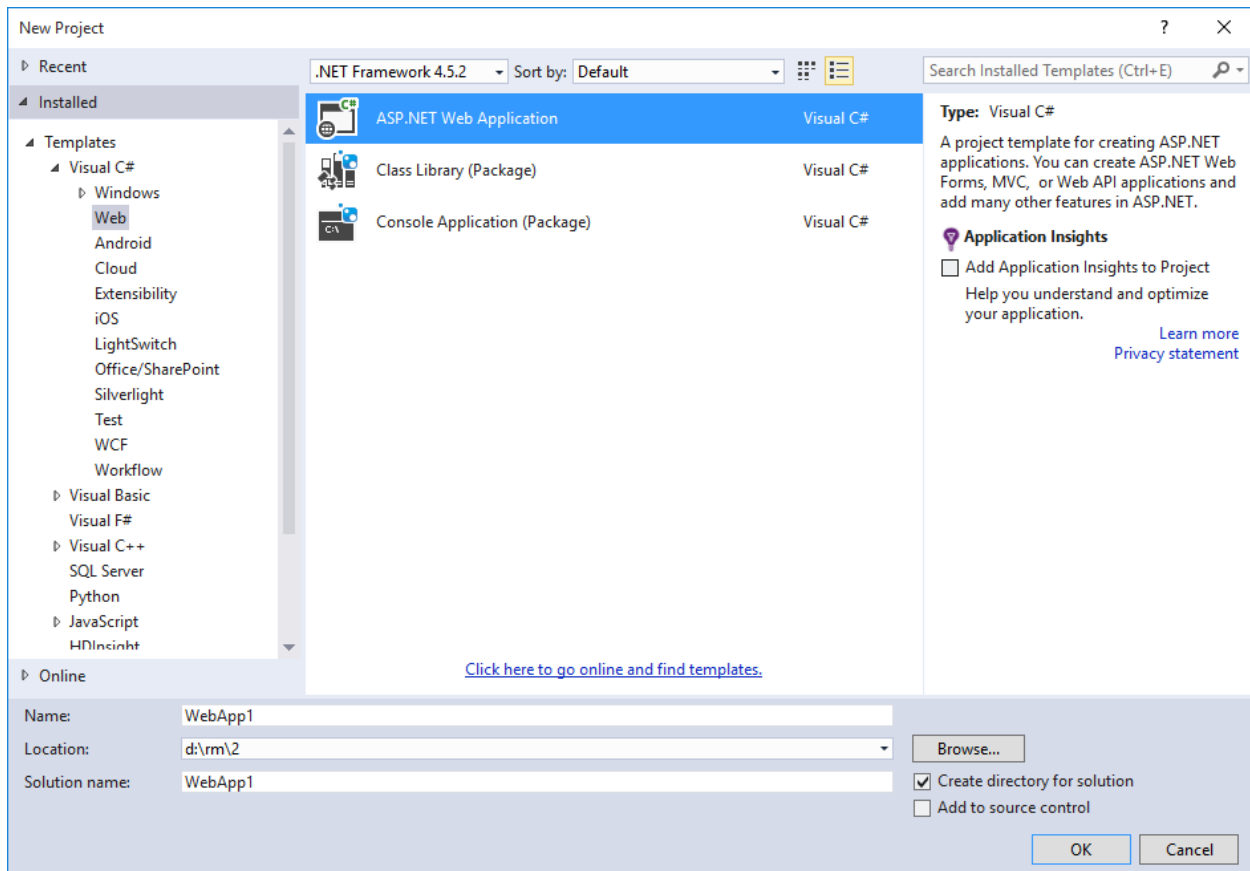
This article shows how to get started migrating an ASP.NET MVC 5 project to ASP.NET MVC 6. In the process, it highlights many of the things that have changed from MVC 5 to MVC 6. Migrating from MVC 5 to MVC 6 is a multiple step process and this article covers the initial setup, basic controllers and views, static content, and client-side dependencies. Additional articles cover migrating ASP.NET Identity models, and startup and configuration code found in many MVC 5 projects.

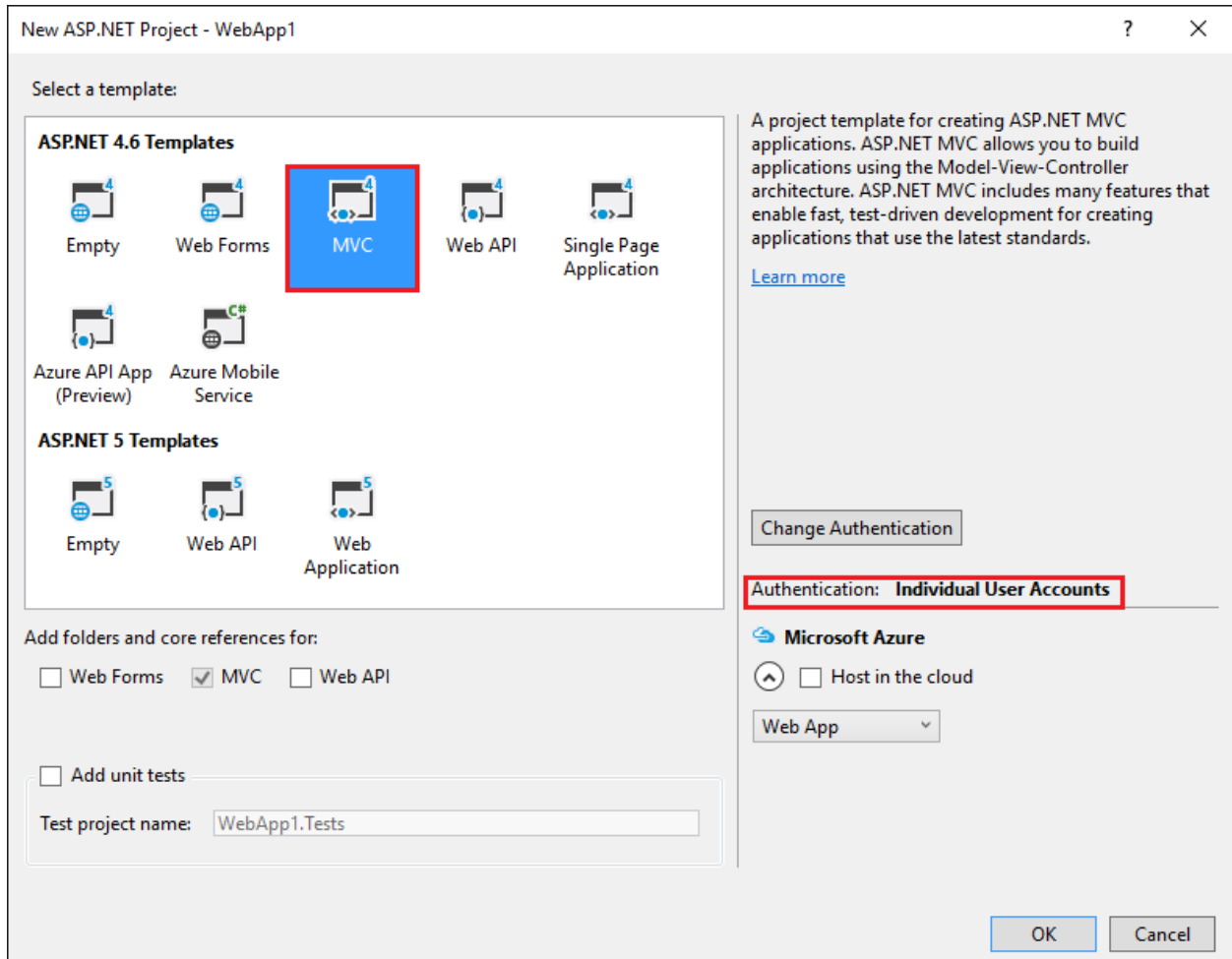
In this article:

- *Create the starter MVC 5 project*
- *Create the MVC 6 project*
- *Configure the site to use MVC*
- *Add a controller and view*
- *Controllers and views*
- *Static content*
- *Gulp*
- *NPM*
- *Migrate the layout file*
- *Configure Bundling*
- *Additional Resources*

9.1.1 Create the starter MVC 5 project

To demonstrate the upgrade, we'll start by creating a new ASP.NET MVC 5 app. Create it with the name *WebApp1* so the namespace will match the MVC 6 project we create in the next step.

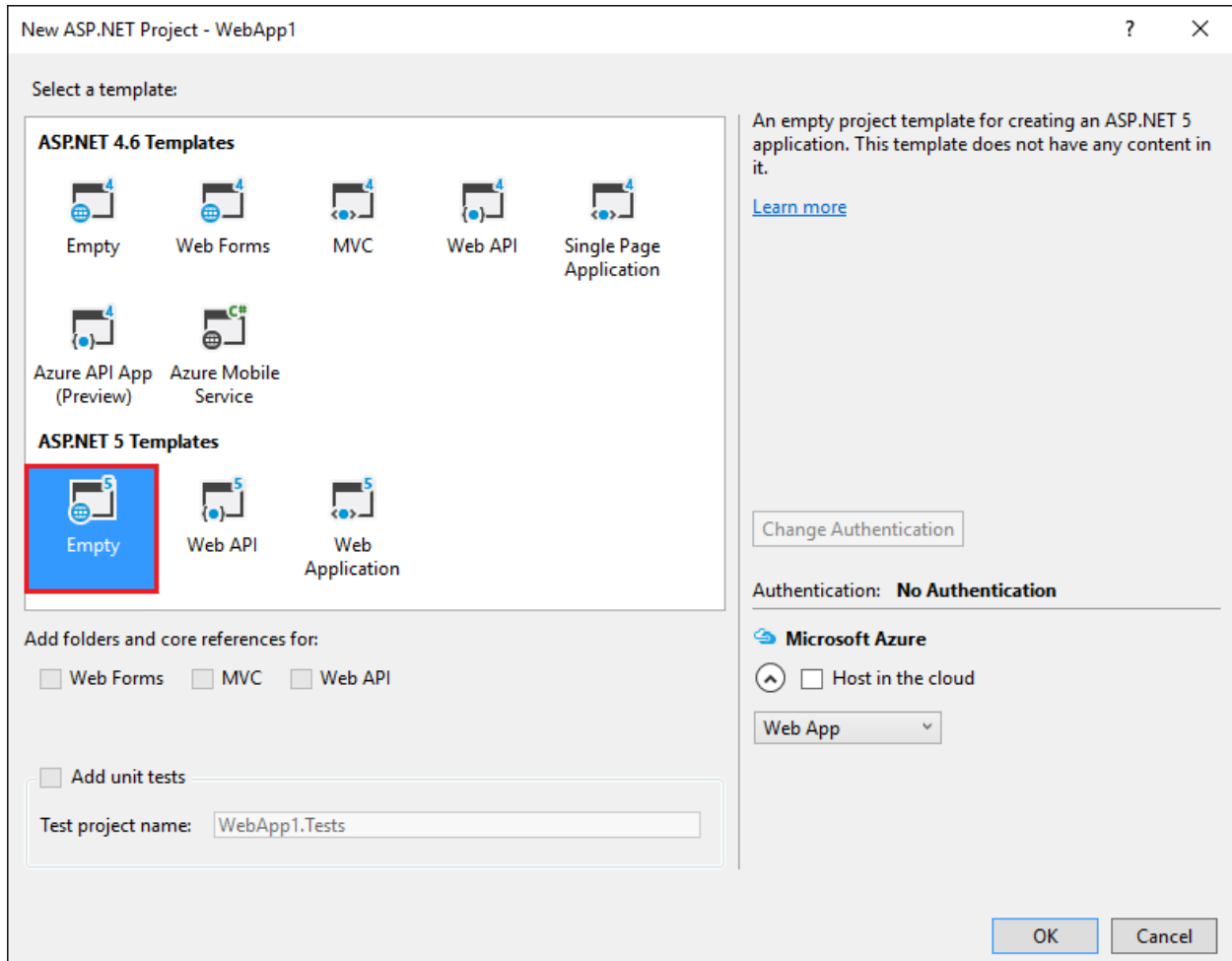




Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio will display the new solution name (*Mvc5*), which will make it easier to tell this project from the next project. You might need to exit Visual Studio and then reload the project to see the new name.

9.1.2 Create the MVC 6 project

Create a new *empty* MVC 6 web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. You'll have to create this project in a different directory than the previous project to use the same name.



- *Optional:* Create a new MVC 6 app named *WebApp1* with authentication set to **Individual User Accounts**. Rename this app *FullMVC6*. Creating this project will save you time in the conversion. You can look at the template generated code to see the end result or to copy code to the conversion project. It's also helpful when you get stuck on a conversion step to compare with the template generated project.

9.1.3 Configure the site to use MVC

- Open the *project.json* file and add `Microsoft.AspNet.Mvc` and `Microsoft.AspNet.StaticFiles` to the dependencies property and the scripts section as highlighted below:

```

1 {
2   "version": "1.0.0-*",
3   "compilationOptions": {
4     "emitEntryPoint": true
5   },
6
7   "dependencies": {
8     "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final",
9     "Microsoft.AspNet.Mvc": "6.0.0-rc1-final",
10    "Microsoft.AspNet.IISPlatformHandler": "1.0.0-rc1-final",
11    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final"
12  },
13 }
```

```

14  "commands": {
15      "web": "Microsoft.AspNet.Server.Kestrel"
16  },
17
18  "frameworks": {
19      "dnx451": { },
20      "dnxcore50": { }
21  },
22
23  "exclude": [
24      "wwwroot",
25      "node_modules"
26  ],
27  "publishExclude": [
28      "**.user",
29      "**.vspscc"
30  ],
31  "scripts": {
32      "prepublish": [ "npm install", "bower install", "gulp clean", "gulp min" ]
33  }
34  }

```

Microsoft.AspNet.StaticFiles is the static file handler. The ASP.NET runtime is modular, and you must explicitly opt in to serve static files (see [Working with Static Files](#)).

The `scripts` section is used to denote when specified build automation scripts should run. Visual Studio now has built-in support for running scripts before and after specific events. The `scripts` section above specifies [NPM](#), [Bower](#) and [Gulp](#) scripts should run on the `prepublish` stage. We'll talk about NPM, Bower, and Gulp later in the tutorial. Note the trailing `","` added to the end of the `publishExclude` section.

For more information, see [project.json](#) and [Introducing .NET Core](#).

- Open the `Startup.cs` file and change the code to match the following:

```

1  public class Startup
2  {
3      // This method gets called by the runtime. Use this method to add services to the container.
4      // For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkId=301871
5      public void ConfigureServices(IServiceCollection services)
6      {
7          services.AddMvc();
8      }
9
10     // This method gets called by the runtime. Use this method to configure the HTTP request pipeline
11     public void Configure(IApplicationBuilder app)
12     {
13         app.UseIISPlatformHandler();
14
15         app.UseStaticFiles();
16
17         app.UseMvc(routes =>
18         {
19             routes.MapRoute(
20                 name: "default",
21                 template: "{controller=Home}/{action=Index}/{id?}");
22         });
23     }

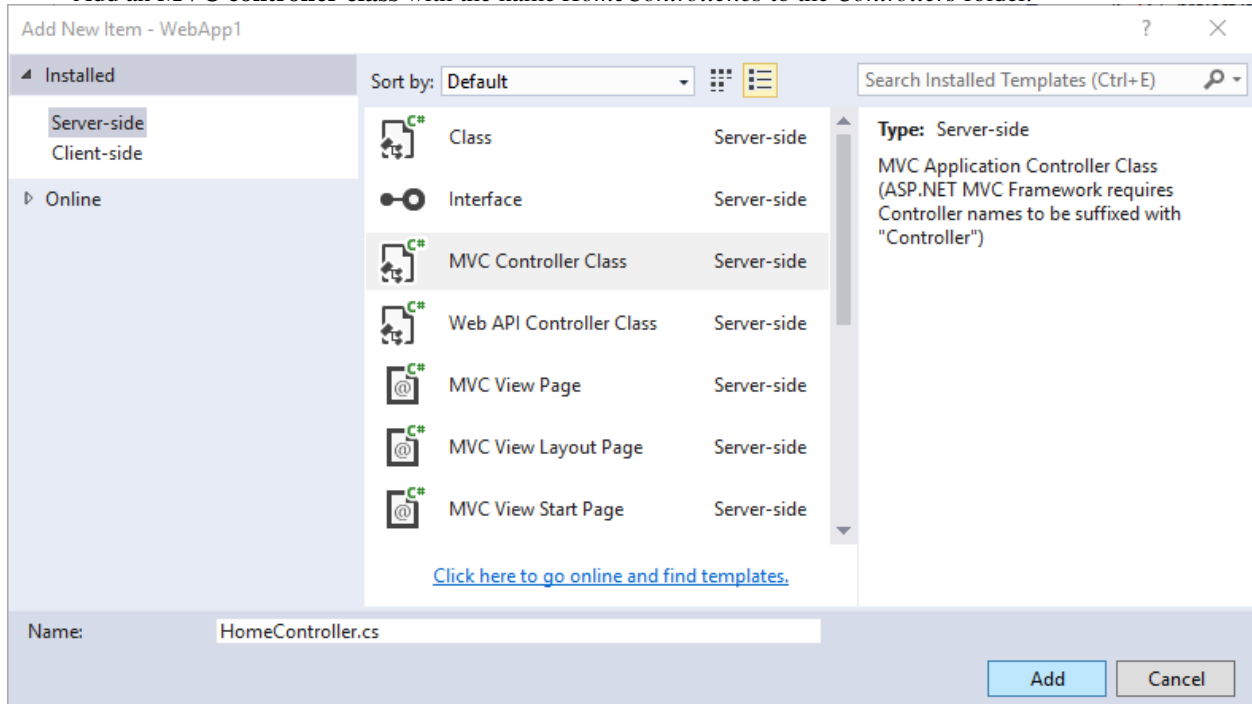
```

`UseStaticFiles` adds the static file handler. As mentioned previously, the ASP.NET runtime is modular, and you must explicitly opt in to serve static files. For more information, see [Application Startup](#) and [Routing](#).

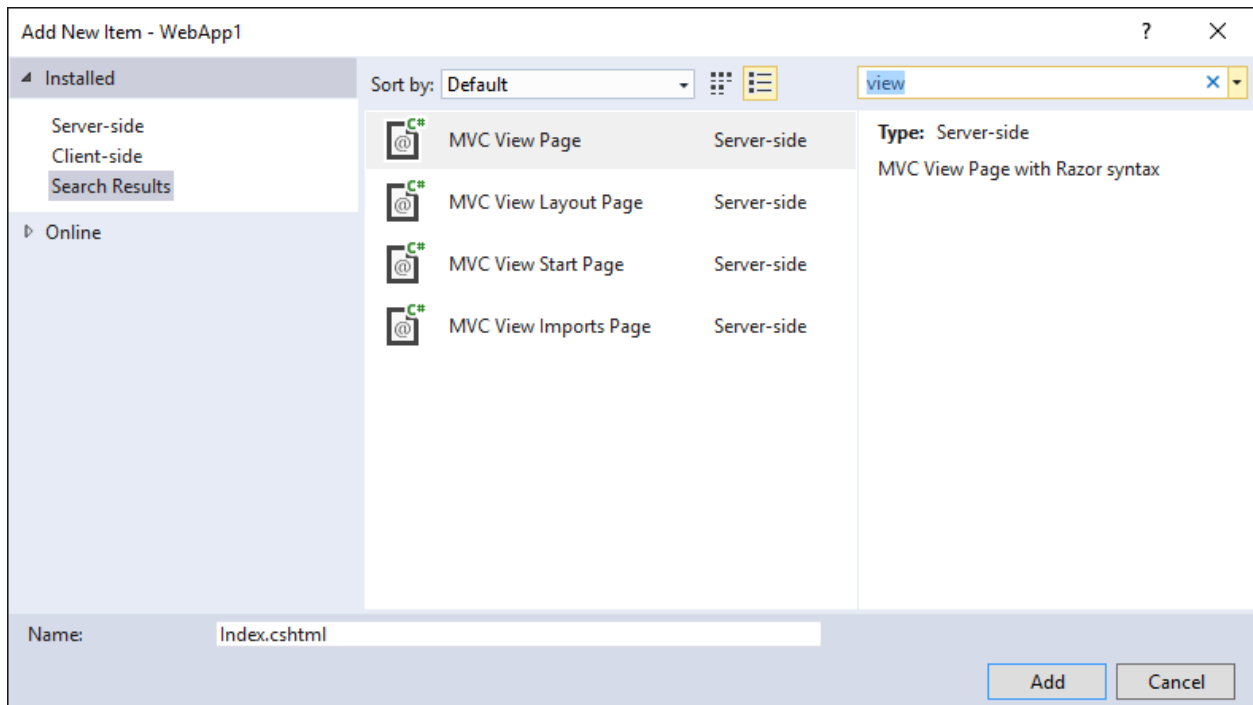
9.1.4 Add a controller and view

In this section, you'll add a minimal controller and view to serve as placeholders for the MVC 5 controller and views you'll migrate in the next section.

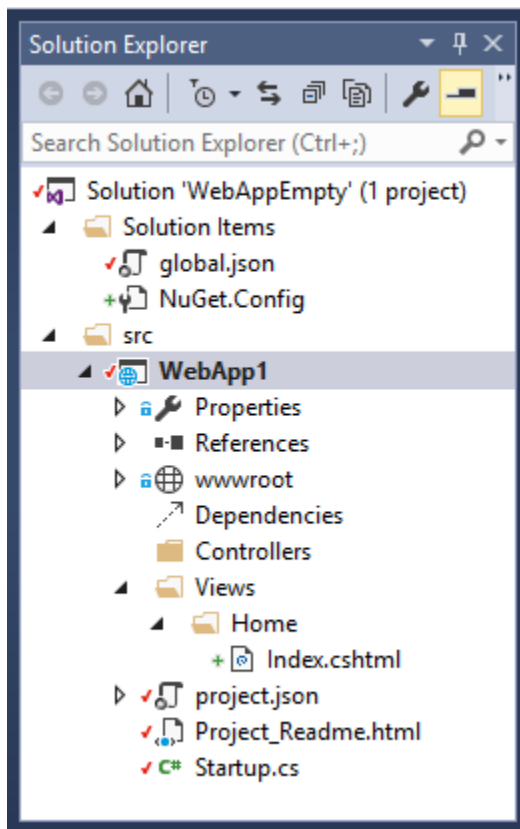
- Add a *Controllers* folder.
- Add an **MVC controller class** with the name *HomeController.cs* to the *Controllers* folder.



- Add a *Views* folder.
- Add a *Views/Home* folder.
- Add an *Index.cshtml* MVC view page to the *Views/Home* folder.



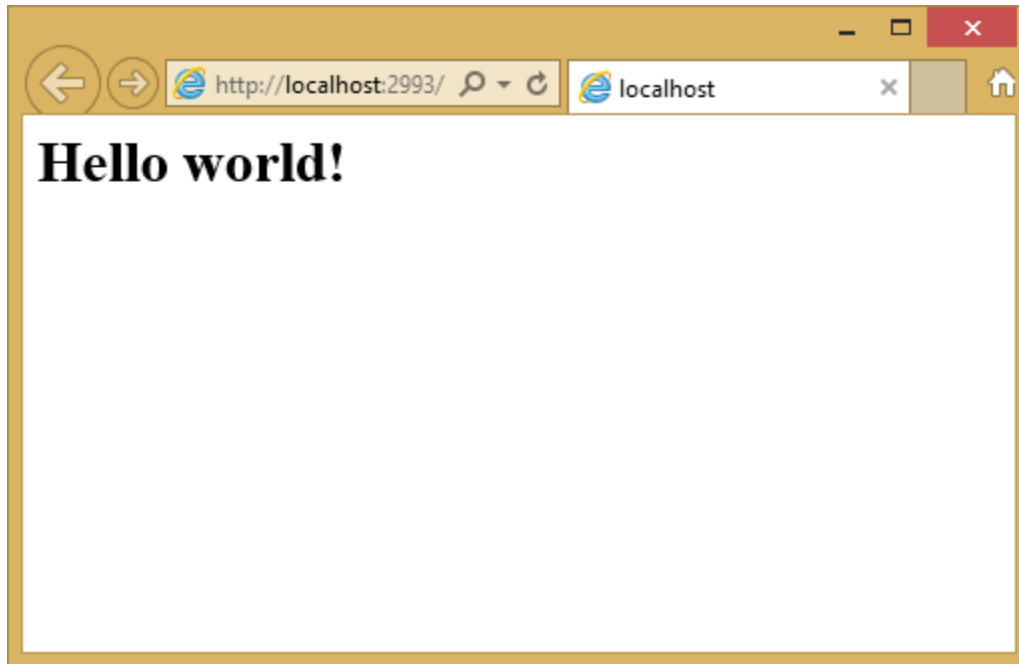
The project structure is shown below:



Replace the contents of the *Views/Home/Index.cshtml* file with the following:

```
<h1>Hello world!</h1>
```

Run the app.



See [Controllers](#) and [Views](#) for more information.

Now that we have a minimal working MVC 6 project, we can start migrating functionality from the MVC 5 project. We will need to move the following:

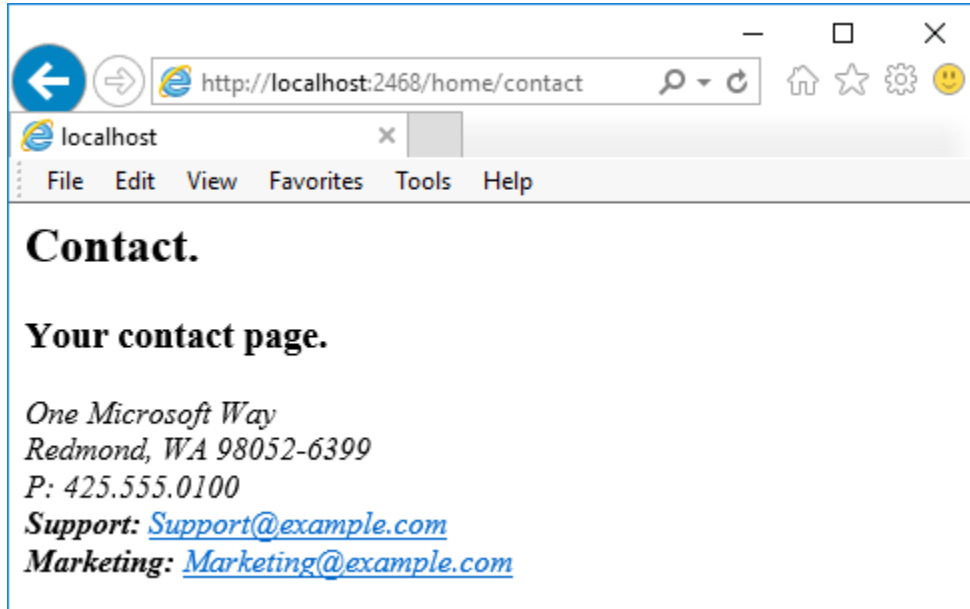
- client-side content (CSS, fonts, and scripts)
- controllers
- views
- models
- bundling
- filters
- Log in/out, identity (This will be done in the next tutorial.)

9.1.5 Controllers and views

- Copy each of the methods from the MVC 5 `HomeController` to the MVC 6 `HomeController`. Note that in MVC 5, the built-in template's controller action method return type is `ActionResult`; in MVC 6, the templates generate `IActionResult` methods. `ActionResult` is the only implementation of `IActionResult`, so there is no need to change the return type of your action methods.
- Delete the `Views/Home/Index.cshtml` view in the MVC 6 project.
- Copy the `About.cshtml`, `Contact.cshtml`, and `Index.cshtml` Razor view files from the MVC 5 project to the MVC 6 project.
- Run the MVC 6 app and test each method. We haven't migrated the layout file or styles yet, so the rendered views will only contain the content in the view files. You won't have the layout file generated links for the

About and Contact views, so you'll have to invoke them from the browser (replace **2468** with the port number used in your project).

- `http://localhost:2468/home/about`
- `http://localhost:2468/home/contact`



Note the lack of styling and menu items. We'll fix that in the next section.

9.1.6 Static content

In previous versions of MVC (including MVC 5), static content was hosted from the root of the web project and was intermixed with server-side files. In MVC 6, static content is hosted in the `wwwroot` folder. You'll want to copy the static content from your MVC 5 app to the `wwwroot` folder in your MVC 6 project. In this sample conversion:

- Copy the `favicon.ico` file from the MVC 5 project to the `wwwroot` folder in the MVC 6 project.

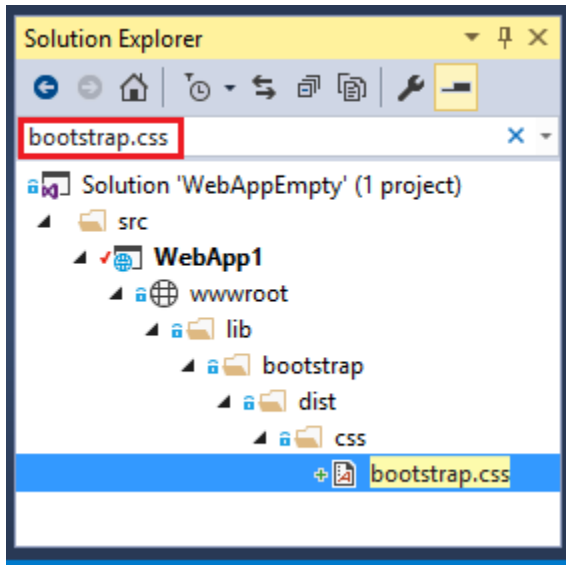
The MVC 5 project uses [Bootstrap](#) for its styling and stores the Bootstrap files in the `Content` and `Scripts` folders. The template-generated MVC 5 project references Bootstrap in the layout file (`Views/Shared/_Layout.cshtml`). You could copy the `bootstrap.js` and `bootstrap.css` files from the MVC 5 project to the `wwwroot` folder in the new project, but that approach doesn't use the improved mechanism for managing client-side dependencies in ASP.NET 5.

In the new project, we'll add support for Bootstrap (and other client-side libraries) using [Bower](#):

- Add a [Bower](#) configuration file named `bower.json` to the project root (Right-click on the project, and then **Add > New Item > Bower Configuration File**). Add [Bootstrap](#) and [jQuery](#) to the file (see the highlighted lines below).

```
{
  "name": "ASP.NET",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.5",
    "jquery": "2.1.4"
  }
}
```

Upon saving the file, Bower will automatically download the dependencies to the `wwwroot/lib` folder. You can use the **Search Solution Explorer** box to find the path of the assets.

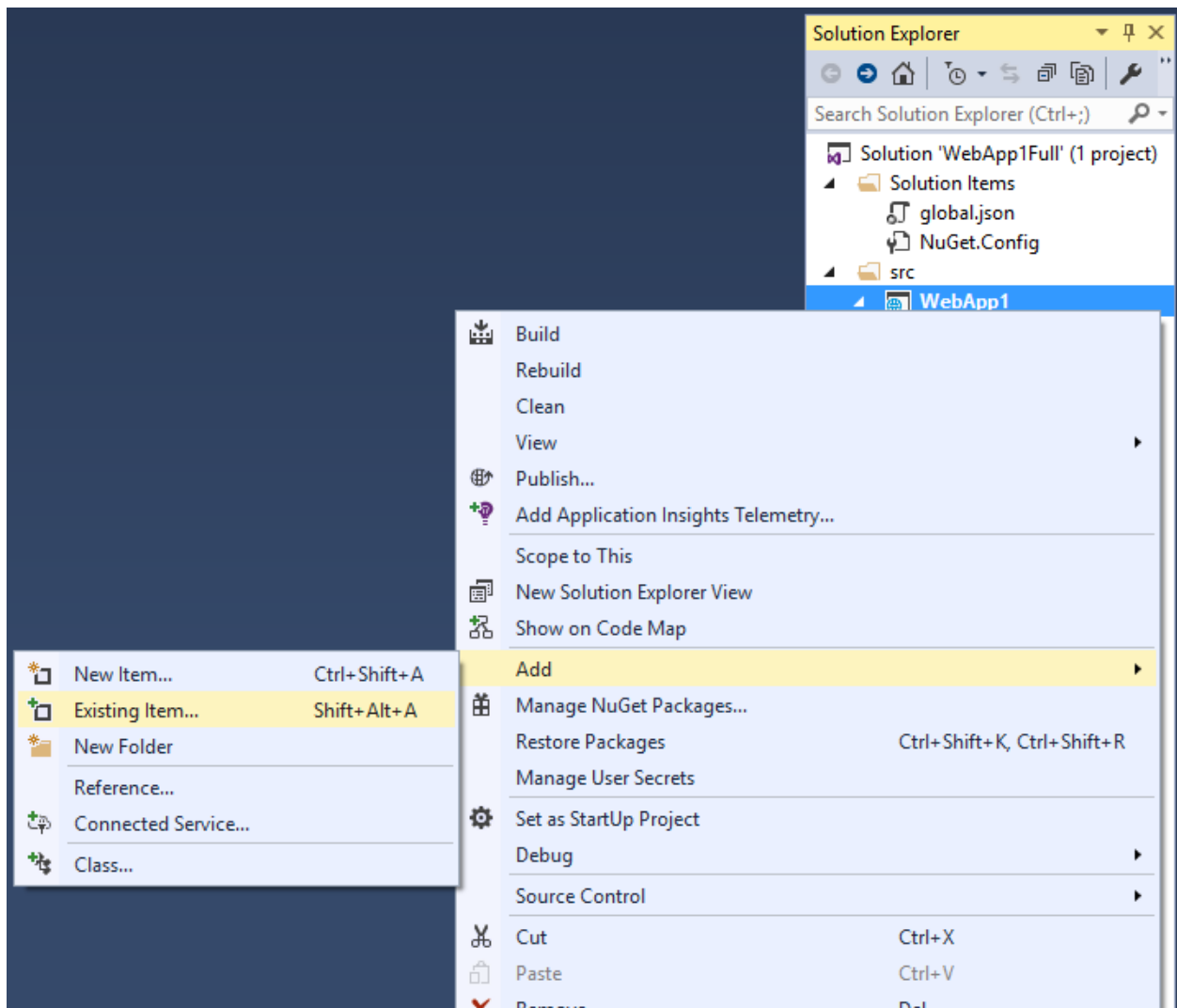


See [Manage Client-Side Packages with Bower](#) for more information.

9.1.7 Gulp

When you create a new web app using the ASP.NET 5 Web Application template, the project is setup to use [Gulp](#). Gulp is a streaming build system for client-side code (HTML, LESS, SASS, etc.). The included `gulpfile.js` in the project contains JavaScript that defines a set of gulp tasks that you can set to run automatically on build events or you can run manually using the **Task Runner Explorer** in Visual Studio. In this section, we'll show how to use the MVC 6 template's generated `gulpfile.js` file to bundle and minify the JavaScript and CSS files in the project.

If you created the optional *FullMVC6* project (a new ASP.NET MVC 6 web app with Individual User Accounts), add `gulpfile.js` from that project to the project we are updating. In Solution Explorer, right-click the web app project and choose **Add > Existing Item**.



Navigate to *gulpfile.js* from the new ASP.NET MVC 6 web app with Individual User Accounts and add the *gulpfile.js* file. Alternatively, right-click the web app project and choose **Add > New Item**. Select **Gulp Configuration File**, and name the file *gulpfile.js*. Replace the contents of the gulp file with the following:

```

/// <binding Clean='clean' />
"use strict";

var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.minCss = paths.webroot + "css/**/*.min.css";

```

```
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.concatCssDest = paths.webroot + "css/site.min.css";

gulp.task("clean:js", function (cb) {
    rimraf(paths.concatJsDest, cb);
});

gulp.task("clean:css", function (cb) {
    rimraf(paths.concatCssDest, cb);
});

gulp.task("clean", ["clean:js", "clean:css"]);

gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("min", ["min:js", "min:css"]);
```

The code above performs these functions:

- Cleans (deletes) the target files.
- Minifies the JavaScript and CSS files.
- Bundles (concatenates) the JavaScript and CSS files.

See [Using Gulp with ASP.NET 5 and Visual Studio](#).

9.1.8 NPM

NPM (Node Package Manager) is a package manager which is used to acquire tooling such as [Bower](#) and [Gulp](#); and, it is fully supported in Visual Studio 2015. We'll use NPM to manage Gulp dependencies.

If you created the optional *FullMVC6* project, add the *package.json* NPM file from that project to the project we are updating. The *package.json* NPM file lists the dependencies for the client-side build processes defined in *gulpfile.js*. Right-click the web app project, choose **Add > Existing Item**, and add the *package.json* NPM file. Alternatively, you can add a new NPM configuration file as follows:

1. In Solution Explorer, right-click the project.
2. Select **Add > New Item**.
3. Select **NPM Configuration File**.
4. Leave the default name: *package.json*.
5. Click **Add**.

Open the *package.json* file, and replace the contents with the following:

```
{
  "name": "ASP.NET",
  "version": "0.0.0",
  "devDependencies": {
    "gulp": "3.8.11",
    "gulp-concat": "2.5.2",
    "gulp-cssmin": "0.1.7",
    "gulp-uglify": "1.2.0",
    "rimraf": "2.2.8"
  }
}
```

Right-click on *gulpfile.js* and select **Task Runner Explorer**. Double-click on a task to run it.

For more information, see [Client-Side Development in ASP.NET 5](#).

9.1.9 Migrate the layout file

- Copy the *_ViewStart.cshtml* file from the MVC 5 project's *Views* folder into the MVC 6 project's *Views* folder. The *_ViewStart.cshtml* file has not changed in MVC 6.
- Create a *Views/Shared* folder.
- Copy the *_Layout.cshtml* file from the MVC 5 project's *Views/Shared* folder into the MVC 6 project's *Views/Shared* folder.

Open *_Layout.cshtml* file and make the following changes (the completed code is shown below):

- Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load *bootstrap.css* (see below)
- Remove `@Scripts.Render("~/bundles/modernizr")`
- Comment out the `@Html.Partial("_LoginPartial")` line (surround the line with `@*...*@`) - we'll return to it in a future tutorial
- Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below)
- Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below)

The replacement CSS link:

```
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
```

The replacement script tags:

```
<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
```

The updated *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@ViewBag.Title - My ASP.NET Application</title>
  <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
```

```

<div class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navb
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class
    </div>
    <div class="navbar-collapse collapse">
      <ul class="nav navbar-nav">
        <li>@Html.ActionLink("Home", "Index", "Home")</li>
        <li>@Html.ActionLink("About", "About", "Home")</li>
        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
      </ul>
      @*@Html.Partial("_LoginPartial")*@
    </div>
  </div>
</div>
<div class="container body-content">
  @RenderBody()
  <hr />
  <footer>
    <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
  </footer>
</div>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
@RenderSection("scripts", required: false)
</body>
</html>

```

View the site in the browser. It should now load correctly, with the expected styles in place.

9.1.10 Configure Bundling

The MVC 5 starter web template utilized the MVC runtime support for bundling. In ASP.NET MVC 6, this functionality is performed as part of the build process using [Gulp](#). We've previously configured bundling and minification; all that's left is to change the references to Bootstrap, jQuery and other assets to use the bundled and minified versions. You can see how this is done in the layout file (*Views/Shared/_Layout.cshtml*) of the full template project. See [Bundling and Minification](#) for more information.

9.1.11 Additional Resources

- [Migrating an ASP.NET MVC 5 App to ASP.NET 5](#)
- [Using Gulp](#)
- [Client-Side Development in ASP.NET 5](#)
- [Manage Client-Side Packages with Bower](#)
- [Bundling and Minification](#)
- [Bootstrap for ASP.NET 5](#)

9.2 Migrating Configuration From ASP.NET MVC 5 to MVC 6

By Steve Smith, Scott Addie

In the previous article, we began [migrating an ASP.NET MVC 5 project to MVC 6](#). In this article, we migrate the configuration feature from ASP.NET MVC 5 to ASP.NET MVC 6.

In this article:

- *Setup Configuration*
- *Migrate Configuration Settings from web.config*
- *Summary*

You can download the finished source from the project created in this article [here](#).

9.2.1 Setup Configuration

ASP.NET 5 and ASP.NET MVC 6 no longer use the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC 5, a *Startup.cs* file was included in the root of the project; and, it was called using an `OwinStartupAttribute` when the application started. ASP.NET 5 (and ASP.NET MVC 6) have adopted this approach completely, placing all startup logic in the *Startup.cs* file.

The *web.config* file has also been replaced in ASP.NET 5. Configuration itself can now be configured, as part of the application startup procedure described in *Startup.cs*. Configuration can still utilize XML files, but typically ASP.NET 5 projects will place configuration values in a JSON-formatted file, such as *appsettings.json*. ASP.NET 5's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that should not be checked into source control.

For this article, we are starting with the partially-migrated ASP.NET MVC 6 project from [the previous article](#). To setup configuration using the default MVC 6 settings, add the following constructor and property to the *Startup.cs* class located in the root of the project:

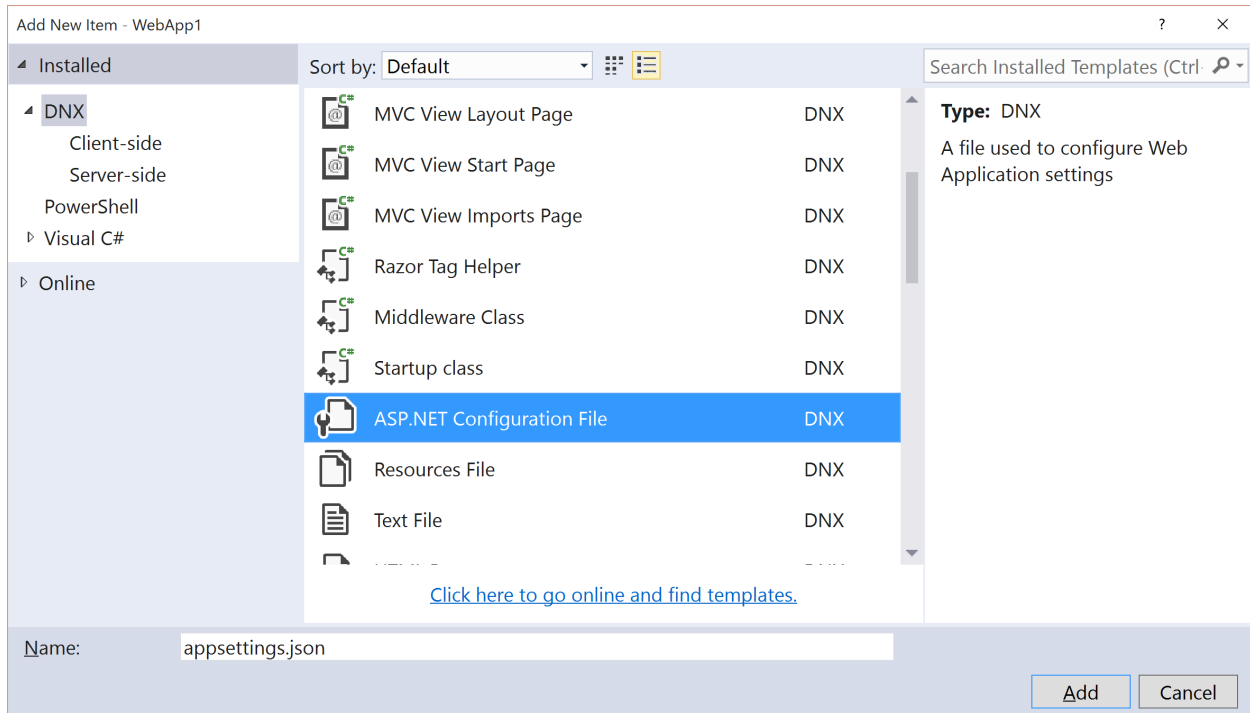
```

1 public Startup(IHostingEnvironment env)
2 {
3     // Set up configuration sources.
4     var builder = new ConfigurationBuilder()
5         .AddJsonFile("appsettings.json")
6         .AddEnvironmentVariables();
7     Configuration = builder.Build();
8 }
9
10 public IConfigurationRoot Configuration { get; set; }
```

Note that at this point the *Startup.cs* file will not compile, as we still need to add the following `using` statement:

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

Add an *appsettings.json* file to the root of the project using the appropriate item template:



9.2.2 Migrate Configuration Settings from web.config

Our ASP.NET MVC 5 project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our MVC 6 project, we are going to store this information in the *appsettings.json* file. Open *appsettings.json*, and note that it already includes the following:

```

1 {
2     "Data": {
3         "DefaultConnection": {
4             "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trust
5         }
6     }
7 }
```

In the highlighted line depicted above, change the name of the database from `_CHANGE_ME`. We are going to point to a new database, which will be named *NewMvc6Project* to match our migrated project name.

9.2.3 Summary

ASP.NET 5 places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

9.3 Migrating From ASP.NET Web API 2 to MVC 6

By Steve Smith, Scott Addie

ASP.NET Web API 2 was separate from ASP.NET MVC 5, with each using their own libraries for dependency resolution, among other things. In MVC 6, Web API has been merged with MVC, providing a single, consistent way of building web applications. In this article, we demonstrate the steps required to migrate from an ASP.NET Web API 2 project to MVC 6.

In this article:

- *Review Web API 2 Project*
- *Create the Destination Project*
- *Migrate Configuration*
- *Migrate Models and Controllers*

You can view the finished source from the project created in this article [on GitHub](#).

9.3.1 Review Web API 2 Project

This article uses the sample project, *ProductsApp*, created in the article [Getting Started with ASP.NET Web API 2 \(C#\)](#) as its starting point. In that project, a simple Web API 2 project is configured as follows.

In *Global.asax.cs*, a call is made to `WebApiConfig.Register`:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web;
5 using System.Web.Http;
6 using System.Web.Routing;
7
8 namespace ProductsApp
9 {
10     public class WebApiApplication : System.Web.HttpApplication
11     {
12         protected void Application_Start()
13         {
14             GlobalConfiguration.Configure(WebApiConfig.Register);
15         }
16     }
17 }
```

`WebApiConfig` is defined in *App_Start*, and has just one static `Register` method:

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Web.Http;
5
6 namespace ProductsApp
7 {
8     public static class WebApiConfig
9     {
10         public static void Register(HttpConfiguration config)
11         {
12             // Web API configuration and services
13
14             // Web API routes
15             config.MapHttpAttributeRoutes();
16         }
17     }
18 }
```

```

16         config.Routes.MapHttpRoute(
17             name: "DefaultApi",
18             routeTemplate: "api/{controller}/{id}",
19             defaults: new { id = RouteParameter.Optional }
20         );
21     };
22 }
23 }
24 }

```

This class configures [attribute routing](#), although it's not actually being used in the project. It also configures the routing table which is used by Web API 2. In this case, Web API will expect URLs to match the format `/api/{controller}/{id}`, with `{id}` being optional.

The *ProductsApp* project includes just one simple controller, which inherits from *ApiController* and exposes two methods:

```

1  using ProductsApp.Models;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Net;
6  using System.Web.Http;
7
8  namespace ProductsApp.Controllers
9  {
10     public class ProductsController : ApiController
11     {
12         Product[] products = new Product[]
13         {
14             new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
15             new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
16             new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
17         };
18
19         public IEnumerable<Product> GetAllProducts()
20         {
21             return products;
22         }
23
24         public IHttpActionResult GetProduct(int id)
25         {
26             var product = products.FirstOrDefault((p) => p.Id == id);
27             if (product == null)
28             {
29                 return NotFound();
30             }
31             return Ok(product);
32         }
33     }
34 }

```

Finally, the model, *Product*, used by the *ProductsApp*, is a simple class:

```

1  namespace ProductsApp.Models
2  {
3      public class Product
4      {

```

```

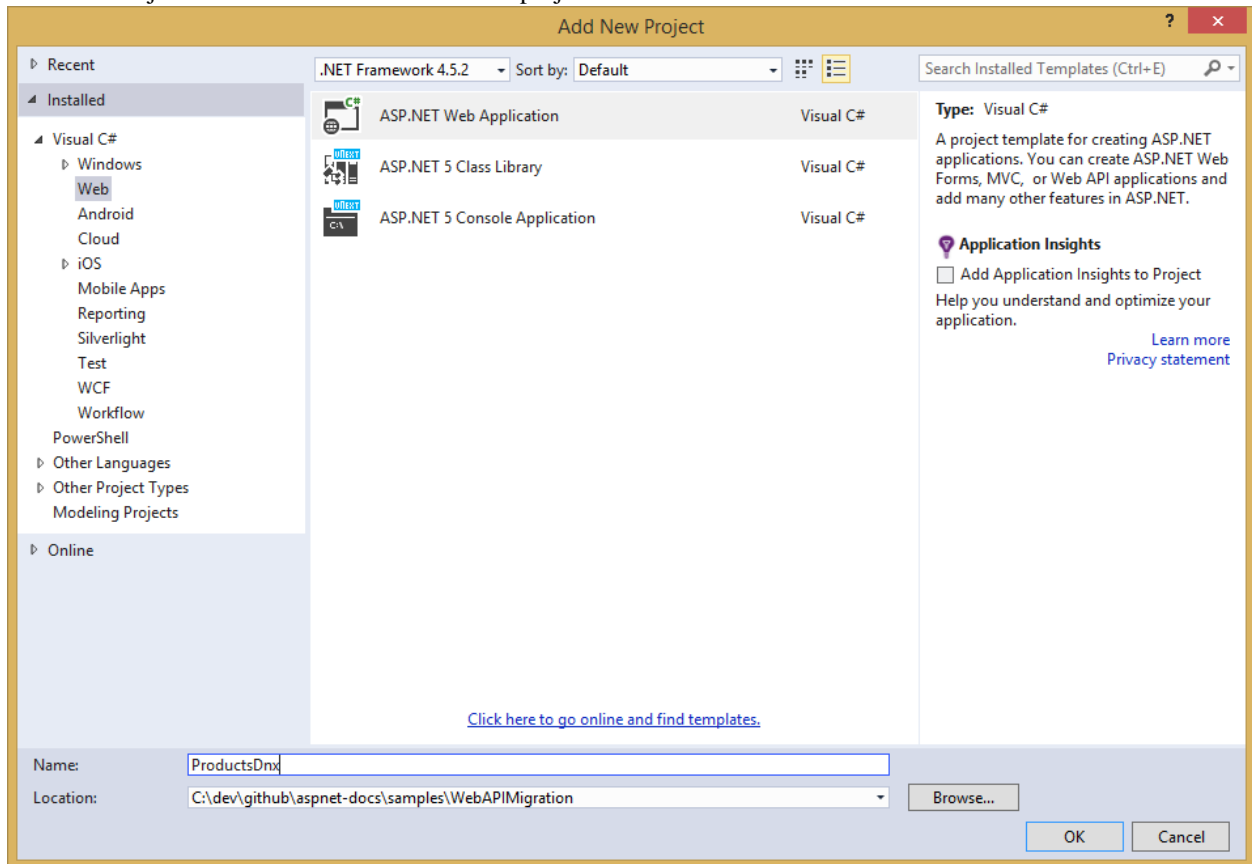
5     public int Id { get; set; }
6     public string Name { get; set; }
7     public string Category { get; set; }
8     public decimal Price { get; set; }
9 }
10

```

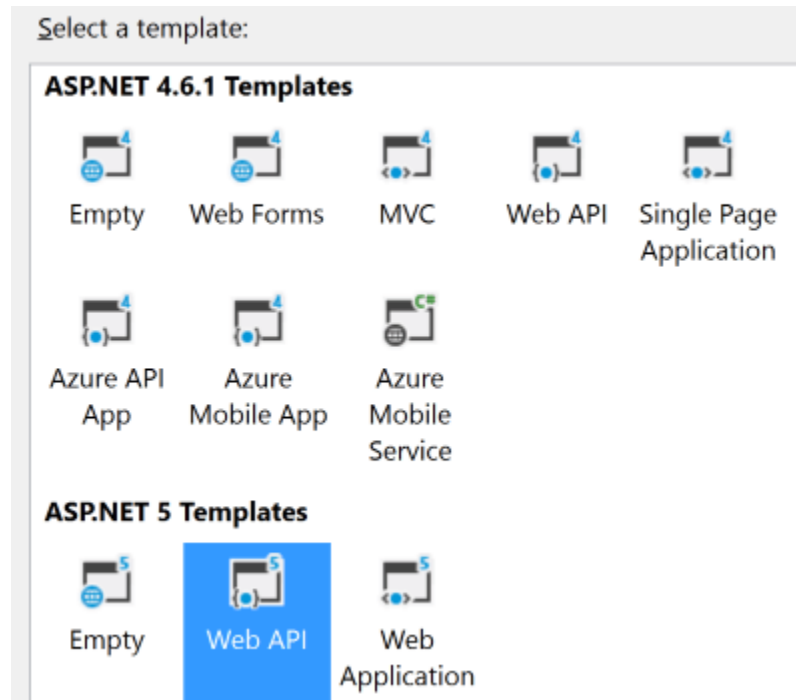
Now that we have a simple project from which to start, we can demonstrate how to migrate this Web API 2 project to ASP.NET MVC 6.

9.3.2 Create the Destination Project

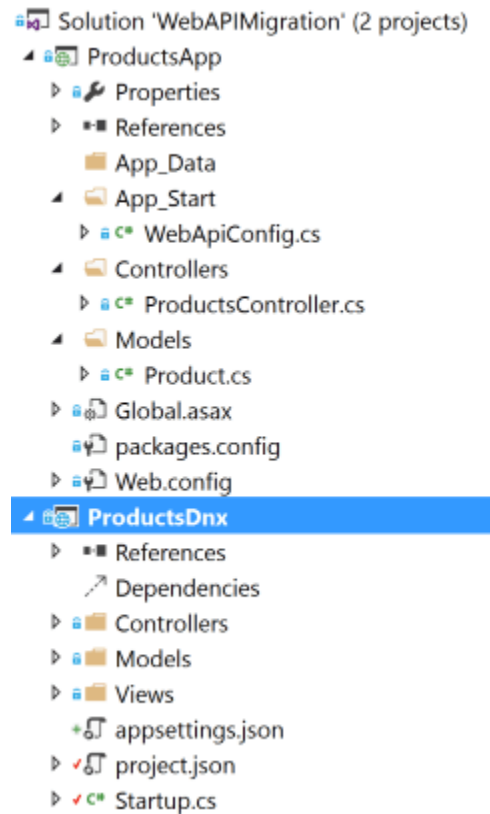
Using Visual Studio 2015, create a new, empty solution, and add the existing *ProductsApp* project to it. Then, add a new Web Project to the solution. Name the new project 'ProductsDnx'.



Next, choose the ASP.NET 5 Web API project template. We will migrate the *ProductsApp* contents to this new project.



Delete the `Project_Readme.html` file from the new project. Your solution should now look like this:



9.3.3 Migrate Configuration

ASP.NET 5 no longer uses *Global.asax*, *web.config*, or *App_Start* folders. Instead, all startup tasks are done in *Startup.cs* in the root of the project, and static configuration files can be wired up from there if needed (learn more about [ASP.NET 5 Application Startup](#)). Since Web API is now built into MVC 6, there is less need to configure it. Attribute-based routing is now included by default when `UseMvc()` is called, and this is the recommended approach for configuring Web API routes (and is how the Web API starter project handles routing).

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNet.Builder;
6  using Microsoft.AspNet.Hosting;
7  using Microsoft.Extensions.Configuration;
8  using Microsoft.Extensions.DependencyInjection;
9  using Microsoft.Extensions.Logging;
10
11 namespace ProductsDnx
12 {
13     public class Startup
14     {
15         public Startup(IHostingEnvironment env)
16         {
17             // Set up configuration sources.
18             var builder = new ConfigurationBuilder()
19                 .AddJsonFile("appsettings.json")
20                 .AddEnvironmentVariables();
21             Configuration = builder.Build();
22         }
23
24         public IConfigurationRoot Configuration { get; set; }
25
26         // This method gets called by the runtime. Use this method to add services to the container.
27         public void ConfigureServices(IServiceCollection services)
28         {
29             // Add framework services.
30             services.AddMvc();
31         }
32
33         // This method gets called by the runtime. Use this method to configure the HTTP request pipe
34         public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
35         {
36             loggerFactory.AddConsole(Configuration.GetSection("Logging"));
37             loggerFactory.AddDebug();
38
39             app.UseIISPlatformHandler();
40
41             app.UseStaticFiles();
42
43             app.UseMvc();
44         }
45
46         // Entry point for the application.
47         public static void Main(string[] args) => WebApplication.Run<Startup>(args);
48     }
49 }

```

Assuming you want to use attribute routing in your project going forward, no additional configuration is needed. Simply apply the attributes as needed to your controllers and actions, as is done in the sample `ValuesController` class that is included in the Web API starter project:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using Microsoft.AspNet.Mvc;
5
6 namespace ProductsDnx.Controllers
7 {
8     [Route("api/[controller]")]
9     public class ValuesController : Controller
10     {
11         // GET: api/values
12         [HttpGet]
13         public IEnumerable<string> Get()
14         {
15             return new string[] { "value1", "value2" };
16         }
17
18         // GET api/values/5
19         [HttpGet("{id}")]
20         public string Get(int id)
21         {
22             return "value";
23         }
24
25         // POST api/values
26         [HttpPost]
27         public void Post([FromBody] string value)
28         {
29         }
30
31         // PUT api/values/5
32         [HttpPut("{id}")]
33         public void Put(int id, [FromBody] string value)
34         {
35         }
36
37         // DELETE api/values/5
38         [HttpDelete("{id}")]
39         public void Delete(int id)
40         {
41         }
42     }
43 }
```

Note the presence of `[controller]` on line 8. Attribute-based routing now supports certain tokens, such as `[controller]` and `[action]`. These tokens are replaced at runtime with the name of the controller or action, respectively, to which the attribute has been applied. This serves to reduce the number of magic strings in the project, and it ensures the routes will be kept synchronized with their corresponding controllers and actions when automatic rename refactorings are applied.

To migrate the Products API controller, we must first copy *ProductsController* to the new project. Then simply include the route attribute on the controller:

```
[Route("api/[controller]")]
```

You also need to add the `[HttpGet]` attribute to the two methods, since they both should be called via HTTP Get. Include the expectation of an “id” parameter in the attribute for `GetProduct()`:

```
// /api/products
[HttpGet]
...

// /api/products/1
[HttpGet("{id}")]
```

At this point, routing is configured correctly; however, we can’t yet test it. Additional changes must be made before *ProductsController* will compile.

9.3.4 Migrate Models and Controllers

The last step in the migration process for this simple Web API project is to copy over the Controllers and any Models they use. In this case, simply copy *Controllers/ProductsController.cs* from the original project to the new one. Then, copy the entire Models folder from the original project to the new one. Adjust the namespaces to match the new project name (*ProductsDnx*). At this point, you can build the application, and you will find a number of compilation errors. These should generally fall into the following categories:

- *ApiController* does not exist
- *System.Web.Http* namespace does not exist
- *IHttpActionResult* does not exist
- *NotFound* does not exist
- *Ok* does not exist

Fortunately, these are all very easy to correct:

- Change *ApiController* to *Controller* (you may need to add *using Microsoft.AspNet.Mvc*)
- Delete any using statement referring to *System.Web.Http*
- Change any method returning *IHttpActionResult* to return a *ActionResult*
- Change *NotFound* to *HttpNotFound*
- Change *Ok(product)* to *new ObjectResult(product)*

Once these changes have been made and unused using statements removed, the migrated *ProductsController* class looks like this:

```
1 using Microsoft.AspNet.Mvc;
2 using ProductsDnx.Models;
3 using System.Collections.Generic;
4 using System.Linq;
5
6 namespace ProductsDnx.Controllers
7 {
8     [Route("api/[controller]")]
9     public class ProductsController : Controller
10     {
11         Product[] products = new Product[]
```

```
12     {
13         new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
14         new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
15         new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
16     };
17
18     // /api/products
19     [HttpGet]
20     public IEnumerable<Product> GetAllProducts()
21     {
22         return products;
23     }
24
25     // /api/products/1
26     [HttpGet("{id}")]
27     public IActionResult GetProduct(int id)
28     {
29         var product = products.FirstOrDefault((p) => p.Id == id);
30         if (product == null)
31         {
32             return HttpNotFound();
33         }
34         return new ObjectResult(product);
35     }
36 }
37 }
```

You should now be able to run the migrated project and browse to `/api/products`; and, you should see the full list of 3 products. Browse to `/api/products/1` and you should see the first product.

9.3.5 Summary

Migrating a simple Web API 2 project to MVC 6 is fairly straightforward, thanks to the merging of Web API into MVC 6 within ASP.NET 5. The main pieces every Web API 2 project will need to migrate are routes, controllers, and models, along with updates to the types used by MVC 6 controllers and actions.

9.3.6 Related Resources

[Create a Web API in MVC 6](#)

9.4 Migrating Authentication and Identity From ASP.NET MVC 5 to MVC 6

By [Steve Smith](#)

In the previous article we [migrated configuration from an ASP.NET MVC 5 project to MVC 6](#). In this article, we migrate the registration, login, and user management features.

This article covers the following topics:

- Configure Identity and Membership
- Migrate Registration and Login Logic
- Migrate User Management Features

You can download the finished source from the project created in this article [HERE \(TODO\)](#).

9.4.1 Configure Identity and Membership

In ASP.NET MVC 5, authentication and identity features are configured in `Startup.Auth.cs` and `IdentityConfig.cs`, located in the `App_Start` folder. In MVC 6, these features are configured in `Startup.cs`. Before pulling in the required services and configuring them, we should add the required dependencies to the project. Open `project.json` and add “Microsoft.AspNet.Identity.EntityFramework” and “Microsoft.AspNet.Identity.Cookies” to the list of dependencies:

```
"dependencies": {
  "Microsoft.AspNet.Server.IIS": "1.0.0-beta3",
  "Microsoft.AspNet.Mvc": "6.0.0-beta3",
  "Microsoft.Framework.ConfigurationModel.Json": "1.0.0-beta3",
  "Microsoft.AspNet.Identity.EntityFramework": "3.0.0-beta3",
  "Microsoft.AspNet.Security.Cookies": "1.0.0-beta3"
},
```

Now, open `Startup.cs` and update the `ConfigureServices()` method to use Entity Framework and Identity services:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddEntityFramework(Configuration)
        .AddSqlServer()
        .AddDbContext<ApplicationDbContext>();

    // Add Identity services to the services container.
    services.AddIdentity<ApplicationUser, IdentityRole>(Configuration)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven’t yet migrated from the MVC 5 project: `ApplicationDbContext` and `ApplicationUser`. Create a new `Models` folder in the MVC 6 project, and add two classes to it corresponding to these types. You will find the MVC 5 versions of these classes in `/Models/IdentityModels.cs`, but we will use one file per class in the migrated project since that’s more clear.

`ApplicationUser.cs`:

```
using Microsoft.AspNet.Identity;

namespace NewMvc6Project.Models
{
    public class ApplicationUser : IdentityUser
    {
    }
}
```

`ApplicationDbContext.cs`:

```
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Data.Entity;

namespace NewMvc6Project.Models
{
}
```

```

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    private static bool _created = false;
    public ApplicationDbContext()
    {
        // Create the database and schema if it doesn't exist
        // This is a temporary workaround to create database until Entity Framework 6
        // are supported in ASP.NET 5
        if (!_created)
        {
            Database.AsMigrationsEnabled().ApplyMigrations();
            _created = true;
        }
    }

    protected override void OnConfiguring(DbContextOptions options)
    {
        options.UseSqlServer();
    }
}

```

The MVC 5 Starter Web project doesn't include much customization of users, or the `ApplicationDbContext`. When migrating a real application, you will also need to migrate all of the custom properties and methods of your application's user and `DbContext` classes, as well as any other Model classes your application utilizes (for example, if your `DbContext` has a `DbSet<Album>`, you will of course need to migrate the `Album` class).

With these files in place, the `Startup.cs` file can be made to compile by updating its using statements:

```

using Microsoft.Framework.ConfigurationModel;
using Microsoft.AspNet.Hosting;
using NewMvc6Project.Models;
using Microsoft.AspNet.Identity;

```

Our application is now ready to support authentication and identity services - it just needs to have these features exposed to users.

9.4.2 Migrate Registration and Login Logic

With identity services configured for the application and data access configured using Entity Framework and SQL Server, we are now ready to add support for registration and login to the application. Recall that *earlier in the migration process* we commented out a reference to `_LoginPartial` in `_Layout.cshtml`. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Update `_Layout.cshtml`; uncomment the `@Html.Partial` line:

```

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>
</ul>
@*@Html.Partial("_LoginPartial")*@
</div>
</div>

```

Now, add a new MVC View Page called `_LoginPartial` to the `Views/Shared` folder:

Update `_LoginPartial.cshtml` with the following code (replace all of its contents):

```

@using System.Security.Principal

@if (User.Identity.IsAuthenticated)
{
    using (Html.BeginForm("LogOff", "Account", FormMethod.Post, new { id = "logoutForm", @class = "na
    {
        @Html.AntiForgeryToken()
        <ul class="nav navbar-nav navbar-right">
            <li>
                @Html.ActionLink("Hello " + User.Identity.GetUserName() + "!", "Manage", "Account", r
            </li>
            <li><a href="javascript:document.getElementById('logoutForm').submit()">Log off</a></li>
        </ul>
    }
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink("Register", "Register", "Account", routeValues: null, htmlAttributes: ne
        <li>@Html.ActionLink("Log in", "Login", "Account", routeValues: null, htmlAttributes: new {
    </ul>
}

```

At this point, you should be able to refresh the site in your browser.

9.4.3 Summary

ASP.NET 5 and MVC 6 introduce changes to the ASP.NET Identity 2 features that shipped with ASP.NET MVC 5. In this article, you have seen how to migrate the authentication and user management features of an ASP.NET MVC 5 project to MVC 6.

Contribute

The documentation on this site is the handiwork of our many [contributors](#).

We accept pull requests! But you're more likely to have yours accepted if you follow these guidelines:

1. Read <https://github.com/aspnet/Docs/blob/master/CONTRIBUTING.md>
2. Follow the [ASP.NET Docs Style Guide](#)