ASP.NET MVC 6 Documentation

Release 0.0.1

Microsoft

November 18, 2015

Contents

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

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.
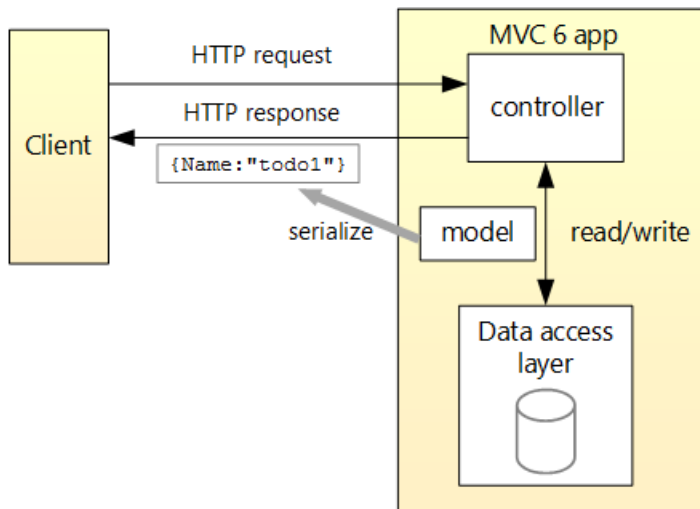
- The client is whatever consumes the web API (browser, mobile app, and so forth). We aren't writing a client in this tutorial.

- 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 Get Started with Entity Framework 7 using ASP.NET MVC 6.

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**.
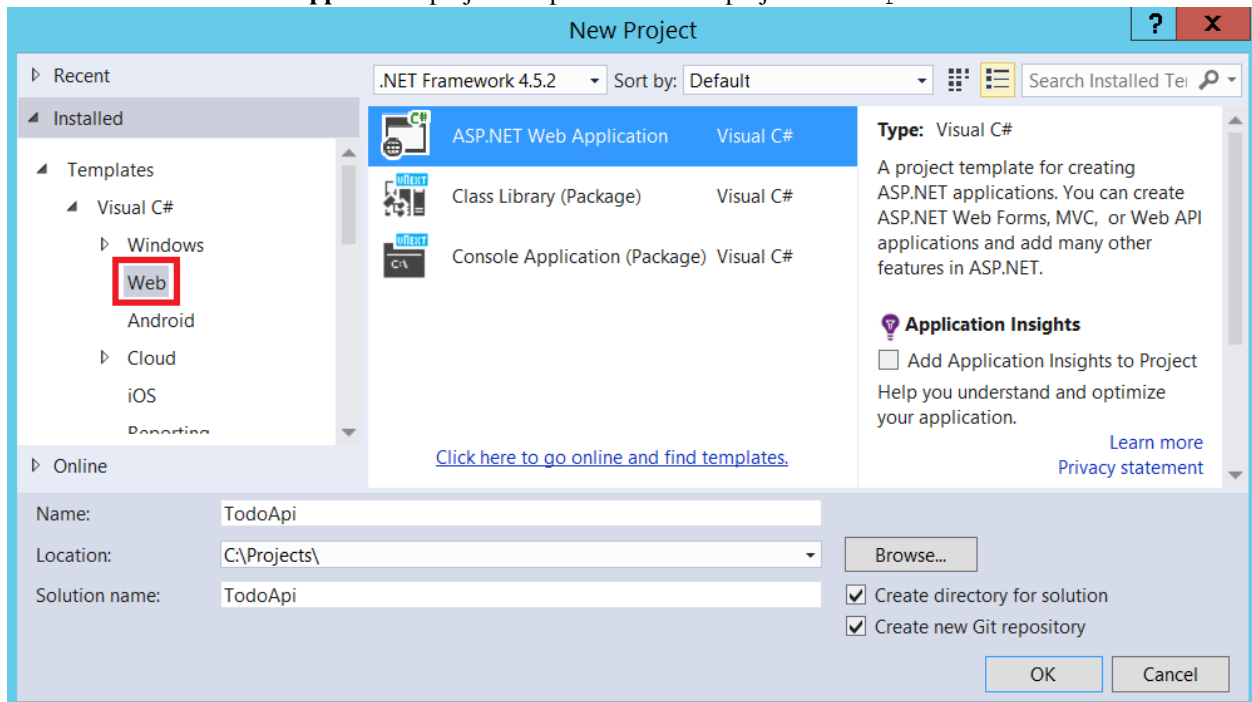
In the **New Project** dialog, select **Web API** under **ASP.NET 5 Preview Templates**. Click **OK**.
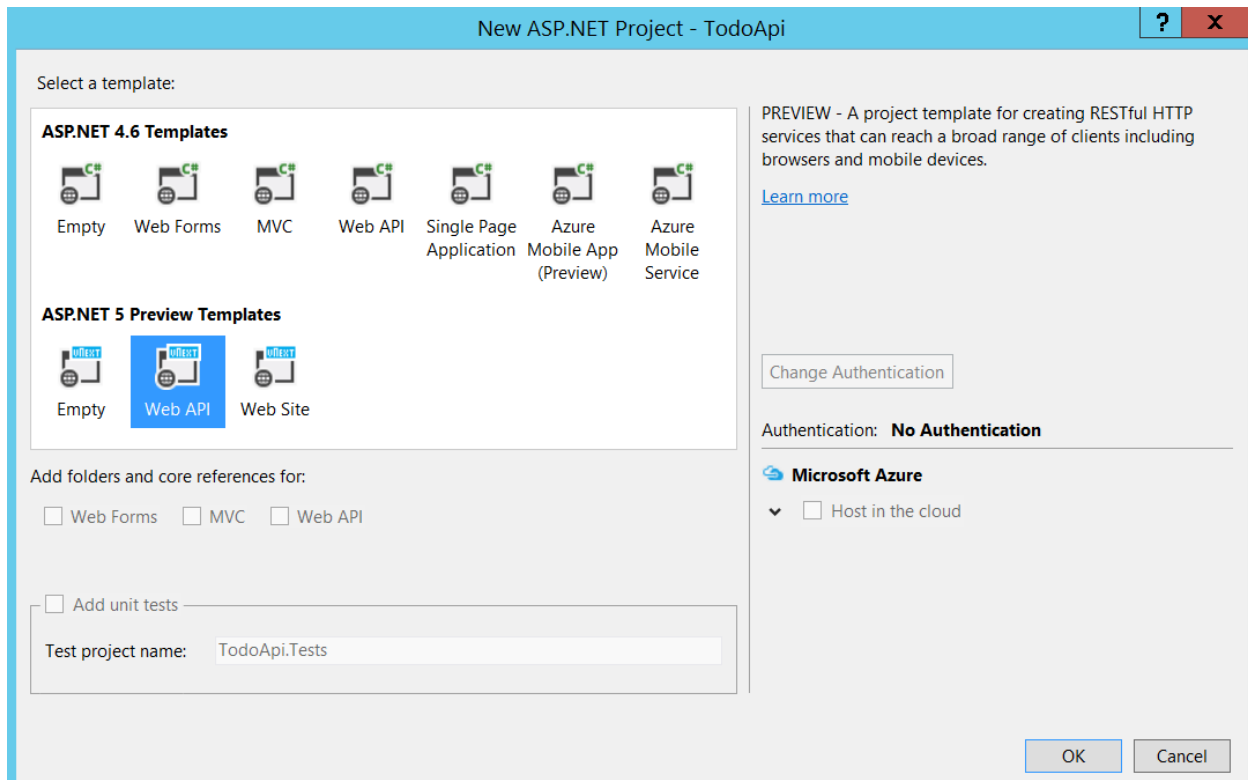
2.2.4 Add a model class

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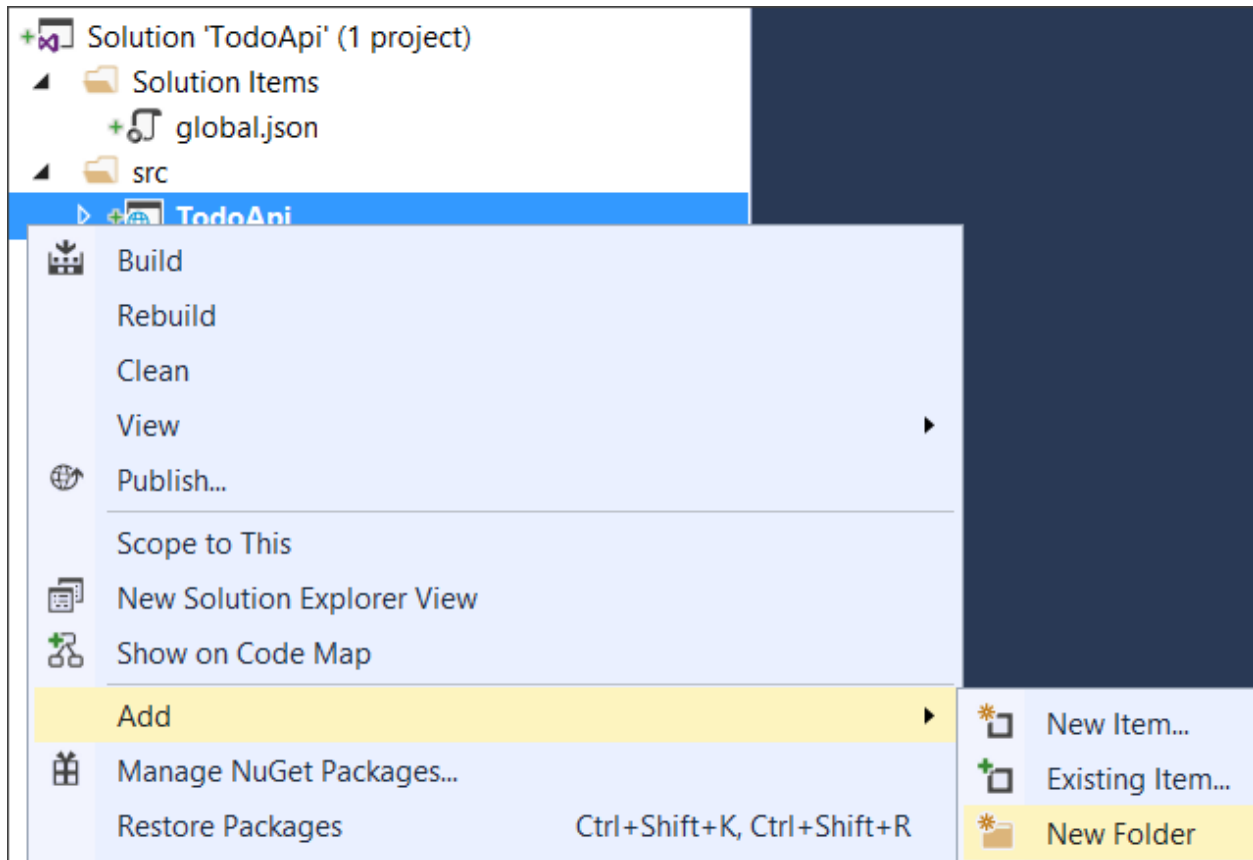
Note: You can put model classes anywhere in your project, but the *Models* folder is used by convention.

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**.
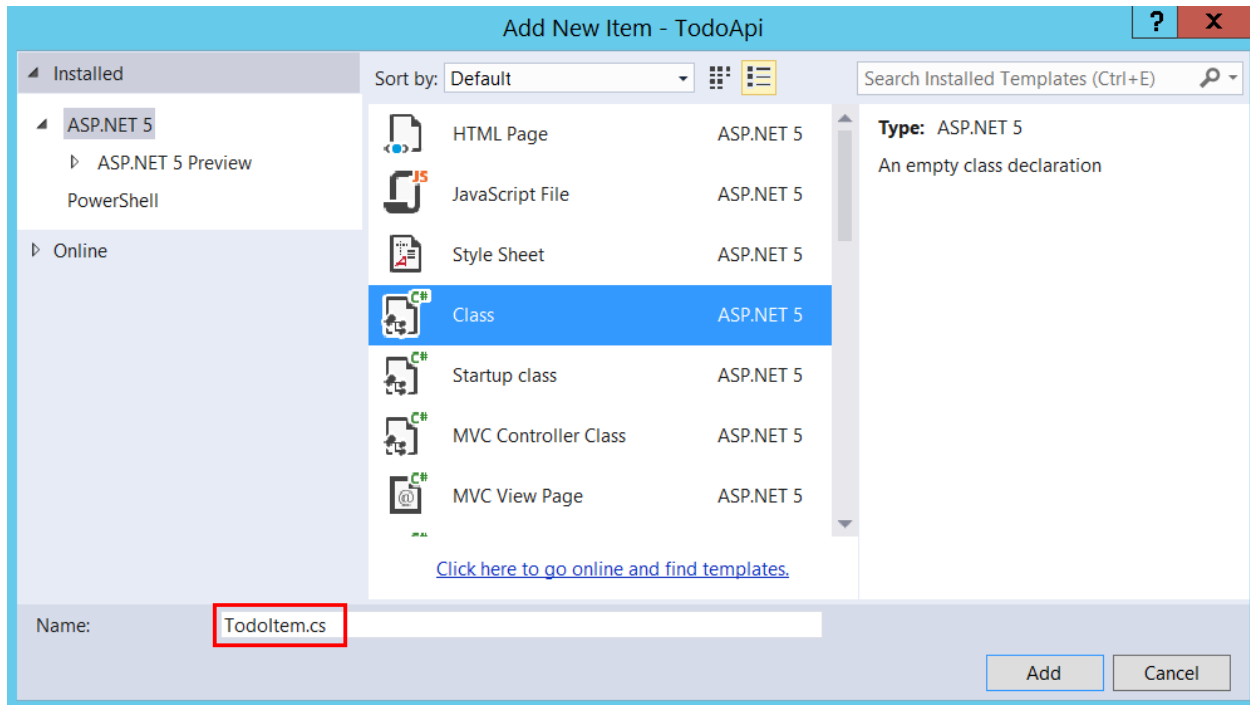
Replace the generated code with:

```
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`:

```csharp
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, TodoI

        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:

```csharp
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:

```csharp
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.

```csharp
[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 **Todo**Controller 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.

```json
{
  "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 `IActionResult` 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.
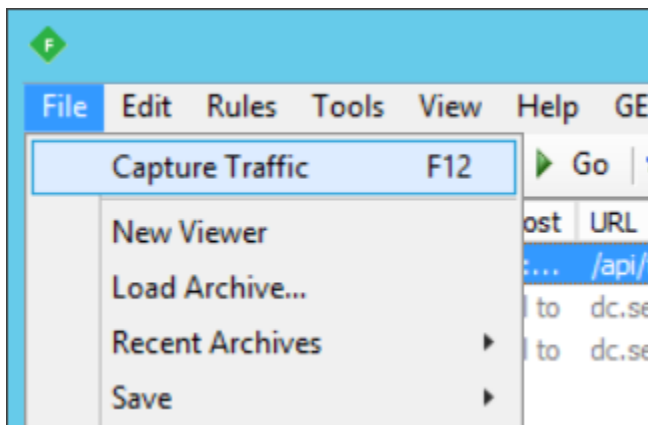
Select the **Composer** page. In the **Parsed** tab, type `http://localhost:port/api/todo`, where *port* is the port number. Click **Execute** to send the request.
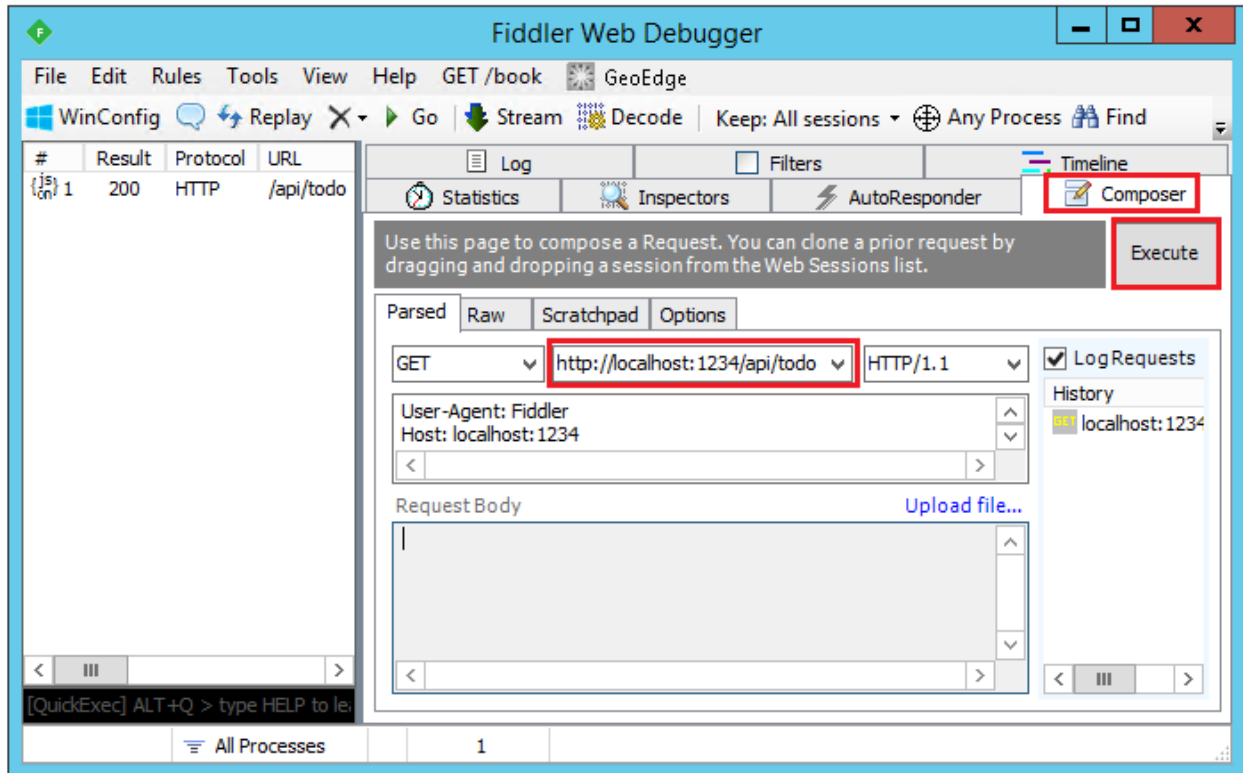
The result appears in the sessions list. The response code should be 200. Use the **Inspectors** tab to view the content of the response, including the response body.
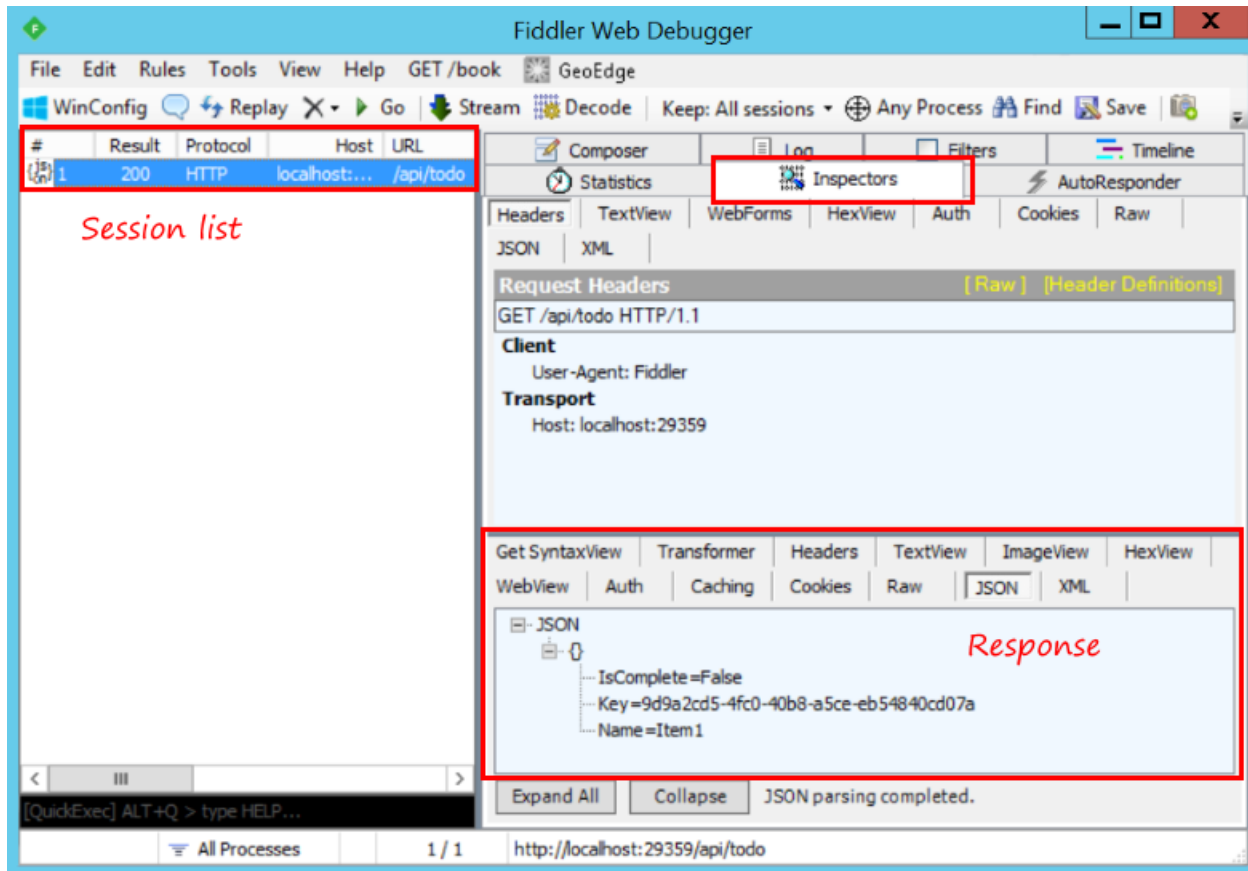
2.2.10 Implement the other CRUD operations

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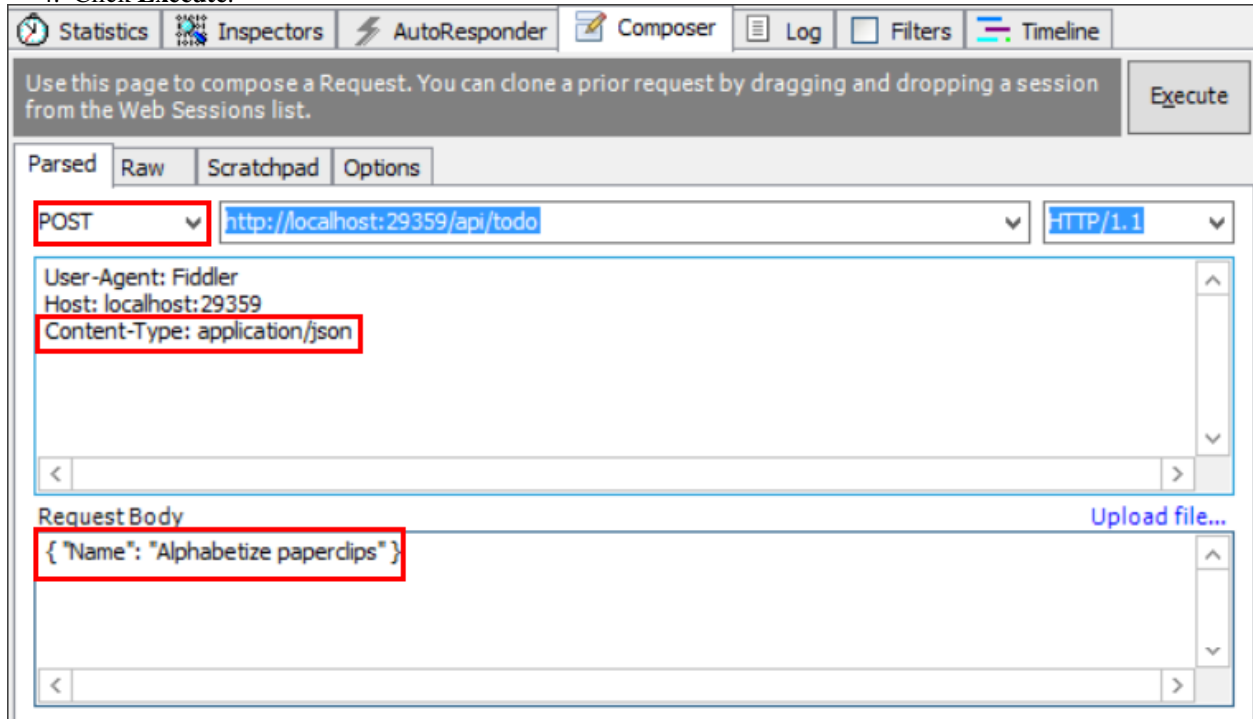
Here is an example HTTP session. Use the **Raw** tab to see the session data in this format.

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 HttpBadRequest();
    }

    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.

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.
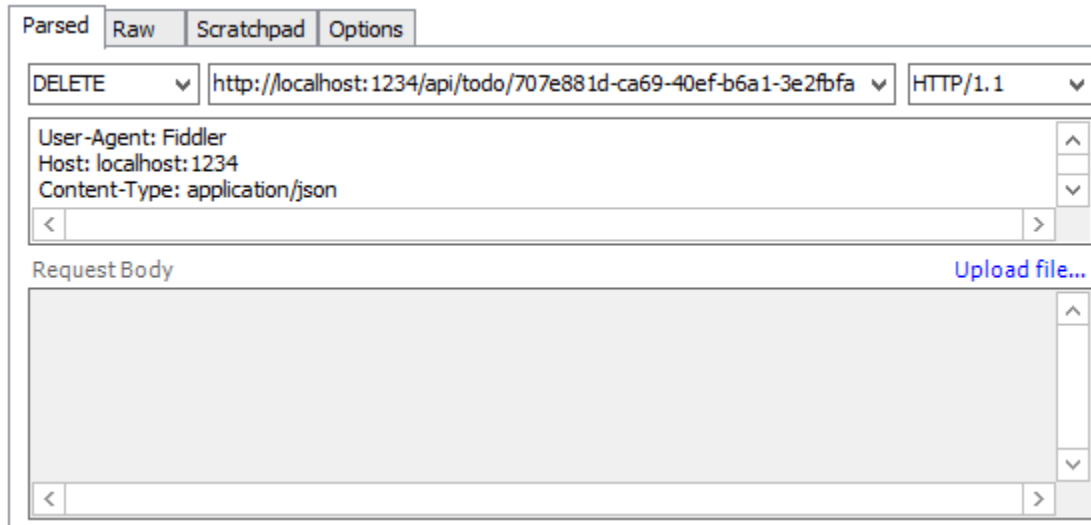
2.2.11 Next steps

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.

Tutorials

3.1 Get Started with Entity Framework 7 using ASP.NET MVC 6

By Mike Wasson and Rick Anderson

In this tutorial, you'll create a simple web app using ASP.NET MVC and Entity Framework (EF). The app stores records in a SQL database and supports the basic CRUD operations (create, read, update, delete).

Note: This tutorial uses Visual Studio 2015. If you are completely new to ASP.NET MVC or Visual Studio, read Building Your First MVC 6 Application first.

The sample app that you'll build manages a list of authors and books. Here is a screen shot of the app:
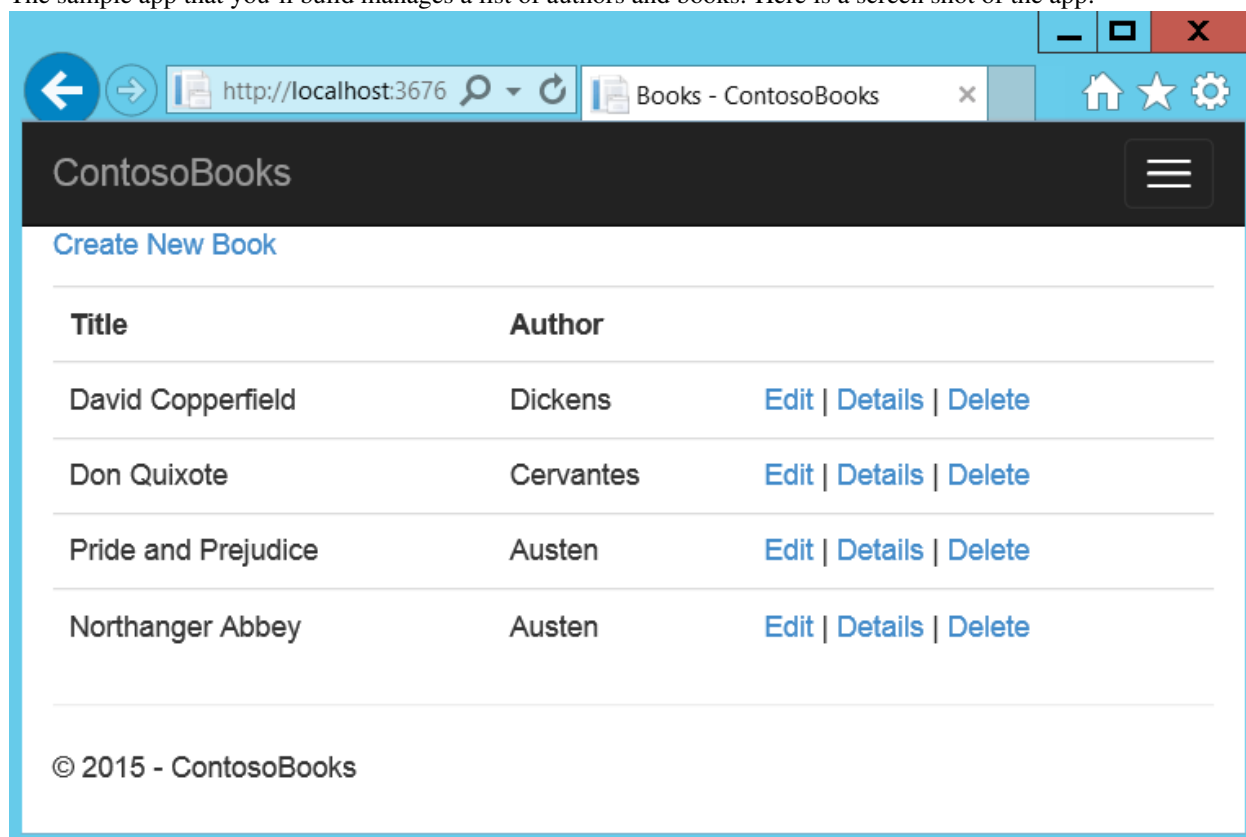
The app uses Razor to generate static HTML. (An alternate approach is to update pages dynamically on the client,

with a combination of AJAX calls, JSON data, and client-side JavaScript. This tutorial doesn't cover that approach.)

In this article:

- *Create the project*
- *Add Entity Framework*
- *Create entity classes*
- *Add a DbContext class*
- *Configure Entity Framework*
- *Add an index page*
- *Add a details page*
- *Add a create page*
- *Add an edit page*
- *Add a delete page*
- *Wrapping up*

You can browse the source code for the sample app on GitHub.

3.1.1 Create the project

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

Select the **ASP.NET Web Application** project template. It appears under **Installed > Templates > Visual C# > Web**. Name the project ContosoBooks and click **OK**.
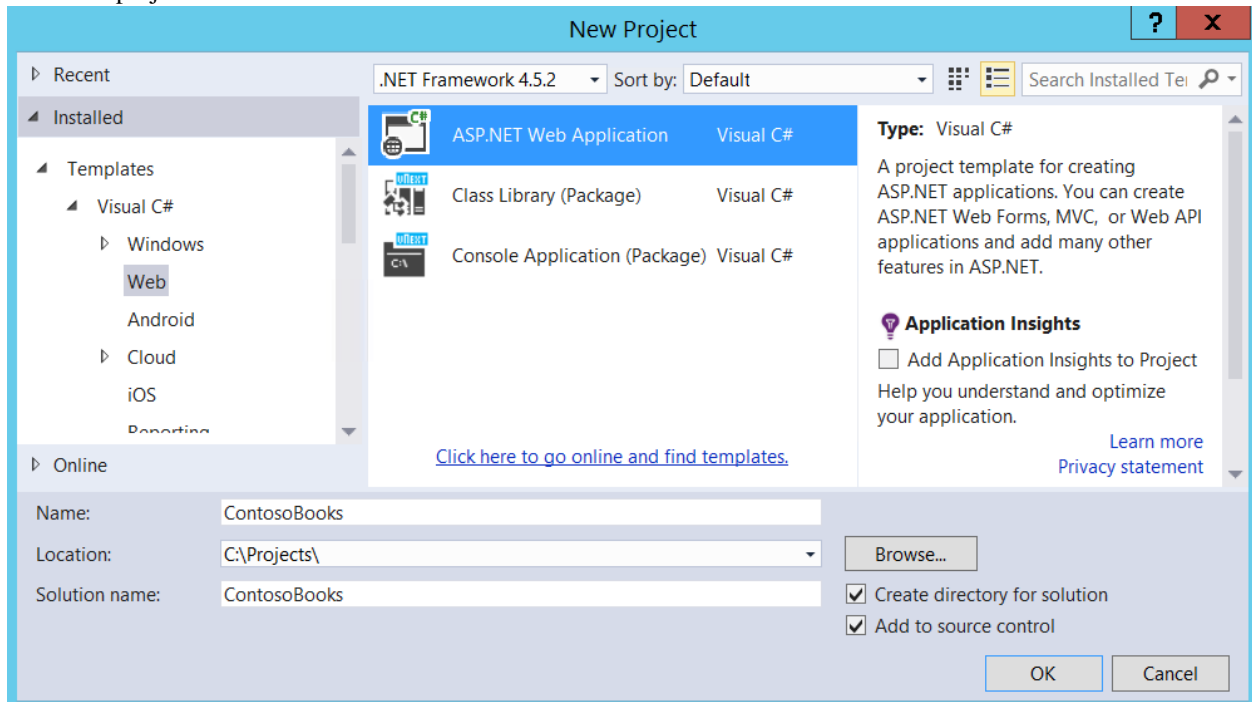
In the **New Project** dialog, select **Web Site** under **ASP.NET 5 Preview Templates**.
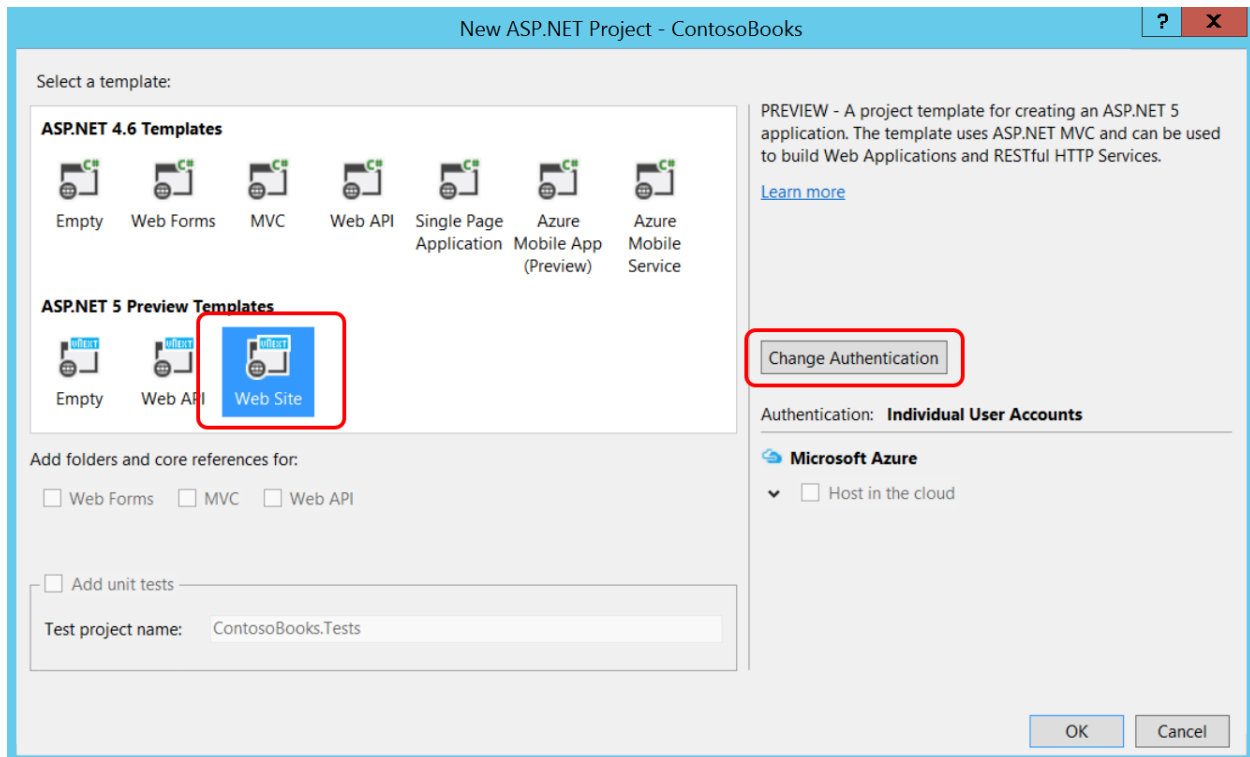
Click **Change Authentication** and select **No Authentication**. You won't need authentication for this sample. Click **OK** twice to complete the dialogs and create the project.
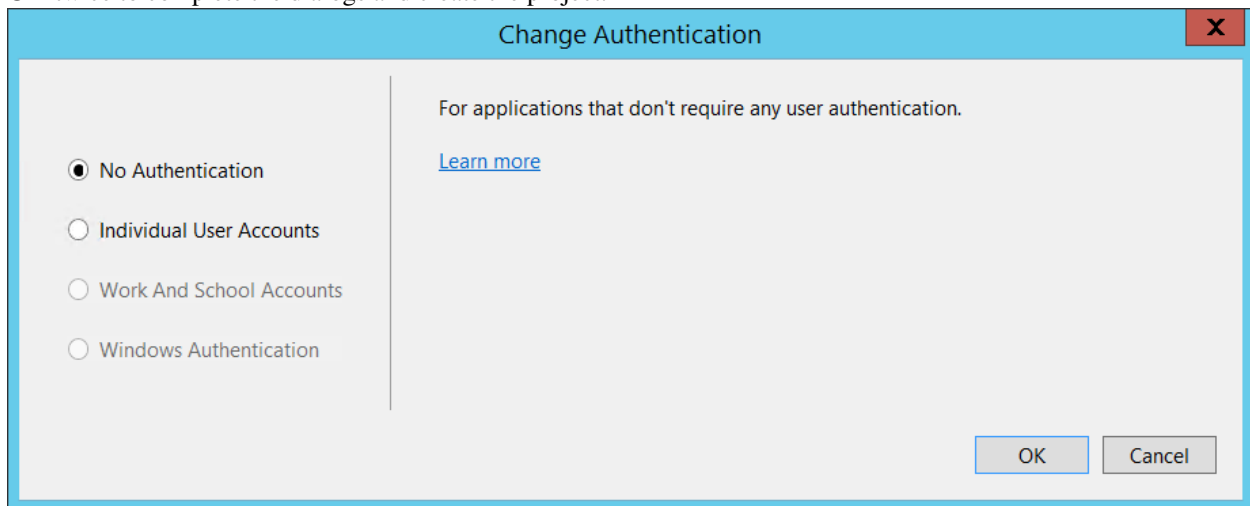
Open the *Views/Shared/_Layout.cshtml* file. Replace the following code:

```
<li><a asp-controller="Home" asp-action="Index">Home</a></li>
<li><a asp-controller="Home" asp-action="About">About</a></li>
<li><a asp-controller="Home" asp-action="Contact">Contact</a></li>
```

with this:

```
<li><a asp-controller="Book" asp-action="Index">Books</a></li>
<li><a asp-controller="Author" asp-action="Index">Authors</a></li>
```

This adds a link to the Books page, which we haven't created yet. (That will come later in tutorial.)

3.1.2 Add Entity Framework

Open the *project.json* file. In the dependencies section, add the following line:

```
"dependencies": {
  ...
  "EntityFramework.SqlServer": "7.0.0-beta8"
},
```

When you save *project.json*, Visual Studio automatically resolves the new package reference.
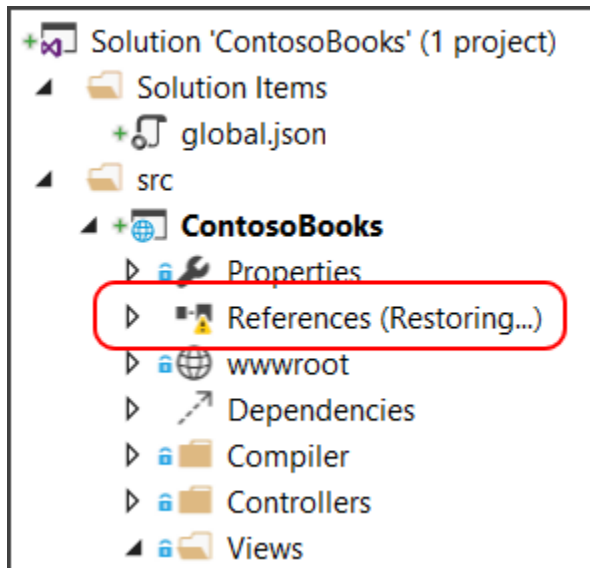
3.1.3 Create entity classes

The app will have two entities:

- Book

- Author

We'll define a class for each. First, add a new folder to the project. In Solution Explorer, right-click the project. (The project appears under the "src" folder.) Select **Add > New Folder**. Name the folder *Models*.

```
Solution Explorer                      ▼ ┼ ✕
  ⊙  ⊙  ⌂ | ⊙ ▼ ⇆ ⊡ | 🔧 ▬ ⅋

  Search Solution Explorer (Ctrl+;)         🔎 ▼

  +⊠ Solution 'ContosoBooks' (1 project)
  ◢  📁 Solution Items
      +⌂ global.json
  ◢  📁 src
    ◢ +🌐 ContosoBooks
```

	Build		
🏭	Build		
	Rebuild		
	Clean		
	View	▶	
🌐	Publish...		
	Scope to This		
🗐	New Solution Explorer View		
⅋	Show on Code Map		
	Add	▶	✳🗂 New Item...
🎁	Manage NuGet Packages...		➕🗂 Existing Item...
	Restore Packages	Ctrl+Shift+K, Ctrl+Shift+R	✳📁 New Folder
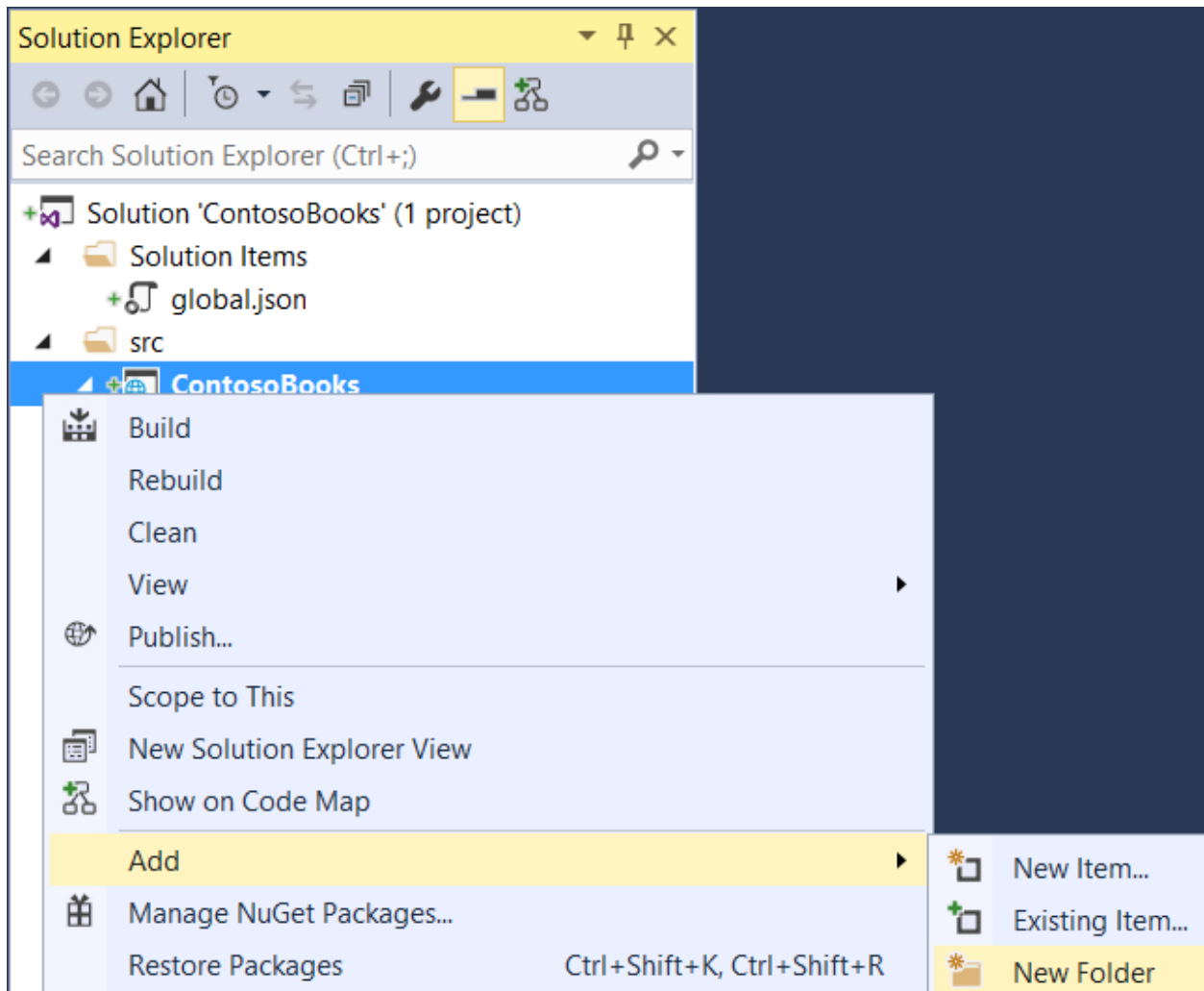
Note: You can put model classes anywhere in your project. The *Models* folder is just a convention.

Right-click the *Models* folder and select **Add > New Item**. In the **Add New Item** dialog, select the **Class** template. In the **Name** edit box, type "Author.cs" and click OK. Replace the boilerplate code with:

```csharp
using System.ComponentModel.DataAnnotations;

namespace ContosoBooks.Models
{
    public class Author
    {
        public int AuthorID { get; set; }

        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
    }
}
```

Repeat these steps to add another class named `Book` with the following code:

```
using System.ComponentModel.DataAnnotations;

namespace ContosoBooks.Models
{
    public class Book
    {
        public int BookID { get; set; }

        public string Title { get; set; }

        public int Year { get; set; }

        public decimal Price { get; set; }

        public string Genre { get; set; }

        public int AuthorID { get; set; }

        // Navigation property
        public Author Author { get; set; }
    }
}
```

To keep the app simple, each book has a single author. The `Author` property provides a way to navigate the relationship from a book to an author. In EF, this type of property is called a *navigation property*. When EF creates the DB schema, EF automatically infers that `AuthorID` should be a foreign key to the Authors table.

3.1.4 Add a DbContext class

In EF 7, the primary class for interacting with data is `Microsoft.Data.Entity.DbContext`. Add a class in the *Models* folder named `BookContext` that derives from `DbContext`:

```
using Microsoft.Data.Entity;

namespace ContosoBooks.Models
{
    public class BookContext : DbContext
    {
        public DbSet<Author> Authors { get; set; }
        public DbSet<Book> Books { get; set; }
    }
}
```

The `DbSet` properties represent collections of entities. These will become tables in the SQL database.

Next, we'll create some sample data. Add a class named `SampleData` in the *Models* folder with the following code:

```
using Microsoft.Data.Entity;
using Microsoft.Framework.DependencyInjection;
using System;
using System.Linq;
using Microsoft.Data.Entity.Storage;

namespace ContosoBooks.Models
{
    public static class SampleData
    {
        public static void Initialize(IServiceProvider serviceProvider)
```

```csharp
        {
            var context = serviceProvider.GetService<BookContext>();

            if(serviceProvider.GetService<IRelationalDatabaseCreator>().Exists())
            {
                if (!context.Books.Any())
                {
                    var austen = context.Authors.Add(
                        new Author { LastName = "Austen", FirstMidName = "Jane" }).Entity;
                    var dickens = context.Authors.Add(
                        new Author { LastName = "Dickens", FirstMidName = "Charles" }).Entity;
                    var cervantes = context.Authors.Add(
                        new Author { LastName = "Cervantes", FirstMidName = "Miguel" }).Entity;

                    context.Books.AddRange(
                        new Book()
                        {
                            Title = "Pride and Prejudice",
                            Year = 1813,
                            Author = austen,
                            Price = 9.99M,
                            Genre = "Comedy of manners"
                        },
                        new Book()
                        {
                            Title = "Northanger Abbey",
                            Year = 1817,
                            Author = austen,
                            Price = 12.95M,
                            Genre = "Gothic parody"
                        },
                        new Book()
                        {
                            Title = "David Copperfield",
                            Year = 1850,
                            Author = dickens,
                            Price = 15,
                            Genre = "Bildungsroman"
                        },
                        new Book()
                        {
                            Title = "Don Quixote",
                            Year = 1617,
                            Author = cervantes,
                            Price = 8.95M,
                            Genre = "Picaresque"
                        }
                    );
                    context.SaveChanges();
                }
            }
        }
    }
}
```

You wouldn't put this into production code, but it's OK for a sample app.

3.1.5 Configure Entity Framework

Open *config.json*. Add the following highlighted lines:

```
{
  "AppSettings": {
    "SiteTitle": "Contoso Books"
  },
  "Data": {
    "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=ContosoBooks;Trusted_Connection=True
  }
}
```

This defines a connection string to LocalDB, which is a lightweight version of SQL Server Express for development.

Open the *Startup.cs* file. In the `ConfigureServices` method, add:

```
services.AddEntityFramework()
    .AddSqlServer()
    .AddDbContext<BookContext>(options =>
    {
        options.UseSqlServer(Configuration["Data:ConnectionString"]);
    });
```

Add the following code at the end of the *Configure* method:

```
SampleData.Initialize(app.ApplicationServices);
```

Notice in *ConfigureServices* that we call `Configuration.Get` to get the database connection string. During development, this setting comes from the *config.json* file. When you deploy the app to a production environment, you set the connection string in an environment variable on the host. If the Configuration API finds an environment variable with the same key, it returns the environment variable instead of the value that is in *config.json*.

Here is the complete *Startup.cs* after these changes:

```
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Diagnostics;
using Microsoft.AspNet.Hosting;
using Microsoft.Data.Entity;
using Microsoft.Framework.Configuration;
using Microsoft.Framework.DependencyInjection;
using Microsoft.Framework.Logging;
using ContosoBooks.Models;
using Microsoft.Dnx.Runtime;

namespace ContosoBooks
{
    public class Startup
    {
        public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
        {

            // Setup configuration sources.
            var builder = new ConfigurationBuilder()
                .SetBasePath(appEnv.ApplicationBasePath)
                .AddJsonFile("config.json")
                .AddEnvironmentVariables();

            Configuration = builder.Build();
        }
```

```
        public IConfiguration Configuration { get; set; }

        // This method gets called by the runtime.
        public void ConfigureServices(IServiceCollection services)
        {
            // Add MVC services to the services container.
            services.AddMvc();

            services.AddEntityFramework()
                .AddSqlServer()
                .AddDbContext<BookContext>(options =>
                {
                    options.UseSqlServer(Configuration["Data:ConnectionString"]);
                });
        }

        // Configure is called after ConfigureServices is called.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory logger
        {
            loggerFactory.MinimumLevel = LogLevel.Information;
            loggerFactory.AddConsole();

            // Configure the HTTP request pipeline.

            // Add the following to the request pipeline only in development environment.
            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
            }
            else
            {
                // Add Error handling middleware which catches all application specific errors and
                // send the request to the following path or controller action.
                app.UseExceptionHandler("/Home/Error");
            }

            // Add static files to the request pipeline.
            app.UseStaticFiles();

            // Add MVC to the request pipeline.
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Book}/{action=Index}/{id?}");

                // Uncomment the following line to add a route for porting Web API 2 controllers.
                // routes.MapWebApiRoute("DefaultApi", "api/{controller}/{id?}");
            });
            SampleData.Initialize(app.ApplicationServices);
        }
    }
}
```

Use data migrations to create the database

Open *project.json.*   - In the "commands" and "dependencies" sections, add an entry for
`EntityFramework.Commands.`

```
{
  "webroot": "wwwroot",
  "version": "1.0.0-*",

  "dependencies": {
    "Microsoft.AspNet.Diagnostics": "1.0.0-beta8",
    "Microsoft.AspNet.Mvc": "6.0.0-beta8",
    "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-beta8",
    "Microsoft.AspNet.Server.WebListener": "1.0.0-beta8",
    "Microsoft.AspNet.StaticFiles": "1.0.0-beta8",
    "Microsoft.AspNet.Tooling.Razor": "1.0.0-beta8",
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-beta8",
    "Microsoft.Framework.Configuration.Json": "1.0.0-beta8",
    "Microsoft.Framework.Logging": "1.0.0-beta8",
    "Microsoft.Framework.Logging.Console": "1.0.0-beta8",
    "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0-beta8",
    "EntityFramework.SqlServer": "7.0.0-beta8",
    "EntityFramework.Commands": "7.0.0-beta8"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:5000",
    "ef": "EntityFramework.Commands"
  },

  "frameworks": {
    "dnx451": { },
    "dnxcore50": { }
  },

  "exclude": [
    "wwwroot",
    "node_modules",
    "bower_components"
  ],
  "publishExclude": [
    "node_modules",
    "bower_components",
    "**.xproj",
    "**.user",
    "**.vspscc"
  ],
  "scripts": {
    "prepublish": [ "npm install", "bower install", "gulp clean", "gulp min" ]
  }
}
```

Build the app.

Open a command prompt in the project directory (ContosoBooks/src/ContosoBooks) and run the following commands:

```
dnvm use 1.0.0-beta8
dnx ef . migration add Initial
dnx ef . migration apply
```

The "`add Initial`" command adds code to the project that allows EF to update the database schema. The "`apply`" command creates the actual database. After you run the run these commands, your project has a new folder named *Migrations*:
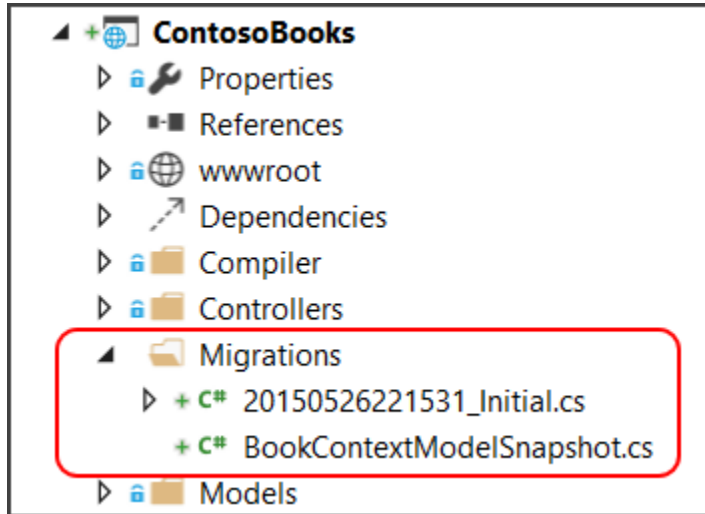
- **dnvm** : The .NET Version Manager, a set of command line utilities that are used to update and configure .NET Runtime. The command `dnvm use 1.0.0-beta8` instructs the .NET Version Manager to add the 1.0.0-beta8 ASP.NET 5 runtime to the `PATH` environment variable for the current shell. For ASP.NET 5 Beta 8, the following is displayed:

```
Adding C:\\Users\\<user>\\.dnx\\runtimes\\dnx-clr-win-x86.1.0.0-beta8\\bin to process PATH
```

- **dnx ef migration add Initial** : DNX is the .NET Execution Environment. The `ef migration apply` command runs pending migration code. For more information about `dnvm`, `dnu`, and `dnx`, see DNX Overview.

3.1.6 Add an index page

In this step, you'll add code to display a list of books.

Right-click the *Controllers* folder. Select **Add > New Item**. Select the **MVC Controller Class** template. Name the class `BookController`.
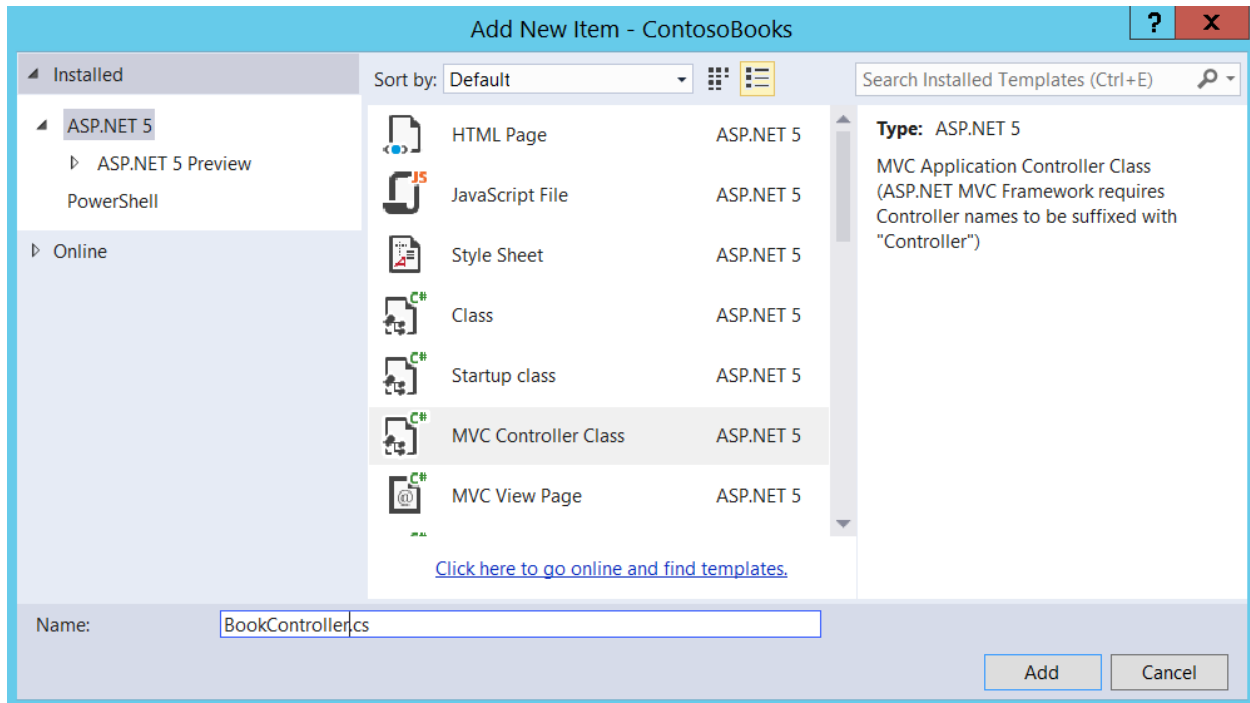
Replace the boilerplate code with the following:

```
using ContosoBooks.Models;
using Microsoft.AspNet.Mvc;
using Microsoft.AspNet.Mvc.Rendering;
using Microsoft.Data.Entity;
using Microsoft.Framework.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoBooks.Controllers
{
    public class BookController : Controller
    {
        [FromServices]
        public BookContext BookContext { get; set; }

        [FromServices]
        public ILogger<BookController> Logger { get; set; }

        public IActionResult Index()
        {
            var books = BookContext.Books.Include(b => b.Author);
            return View(books);
        }
    }
}
```

Notice that we don't set any value for `Logger` and `BookContext`. The dependency injection (DI) subsystem automatically sets these properties at runtime. DI also handles the object lifetimes, so you don't need to call `Dispose`. For more information, see Dependency Injection.

In the *Views* folder, make a sub-folder named *Book*. You can do this by right-clicking the *Views* folder in Solution

Explorer and clicking **Add New Folder**.

Right-click the *Views/Book* subfolder that you just created, and select **Add > New Item**. Select the **MVC View Page** template. Keep the default name, *Index.cshtml*.

Note: For views, the folder and file name are significant. The view defined in *Views/Book/Index.cshtml* corresponds to the action defined in the `BookController.Index` method.

Replace the boilerplate code with:

```
@model IEnumerable<ContosoBooks.Models.Book>
@{
    ViewBag.Title = "Books";
}
<p>
    <a asp-action="Create">Create New Book</a>
</p>

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Author)
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Author.LastName)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.BookID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.BookID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.BookID">Delete</a>
            </td>
        </tr>
    }
</table>
```

Run the app and click the "Books" link in the top nav bar. You should see a list of books. The links for create, edit, details, and delete are not functioning yet. We'll add those next.

3.1.7 Add a details page

Add the following method to the `BooksController` class:

```
public async Task<ActionResult> Details(int id)
{
    Book book = await BookContext.Books
        .Include(b => b.Author)
```

```
            .SingleOrDefaultAsync(b => b.BookID == id);
    if (book == null)
    {
        Logger.LogInformation("Details: Item not found {0}", id);
        return HttpNotFound();
    }
    return View(book);
}
```

This code looks up a book by ID. In the EF query:

- The `Include` method tells EF to fetch the related `Author` entity.

- The `SingleOrDefaultAsync` method returns a single entity, or `null` if one is not found.

If the EF query returns `null`, the controller method returns `HttpNotFound`, which ASP.NET translates into a 404 response. Otherwise, the controller passes *book* to a view, which renders the details page. Let's add the view now.

In the *Views/Book* folder, add a view named *Details.cshtml* with the following code:

```
@model ContosoBooks.Models.Book

@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>
<div>
  <dl class="dl-horizontal">
    <dt>@Html.DisplayNameFor(model => model.Title)</dt>
    <dd>@Html.DisplayFor(model => model.Title)</dd>

    <dt>@Html.DisplayNameFor(model => model.Author)</dt>
    <dd>
      @Html.DisplayFor(model => model.Author.FirstMidName)
      @Html.DisplayFor(model => model.Author.LastName)
    </dd>

    <dt>@Html.DisplayNameFor(model => model.Year)</dt>
    <dd>@Html.DisplayFor(model => model.Year)</dd>

    <dt>@Html.DisplayNameFor(model => model.Genre)</dt>
    <dd>@Html.DisplayFor(model => model.Genre)</dd>

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

3.1.8 Add a create page

Add the following two methods to `BookController`:

```csharp
public ActionResult Create()
{
    ViewBag.Items = GetAuthorsListItems();
    return View();
}

[HttpPost]
{
    var tmp = BookContext.Authors.ToList();  // Workaround for https://github.com/aspnet/EntityFrame

    // Create authors list for <select> dropdown
    return tmp
        .OrderBy(author => author.LastName)
        .Select(author => new SelectListItem
        {
            Text = String.Format("{0}, {1}", author.LastName, author.FirstMidName),
            Value = author.AuthorID.ToString(),
            Selected = author.AuthorID == selected
        });
}
```

Add a view named *Views/Book/Create.cshtml*.

```html
@model ContosoBooks.Models.Book

<div>
  <form asp-controller="Book" asp-action="Create" method="post">
    <div asp-validation-summary="ValidationSummary.ModelOnly" class="text-danger"></div>
    <div class="form-group">
      <label asp-for="Title"></label>
      <input asp-for="Title" class="form-control" placeholder="Title"/>
      <span asp-validation-for="Title" class="text-danger"></span>
    </div>
    <div class="form-group">
     <select asp-for="AuthorID" asp-items="@ViewBag.Items"></select>
    </div>
    <div class="form-group">
      <label asp-for="Year"></label>
      <input asp-for="Year" class="form-control" placeholder="1900"/>
      <span asp-validation-for="Year" class="text-danger"></span>
    </div>
    <div class="form-group">
      <label asp-for="Price"></label>
      <input asp-for="Price" class="form-control" placeholder="1.00"/>
      <span asp-validation-for="Price" class="text-danger"></span>
    </div>
    <div class="form-group">
      <label asp-for="Genre"></label>
      <input asp-for="Genre" class="form-control" placeholder="Genre"/>
      <span asp-validation-for="Genre" class="text-danger"></span>
    </div>
    <input type="submit" class="btn btn-default" value="Create" />
  </form>
</div>

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

```
}
```

This view renders an HTML form. In the `form` element, the `asp-action` tag helper specifies the controller action to invoke when the client submits the form. Notice that the form uses HTTP POST.

```html
<form asp-controller="Book" asp-action="Create" method="post">
```

Now let's write the controller action to handle the form post. In the `BookController` class, add the following method.

```csharp
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Create([Bind("Title", "Year", "Price", "Genre", "AuthorID")] Book boo
{
    try
    {
        if (ModelState.IsValid)
        {
            BookContext.Books.Add(book);
            await BookContext.SaveChangesAsync();
            return RedirectToAction("Index");
        }
    }
    catch (DbUpdateException)
    {
        ModelState.AddModelError(string.Empty, "Unable to save changes.");
    }
    return View(book);
}
```

The `[HttpPost]` attribute tells MVC that this action applies to HTTP POST requests. The `[ValidateAntiForgeryToken]` attribute is a security feature that guards against cross-site request forgery. For more information, see Anti-Request Forgery.

Inside this method, we check the model state (`ModelState.IsValid`). If the client submitted a valid model, we add it to the database. Otherwise, we return the original view with validation errors shown:

```html
<span asp-validation-for="Title" class="text-danger"></span>
```

Add validation rules to the book model

To see how validation works, let's add some validation rules to the `Book` model.

1. Open Book.cs.

2. Add the `[Required]` attribute to the `Title` property.

3. Add the `[Range]` property to the `Price` property, as shown below.

```csharp
public class Book
{
    public int BookID { get; set; }

    [Required]
    public string Title { get; set; }

    public int Year { get; set; }

    [Range(1, 500)]
```

```
    public decimal Price { get; set; }

    public string Genre { get; set; }

    public int AuthorID { get; set; }

    // Navigation property
    public Author Author { get; set; }
}
```

Run the app. Click **Books** > **Create New Book**. Leave Title blank, set Price to zero, and click **Create**.

Notice how the form automatically adds error messages next to the fields with invalid data. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (using `ModelState`).

Client-side validation alerts the user before the form is submitted, which avoids a round-trip. However, server-side validation is still important, because it guards against malicious requests, and works even if the user has JavaScript disabled.

The data annotation attributes like `[Required]` and `[Range]` only give you basic validation. To validate more complex business rules, you'll need to write additional code that is specific to your domain.

3.1.9 Add an edit page

Add the following methods to `BookController`:

```
public async Task<ActionResult> Edit(int id)
{
    Book book = await FindBookAsync(id);
    if (book == null)
    {
        Logger.LogInformation("Edit: Item not found {0}", id);
        return HttpNotFound();
    }

    ViewBag.Items = GetAuthorsListItems(book.AuthorID);
    return View(book);
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Update(int id, [Bind("Title", "Year", "Price", "Genre", "AuthorID")]
{
    try
    {
        book.BookID = id;
        BookContext.Books.Attach(book);
        BookContext.Entry(book).State = EntityState.Modified;
        await BookContext.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    catch (DbUpdateException)
    {
        ModelState.AddModelError(string.Empty, "Unable to save changes.");
    }
    return View(book);
}

private Task<Book> FindBookAsync(int id)
{
    return BookContext.Books.SingleOrDefaultAsync(book => book.BookID == id);
}
```

This code is very similar to adding a new entity, except for the code needed to update the database:

```
BookContext.Entry(book).State = EntityState.Modified;
await BookContext.SaveChangesAsync();
```

Add a view named *Views/Book/Edit.cshtml* view with the following code:

```
@model ContosoBooks.Models.Book

<div>
  <form asp-controller="Book" asp-action="Update" method="post" asp-route-id="@Model.BookID">
    <div asp-validation-summary="ValidationSummary.ModelOnly" class="text-danger"></div>
    <div class="form-group">
      <label asp-for="Title"></label>
```

```html
      <input asp-for="Title" class="form-control"/>
      <span asp-validation-for="Title" class="text-danger"></span>
    </div>
    <div class="form-group">
      <select asp-for="AuthorID" asp-items="@ViewBag.Items"></select>
    </div>
    <div class="form-group">
      <label asp-for="Year"></label>
      <input asp-for="Year" class="form-control" />
      <span asp-validation-for="Year" class="text-danger"></span>
    </div>
    <div class="form-group">
      <label asp-for="Price"></label>
      <input asp-for="Price" class="form-control" />
      <span asp-validation-for="Price" class="text-danger"></span>
    </div>
    <div class="form-group">
      <label asp-for="Genre"></label>
      <input asp-for="Genre" class="form-control" />
      <span asp-validation-for="Genre" class="text-danger"></span>
    </div>
    <input type="submit" class="btn btn-default" value="Save" />
  </form>
</div>

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

This view defines a form, very similar to the Create form.

3.1.10 Add a delete page

Add the following code to `BookController`.

```csharp
[HttpGet]
[ActionName("Delete")]
public async Task<ActionResult> ConfirmDelete(int id, bool? retry)
{
    Book book = await FindBookAsync(id);
    if (book == null)
    {
        Logger.LogInformation("Delete: Item not found {0}", id);
        return HttpNotFound();
    }
    ViewBag.Retry = retry ?? false;
    return View(book);
}


[HttpPost]
[ValidateAntiForgeryToken]
public async Task<ActionResult> Delete(int id)
{
    try
    {
        Book book = await FindBookAsync(id);
```

```
        BookContext.Books.Remove(book);
        await BookContext.SaveChangesAsync();
    }
    catch (DbUpdateException)
    {
        return RedirectToAction("Delete", new { id = id, retry = true });
    }
    return RedirectToAction("Index");
}
```

Add a view named *Views/Book/Delete.cshtml* view with the following code:

```
@model ContosoBooks.Models.Book

@{
    ViewBag.Title = "Confirm Delete";
}

<h3>Are you sure you want to delete this?</h3>

@if (ViewBag.Retry)
{
    <p class="alert-danger">Error deleting. Retry?</p>
}

<div>
  <dl class="dl-horizontal">
    <dt>
      @Html.DisplayNameFor(model => model.Title)
    </dt>
    <dd>
      @Html.DisplayFor(model => model.Title)
    </dd>

    <dt>
      @Html.DisplayNameFor(model => model.Year)
    </dt>
    <dd>
      @Html.DisplayFor(model => model.Year)
    </dd>
  </dl>

  <div>
    <form asp-controller="Book" asp-action="Delete" method="post">
      <div class="form-group">
        <input type="submit" class="btn btn-default" value="Delete" />
      </div>
    </form>

    <p><a asp-controller="Author" asp-action="Index">Back to List</a></p>
  </div>
</div>
```

The basic flow is:

1. From the details page, the user clicks the "Delete" link.

2. The app displays a confirmation page.

3. The confirmation page is a form. Submitting the form (via HTTP POST) does the actual deletion.

You don't want the "Delete" link itself to delete the item. Performing a delete operation in response to a GET request creates a security risk. For more information, see ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes on Stephen Walther's blog.

3.1.11 Wrapping up

The sample app has equivalent pages for authors. However, they don't contain any new concepts, so I won't show them in the tutorial. You can browse the source code on GitHub.

For information about deploying your app, see Publishing and Deployment.

3.2 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.

3.3 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.

Models

4.1 Model Binding Request Data

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.

4.2 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.

Views

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 `<span>` with the Tag Helper opt-out character:

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

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

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. In the code image below, the Tag Helper prefix is set to "th:", so only those elements using the prefix "th:" support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<span>` 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:
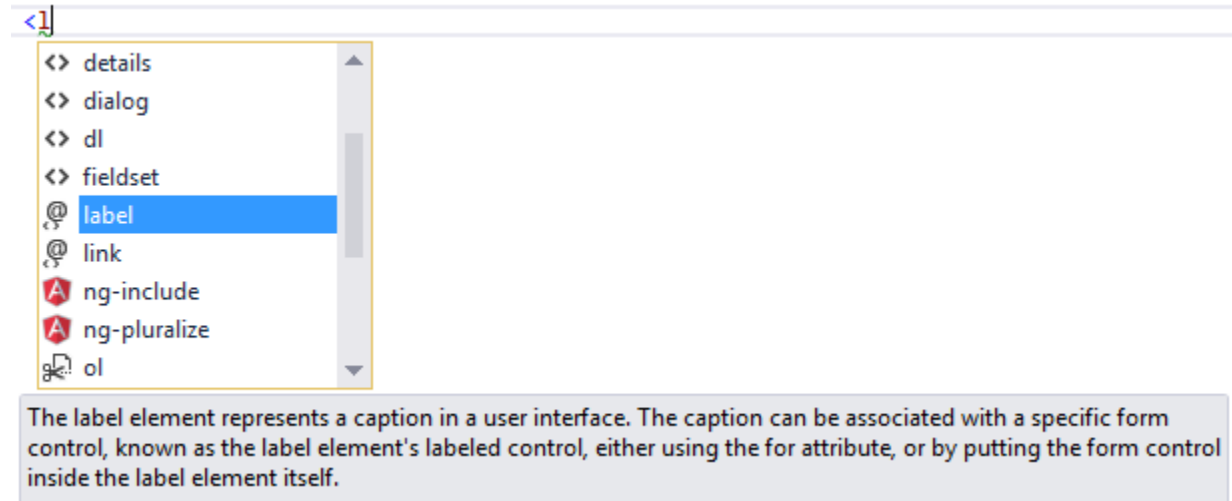
Not only do you get HTML help, but the icon (the "@" symbol with "<>" under it).

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.

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:
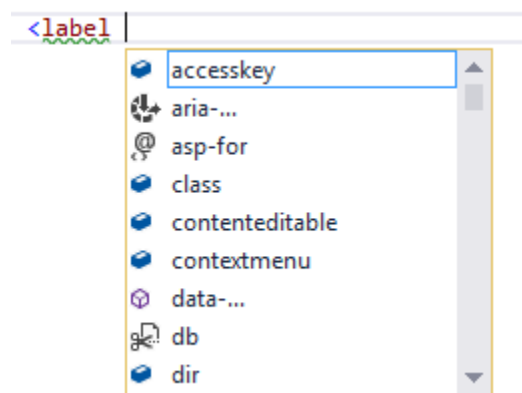
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:
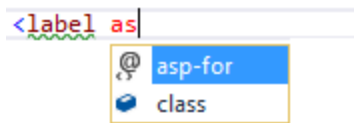
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the default) inside the double quotes (""), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.
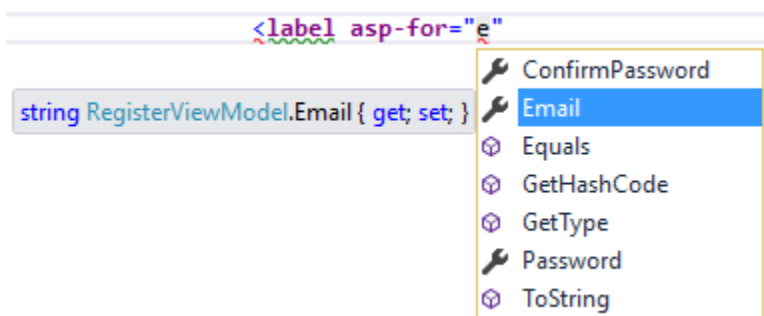
IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:
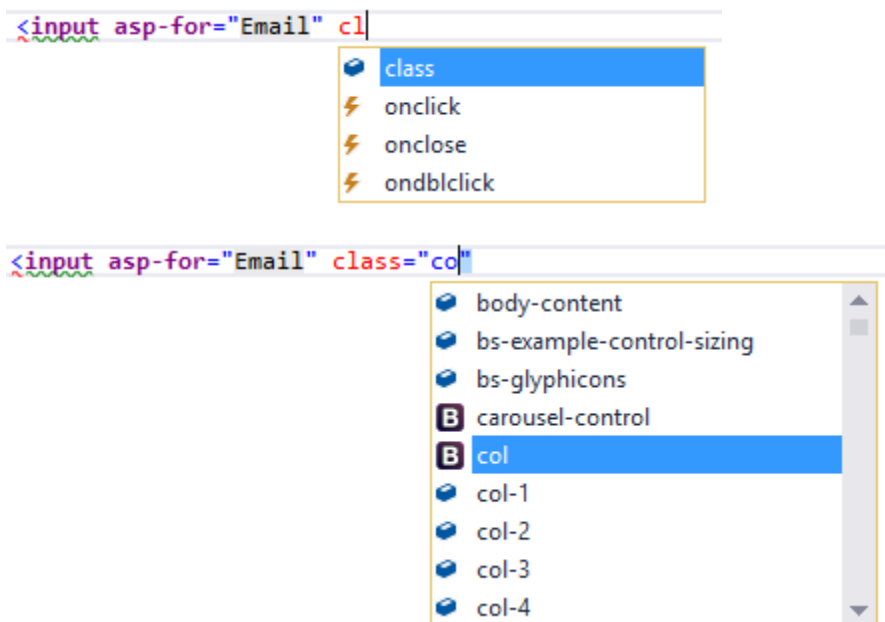
Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

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

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

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

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

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

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

With the Tag Helper version, as soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:
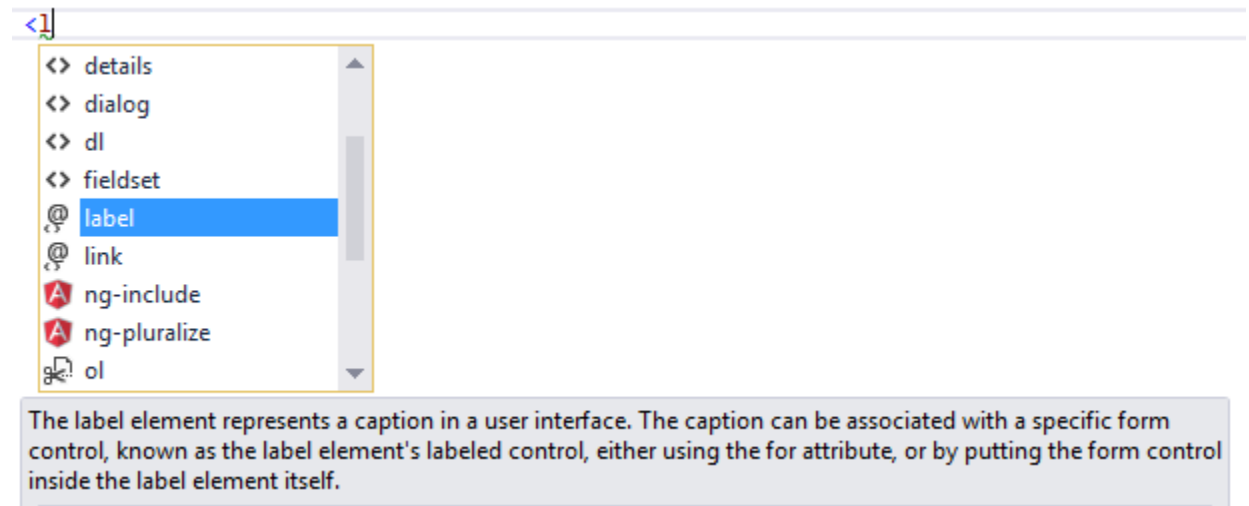
IntelliSense helps you write the entire line. The `LabelTagHelper` also defaults to setting the content of the `asp-for` attribute value ("FirstName") to "First Name"; It converts camel-cased properties to a sentence composed of the property name with a space where each new upper-case letter occurs. In the following markup:

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

generates:

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

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

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

generates:

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

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

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

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

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

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

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

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

Consider the *Email* group:

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

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

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

Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET Web Server controls are declared and invoked on a page.

- Web Server controls have a non-trivial lifecycle that can make developing and debugging difficult.

- Web Server controls allow you to add functionality to the client Document Object Model (DOM) elements by using a client control. Tag Helpers have no DOM.

- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.

- Multiple Tag Helpers can act on the same element (see Avoiding Tag Helper conflicts) while you typically can't compose Web Server controls.

- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.

- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.

- Web Server controls use System.ComponentModel to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

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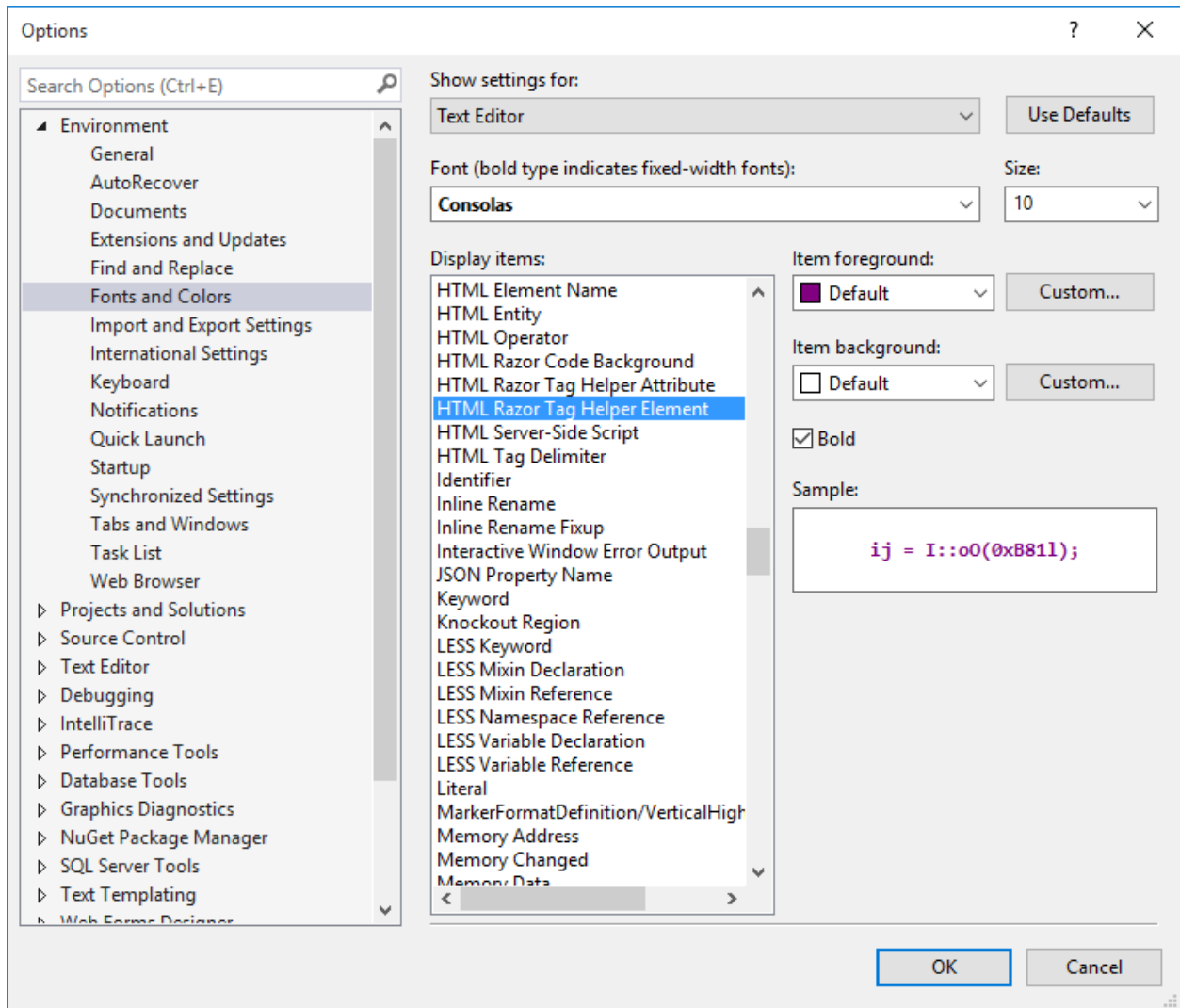
Additional Resources

- 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 Using 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.

5.4.3 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.AspNet.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 **Email**TagHelper is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.

- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides 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 (*AuthoringTagHelpers*). Most developers will prefer to use the wildcard syntax. Using 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</a>`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

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

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 <a href="mailto:Rick@contoso.com" />. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that is self-closing. Tag helpers set the type of the TagMode property after reading a tag.

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 `context` 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.AspNet.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.SetContent("<strong>");
            output.PostContent.SetContent("</strong>");
        }
    }
}
```

Notes:

- The `[TargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run.

In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.

- Because we don't want to replace the existing tag content, we must write the opening `<strong>` tag with the `PreContent.SetContent` method and the closing `</strong>` tag with the `PostContent.SetContent` 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 `[TargetElement]` 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 `[TargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **Bold**TagHelper to `<bold>` tags.

5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[TargetElement]` 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.

```
[TargetElement("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 `[TargetElement]` 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:

```
[TargetElement("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.AspNet.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.SetContent(
$@"<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 `[TargetElement]` 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):

```
[TargetElement("WebsiteInformation")]
```

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

```
[TargetElement("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.cshtml* view. The highlighted markup displays the web site information.

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

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

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="new WebsiteContext {
                                 Version = new Version(1, 3),
                                 CopyrightYear = 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.AspNet.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-InforMation info=Model />
    <div condition="Model.Approved">
        <p>
            This website has <strong surround="em"> @Model.Approved </strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. 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 context.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetContent(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 context.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetContent(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 context.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetContent(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 context.GetChildContentAsync()).GetContent();

            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetContent(Regex.Replace(
                childContent,
                @"\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 = output.Content.IsModified ? output.Content.GetContent() :
                (await context.GetChildContentAsync()).GetContent();

            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetContent(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 context.GetChildContentAsync();
```

That is, we call `GetChildContentAsync` using the `TagHelperContext` 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 context.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)
```

5.4. Tag Helpers

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

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetContent(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.

5.4.4 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 a Service Into a View

ASP.NET MVC 6 now supports injection into a view from a class. For this example, we'll create a simple class that exposes the total *todo* count, completed count and average priority.

1. Examine the *Services\StatisticsService.cs* class.

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

2. Update the *Index* view to inject the *todo* statistical data. Add the `inject` statement to the top of the file:

```
1   @inject TodoList.Services.StatisticsService Statistics
```

3. Add markup calling the StatisticsService to the end of the 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        @await Component.InvokeAsync("PriorityList", 4, true)
6        <h3>Stats</h3>
7        <ul>
8          <li>Items: @await Statistics.GetCount()</li>
9          <li>Completed:@await Statistics.GetCompletedCount()</li>
10         <li>Average Priority:@await Statistics.GetAveragePriority()</li>
11       </ul>
12     </div>
13   </div>
```

4. Register the `StatisticsService` class in the *Startup.cs* file:

```
1   public void ConfigureServices(IServiceCollection services)
2   {
3     // Code removed for brevity.
4     // Add MVC services to the services container.
5     services.AddMvc();
6     services.AddTransient<TodoList.Services.StatisticsService>();
7   }
```

The statistics are displayed:
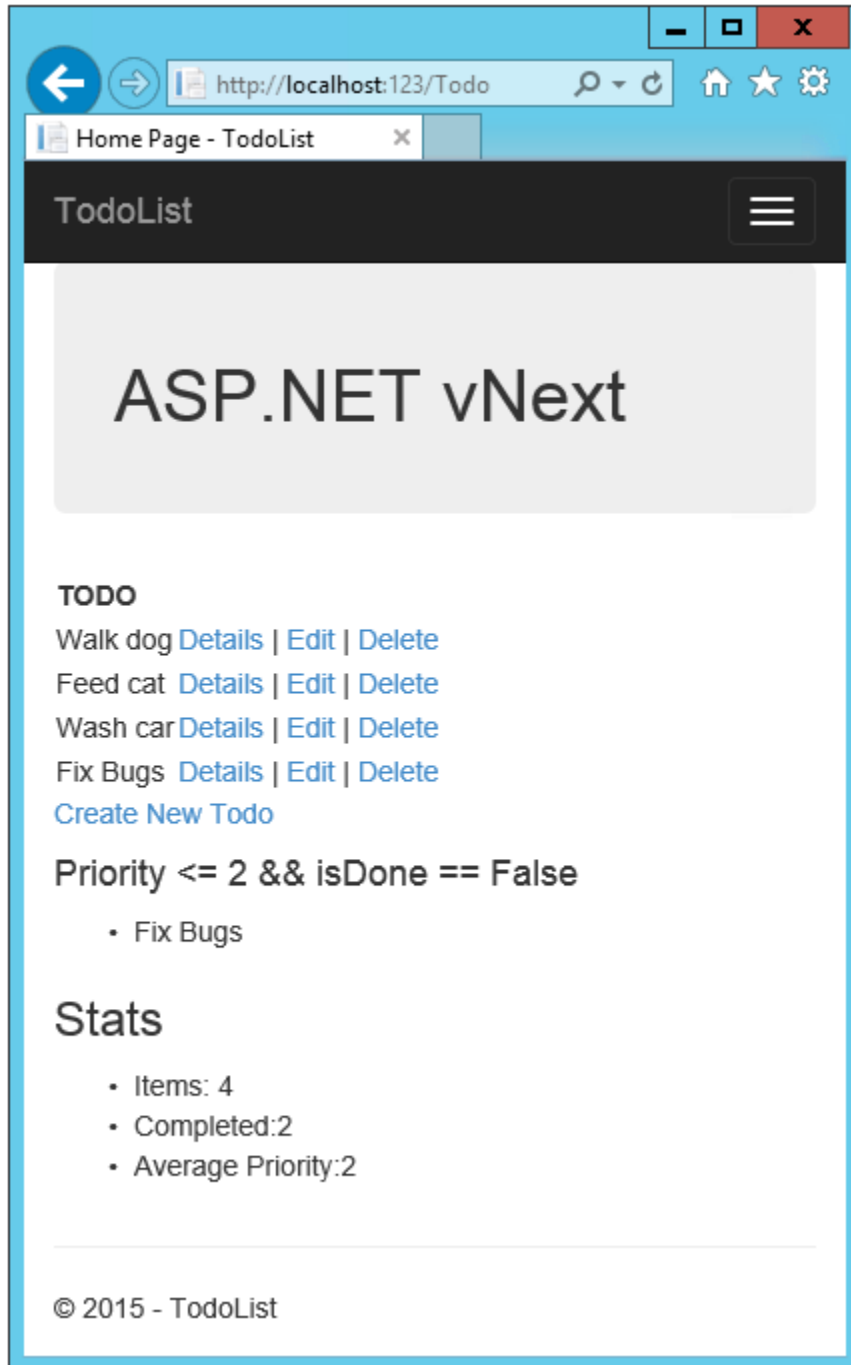
5.7 View Components in MVC 6

By Rick Anderson

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. You can think of a view component as a mini-controller—it's 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.AspNet.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 class, or the name of the class minus the suffix (if we followed convention and used the *ViewComponent* suffix in the class name). If you used the the `ViewComponent` attribute, the class 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)
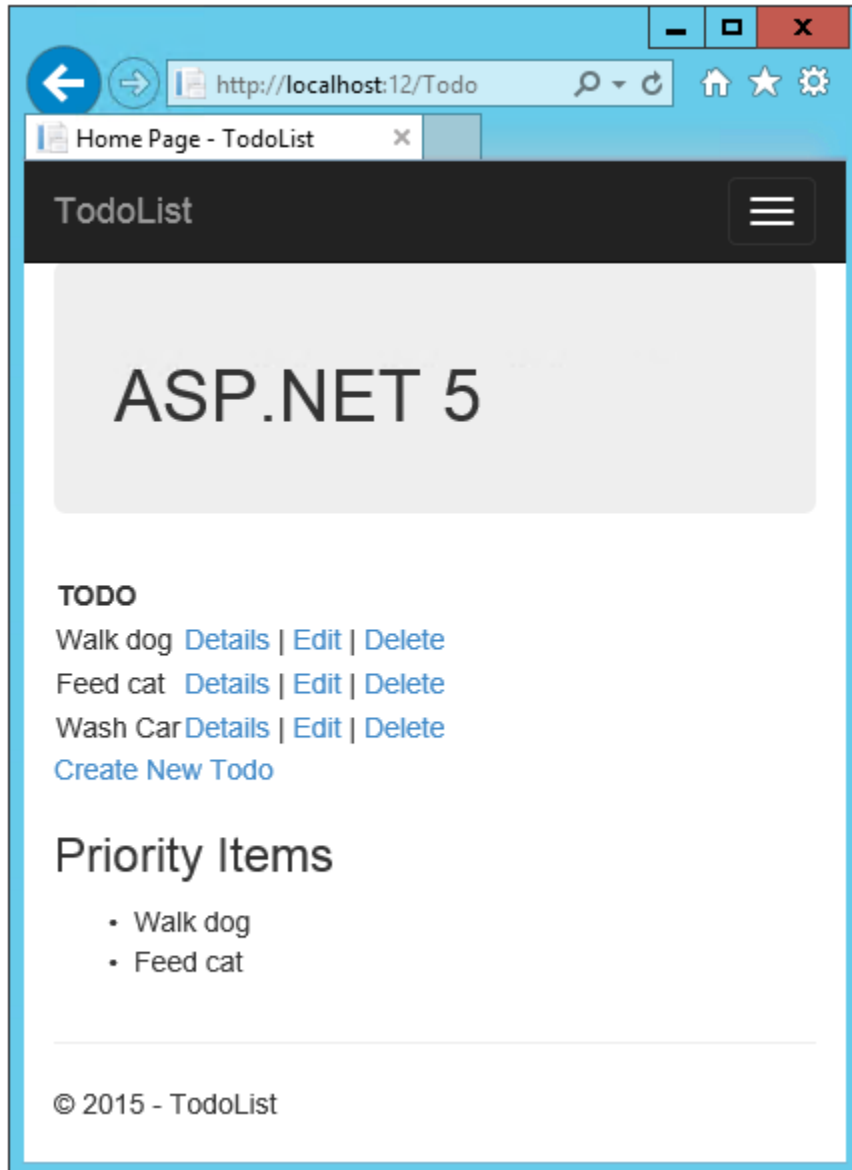
5.7.4 Add InvokeAsync to the priority view component

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:
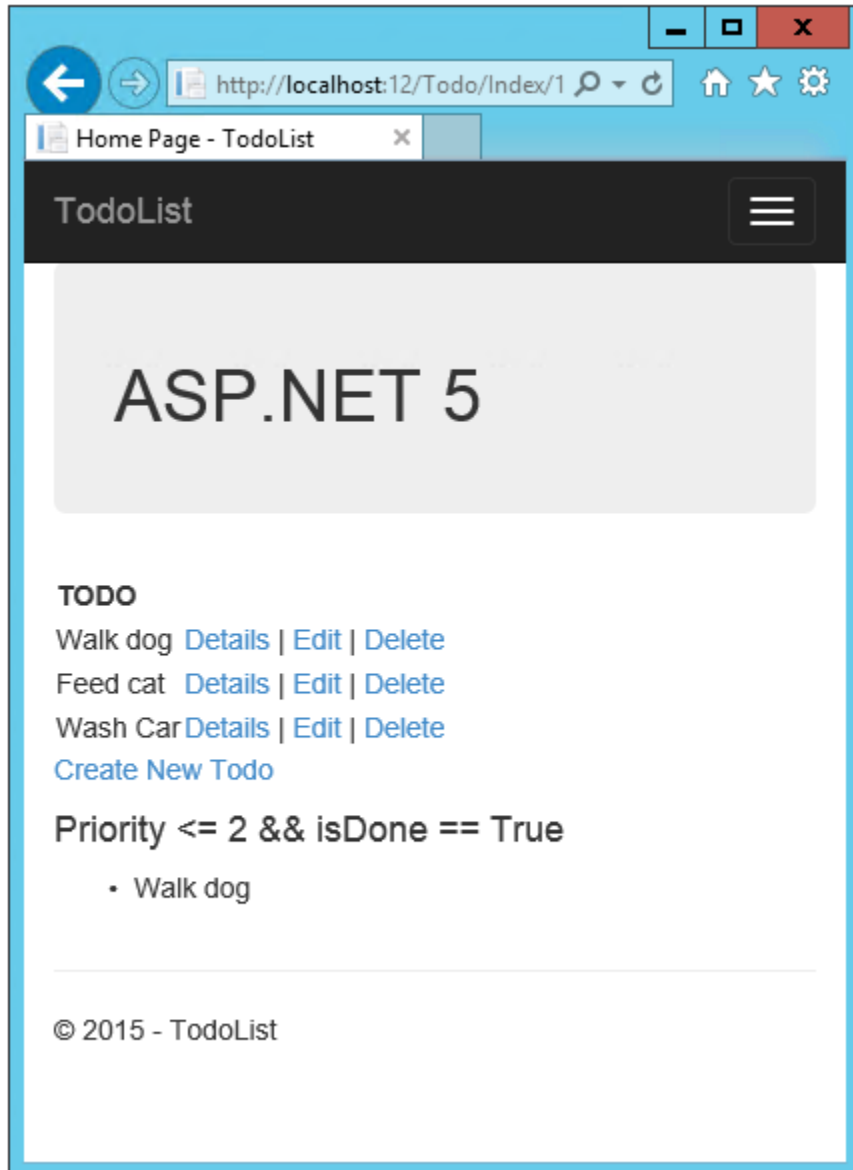
5.7.5 Specifying a view name

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.
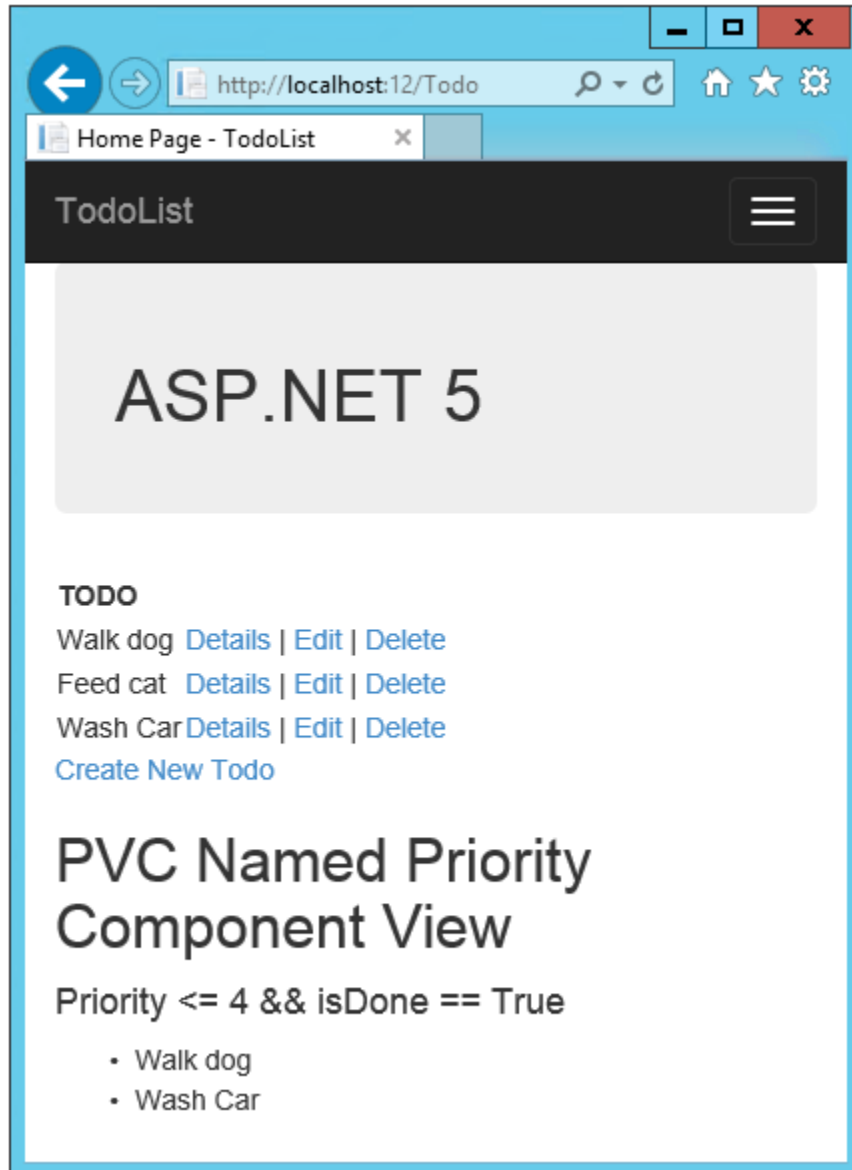
5.8 Creating a Custom View Engine

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 Actions and Action Results

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.

6.2 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

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.

6.6 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.

Security

7.1 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.

7.2 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.

7.3 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.

7.4 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.)

7.4.1 Add the CORS package

In your project.json file, add the following:

```
"dependencies": {
  "Microsoft.AspNet.Cors": "6.0.0-beta8"
},
```

7.4.2 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.

7.4.3 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

8.1 Migrating From ASP.NET MVC 5 to MVC 6

By Steve Smith

Migrating from ASP.NET MVC 5 to ASP.NET 5 and MVC 6 requires a few steps to complete, since ASP.NET 5 introduces a number of new concepts. In this article you will learn how to migrate from the ASP.NET MVC 5 default project template to ASP.NET MVC 6, including initial setup, basic controllers and views, static content, and client side dependencies.

In this article:

- *Create the Initial Project*

- *Create the Destination Solution*

- *Migrate Basic Controllers, Views, and Static Content*

- *Configure Bundling*

Download the finished source from the project created in this article.

8.1.1 Create the Initial Project

For the purposes of this article, we will be starting from the default ASP.NET MVC 5 starter web project, which you can create in Visual Studio 2015 by adding a new web project and choosing MVC 5.
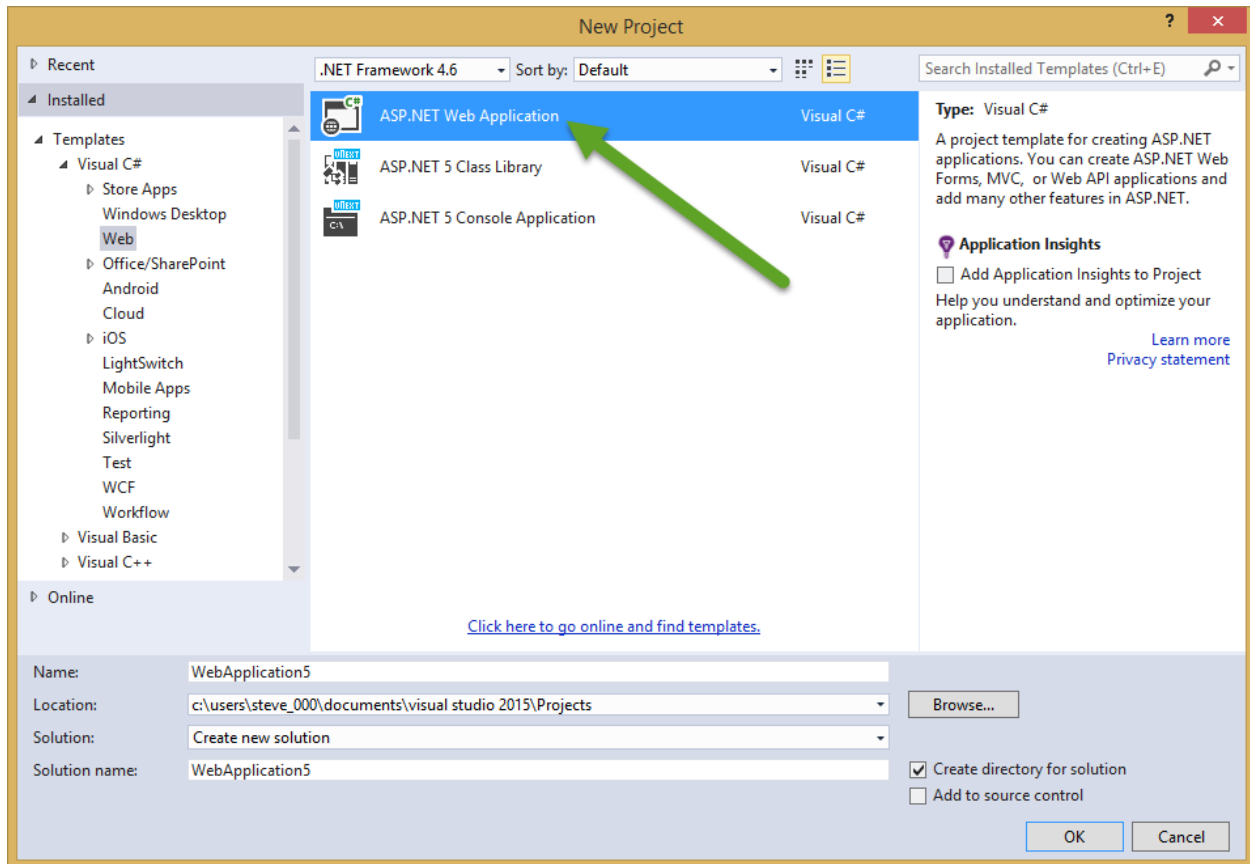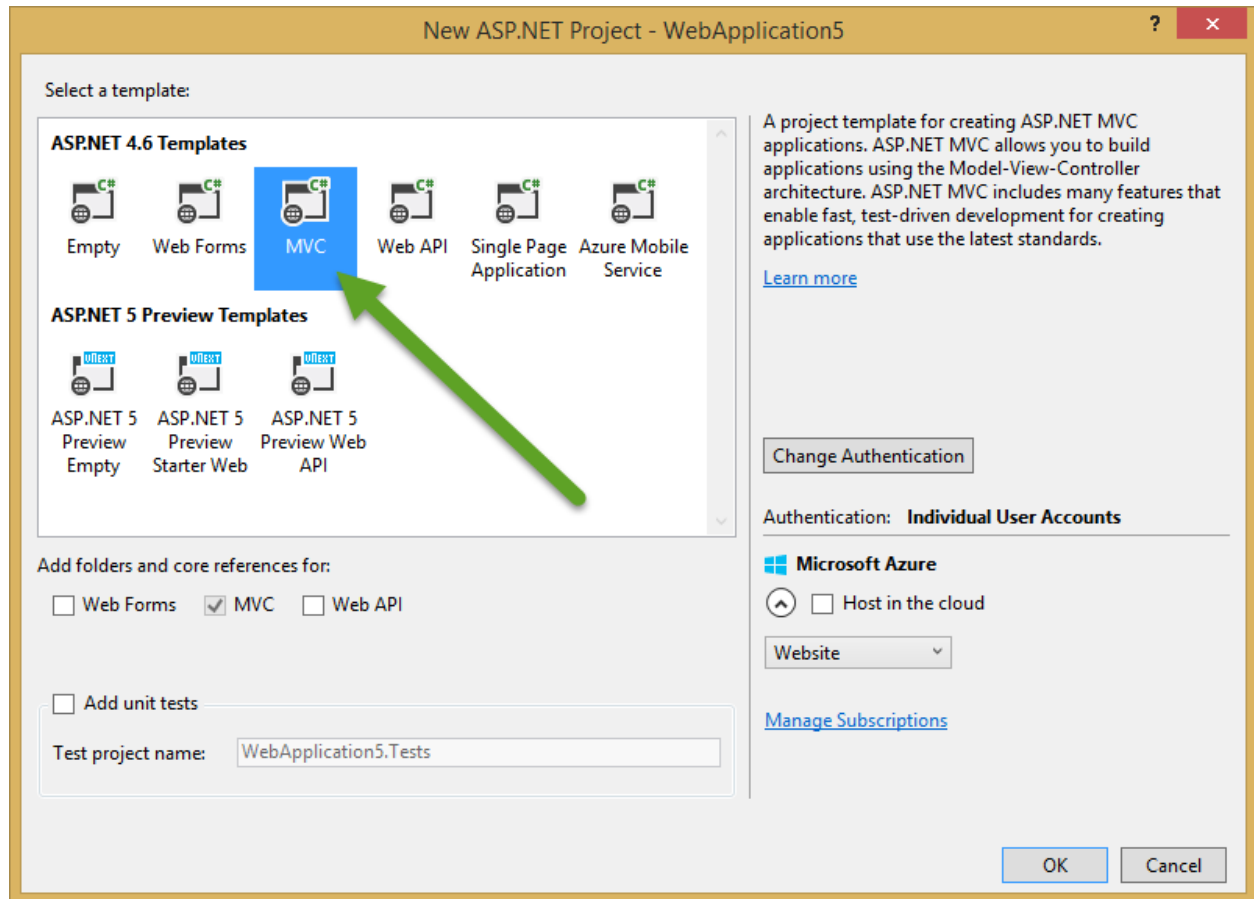
If you prefer, you can view or download the MVC 5 Project used in this article.

This sample web project will demonstrate how to migrate an MVC 5 web project that includes controllers, views, and ASP.NET Identity models, as well as startup and configuration logic common to many MVC 5 projects.

8.1.2 Create the Destination Solution

We will begin our migration by creating a new, empty ASP.NET 5 solution. Create a new project in Visual Studio 2015, choose an ASP.NET Web Application, and then choose the ASP.NET 5 Empty template.
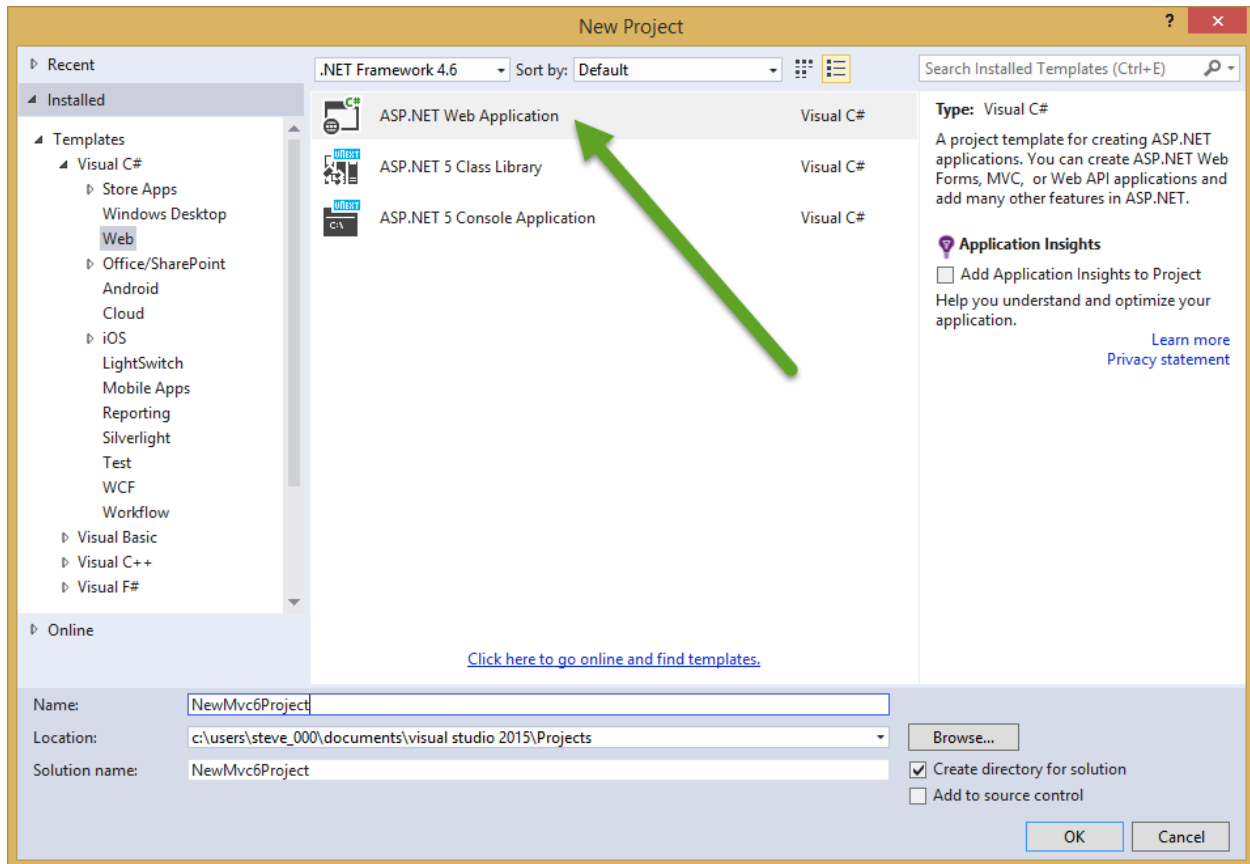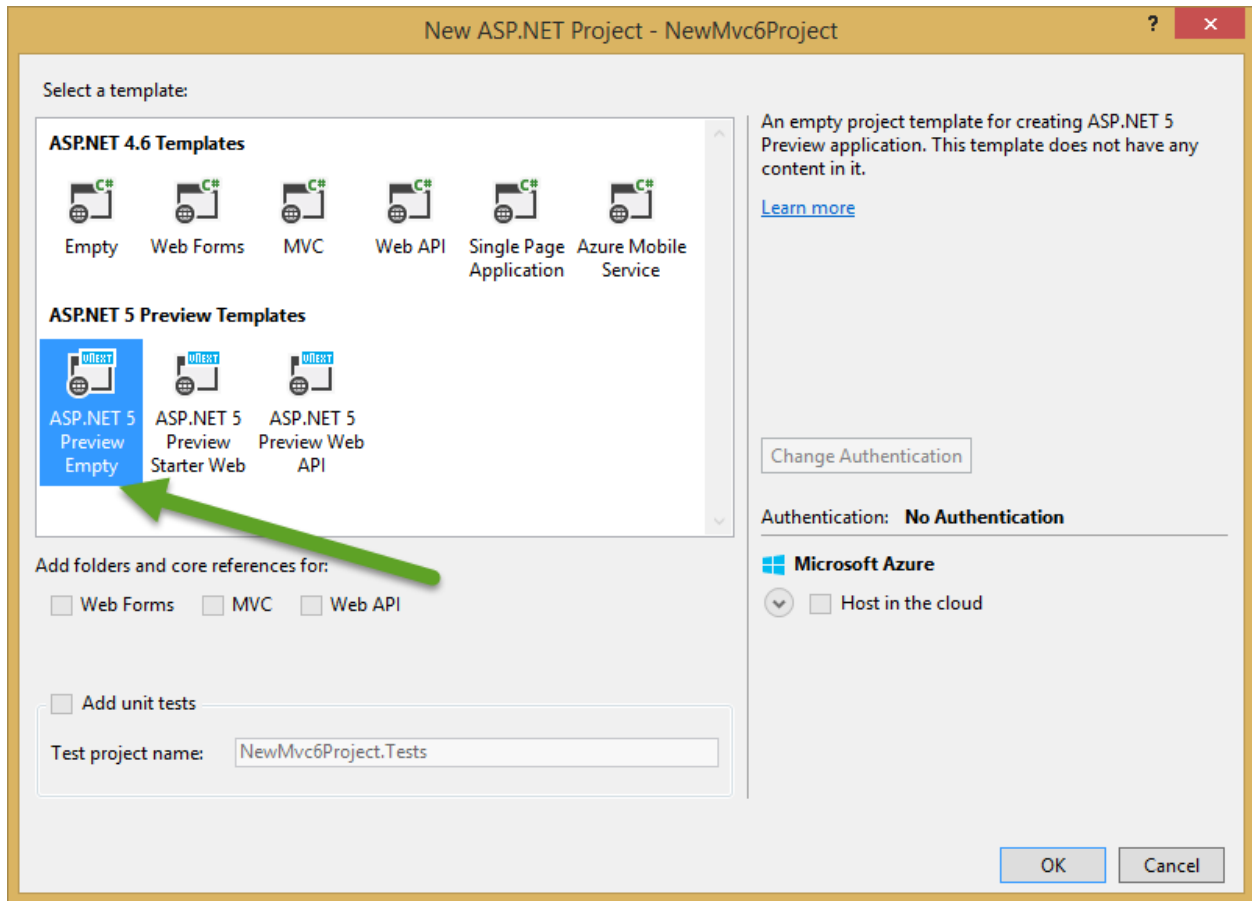
This migration will start from an empty template. If you're already familiar with ASP.NET 5 and its starter templates and there are features in a starter template you would like to take advantage of, you may wish to start from another template. The next step is to configure the site to use MVC. This requires changes to the project.json file and Startup.cs file. First, open project.json and add "Microsoft.AspNet.Mvc" to the "dependencies" property:

```
"dependencies": {
        "Microsoft.AspNet.Server.IIS": "1.0.0-beta4",
        "Microsoft.AspNet.Mvc": "6.0.0-beta4"
},
```

Now open Startup.cs and modify it as follows:

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

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

At this point we are ready to create a simple Controller and View. Add a Controllers folder and a Views folder to the project. Add an MVC Controller called HomeController.cs class to the Controllers folder and a Home folder in the Views folder. Finally, add an Index.cshtml MVC View Page to the Views/Home folder. The project structure should be as shown:
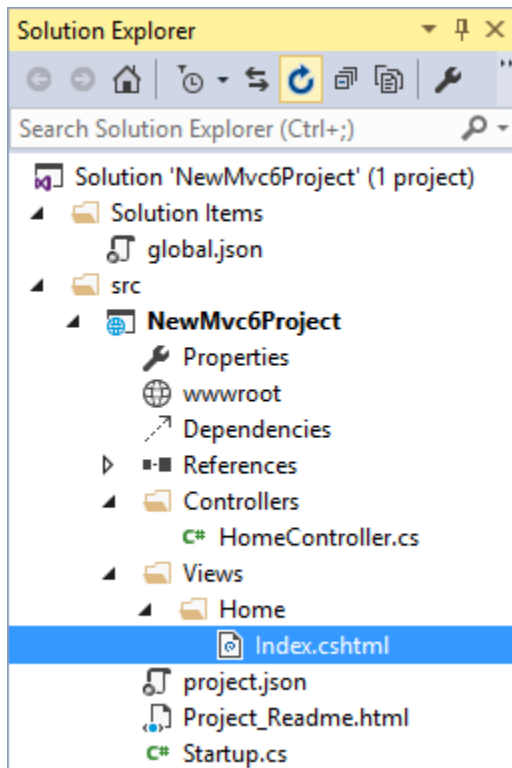
Modify Index.cshtml to show a welcome message:

```html
<h1>Hello world!</h1>
```

Run the application - you should see Hello World output in your browser.
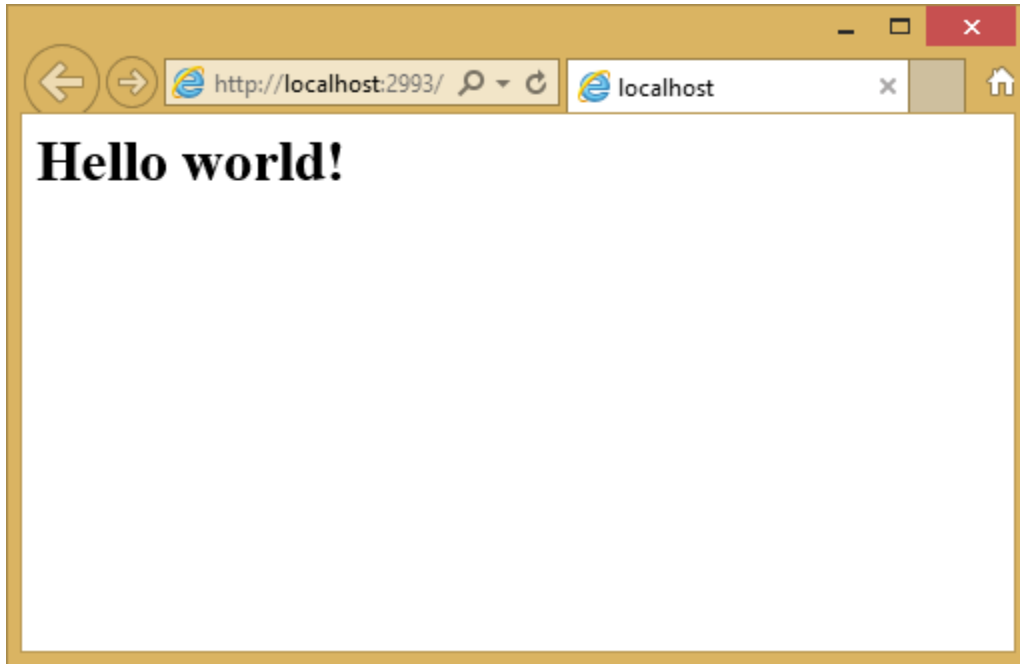
8.1.3 Migrate Basic Controllers, Views, and Static Content

Now that we've confirmed we have a simple, working ASP.NET MVC 6 project, it's time to start migrating functionality from the source project. There are many different ways one can approach this task. We will need to move all of the client-side content files (CSS, fonts, scripts), all of the controllers, views, and models, and migrate configured features like bundling, filters, and identity. Let's begin by replacing our simple "hello world" implementation of HomeController with the actual HomeController and Views from the source project.

Copy each of the methods from the source HomeController to the HomeController we added to the project in the previous section. Note that in MVC 5, actions typically returned ActionResult, but in MVC 6 this has changed to IActionResult (though it will still compile if you leave it as ActionResult).

Next, create new MVC View Pages in the Views/Home folder for About and Contact. Copy the contents of the corresponding views in the old project to these new views, as well as Index.cshtml. At this point you should once again be able to run the new application, and although the styles are not yet in place, you should see the correct content on the home page as well as /home/about and /home/contact (contact is shown here):
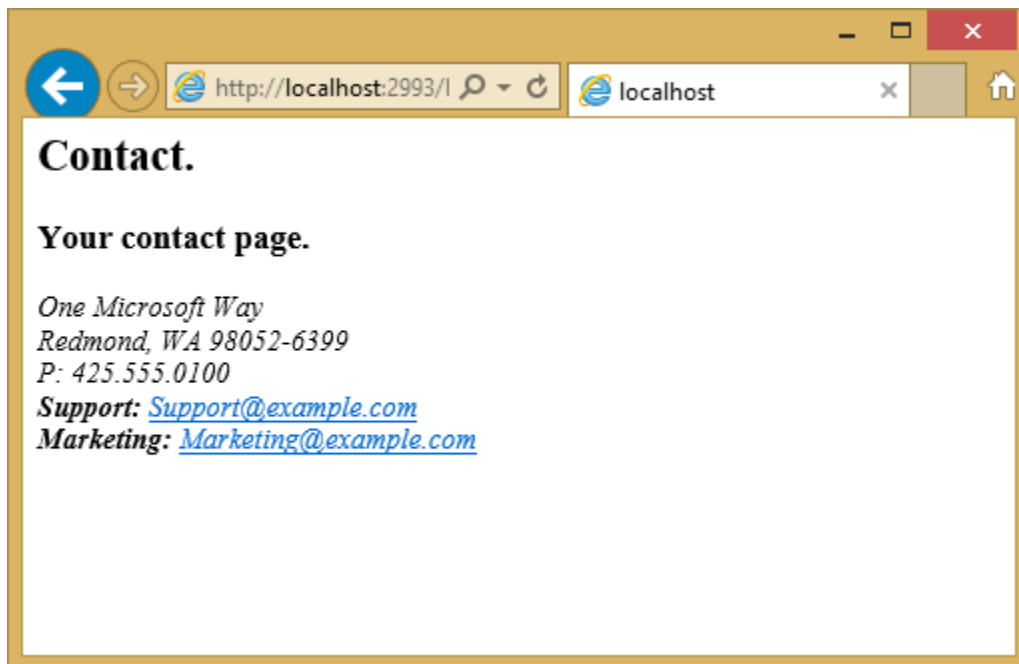
In MVC 5 and previous versions of ASP.NET, static content was hosted from the root of the web project, and was intermixed with server-side files. In MVC 6, all static content files are hosted from the /wwwroot folder, so we will need to adjust where we are storing our static content files. For instance, we can copy the favicon.ico file from the root of the original project to the /wwwroot folder in the new project.

The MVC 5 project uses Bootstrap for its styling, with files stored in /Content and /Scripts and referenced in /Views/Shared/_Layout.cshtml. We could simply copy the bootstrap.js and bootstrap.css files from the old project to the /wwwroot folder in the new project, but there are better ways to handle these kinds of client-side library dependencies in ASP.NET 5.

In our new project, we'll add support for Bootstrap (and other client-side libraries), but we'll do so using the new support for client-side build tooling using Bower and grunt. First, add a new Bower JSON Configuration File to the project root, called bower.json. In its "dependencies" property, add bootstrap, jquery, jquery-validation, and jquery-validation-unobtrusive. Add new properties for these items to the "exportsOverride" property as well, so that the complete bower.json file looks like this:

```
{
        "name": "NewMvc6Project",
        "private": true,
        "dependencies": {
                "bootstrap": "3.0.0",
                "jquery": "1.10.2",
                "jquery-validation": "1.11.1",
                "jquery-validation-unobtrusive": "3.2.2"
        },
        "exportsOverride": {
                "bootstrap": {
                        "js": "dist/js/*.*",
                        "css": "dist/css/*.*",
                        "fonts": "dist/fonts/*.*"
                },

                "jquery": {
                        "": "jquery.{js,min.js,min.map}"
                },
```

```
                "jquery-validation": {
                        "": "jquery.validate.js"
                },
                "jquery-validation-unobtrusive": {
                        "": "jquery.validate.unobtrusive.{js,min.js}"
                }
        }
}
```

Bower will automatically download the specified dependencies, but for now the files are not yet in the wwwroot folder, and so cannot be requested by the browser:
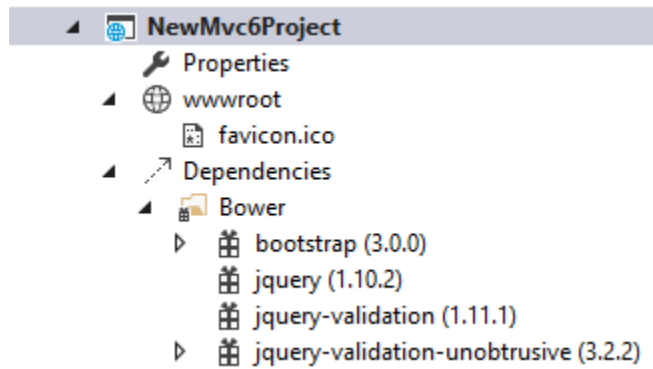
Next, we will configure Gulp to process these files and place them where we want them in the wwwroot folder. First, we need to make sure Gulp is installed locally for the project. This is accomplished using NPM, which is similar to Bower but requires a different configuration file, "package.json". Add a new NPM configuration file to the root of the project, called package.json. Add *gulp*, *rimraf*, and *gulp-concat* to the devDependencies property (you should get Intellisense as you type each package name). When you're finished, your file should look similar to this one:

```
1  {
2      "version": "1.0.0",
3      "name": "NewMvc6Project",
4      "private": true,
5      "devDependencies": {
6          "gulp": "3.8.11",
7          "rimraf": "2.3.2",
8          "gulp-concat": "2.5.2"
9      }
10 }
```

Save your changes. You should see a new NPM folder in your project, under Dependencies, and it should now include gulp (3.8.11) as well as the related packages. In addition to Gulp itself, these two packages will allow us to clean up folders before we copy files to them, and to concatenate two or more files together, to achieve bundling.

Next, add a new Gulp Configuration file (Gulpfile.js) to the root of the project.
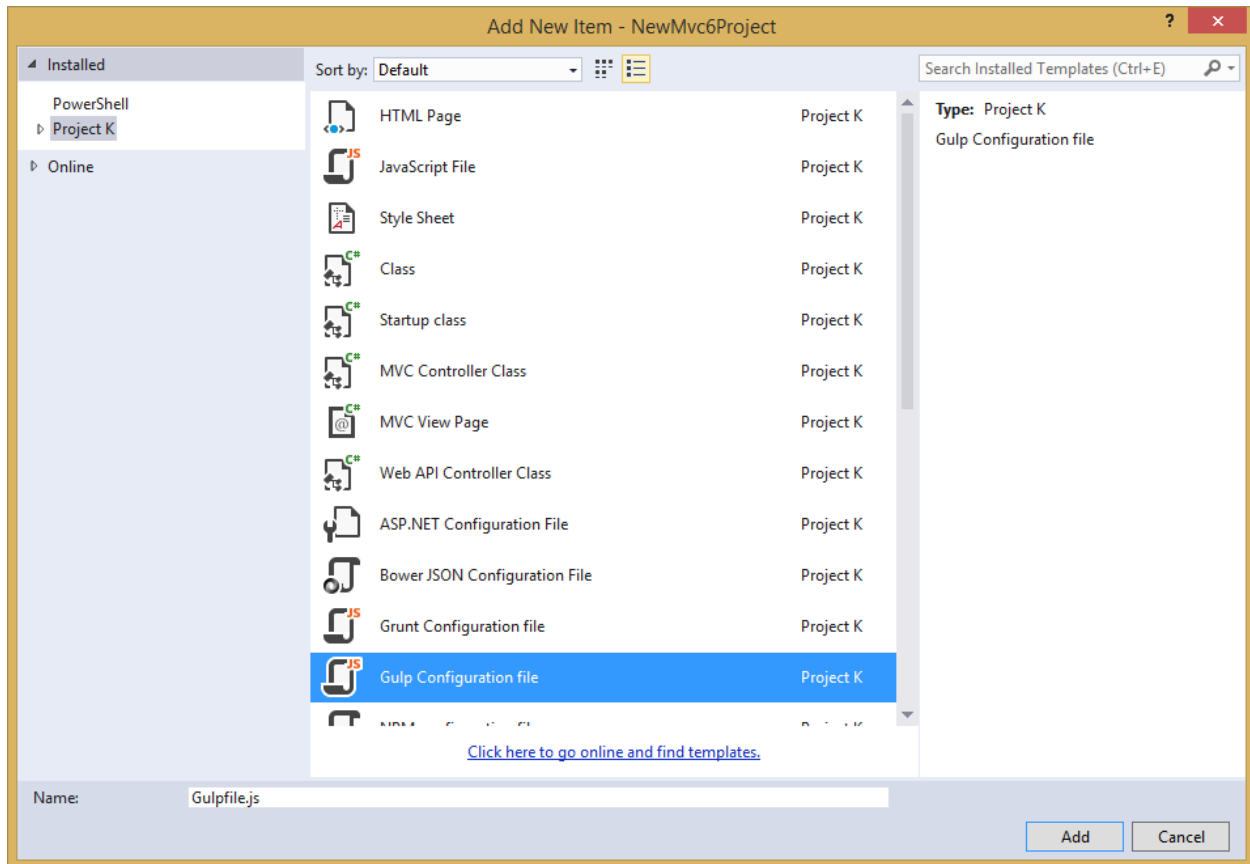
We need to configure Gulp to use Bower, and then register tasks associated with this configuration. Modify Gulpfile.js to match this file:

```javascript
var gulp = require('gulp');
var rimraf = require('rimraf');

var paths = {
        bower: "./bower_components/",
        lib: "./wwwroot/lib/"
};

gulp.task('clean', function (callback) {
        rimraf(paths.lib, callback);
});

gulp.task('default', ['clean'], function () {
        var bower = {
                "bootstrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}",
                "jquery": "jquery/jquery*.{js,map}",
                "jquery-validation": "jquery-validation/jquery.validate.js",
                "jquery-validation-unobtrusive":
                        "jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"
        };

        for (var destinationDir in bower) {
                gulp.src(paths.bower + bower[destinationDir])
                        .pipe(gulp.dest(paths.lib + destinationDir));
        }
```

```
});
```

Now that we've finished setting things up, we're ready to let these tools manage our static files and client-side dependencies for us. Right click on Gulpfile.js in your project, and select Task Runner Explorer. Double-click on the default task to run it.
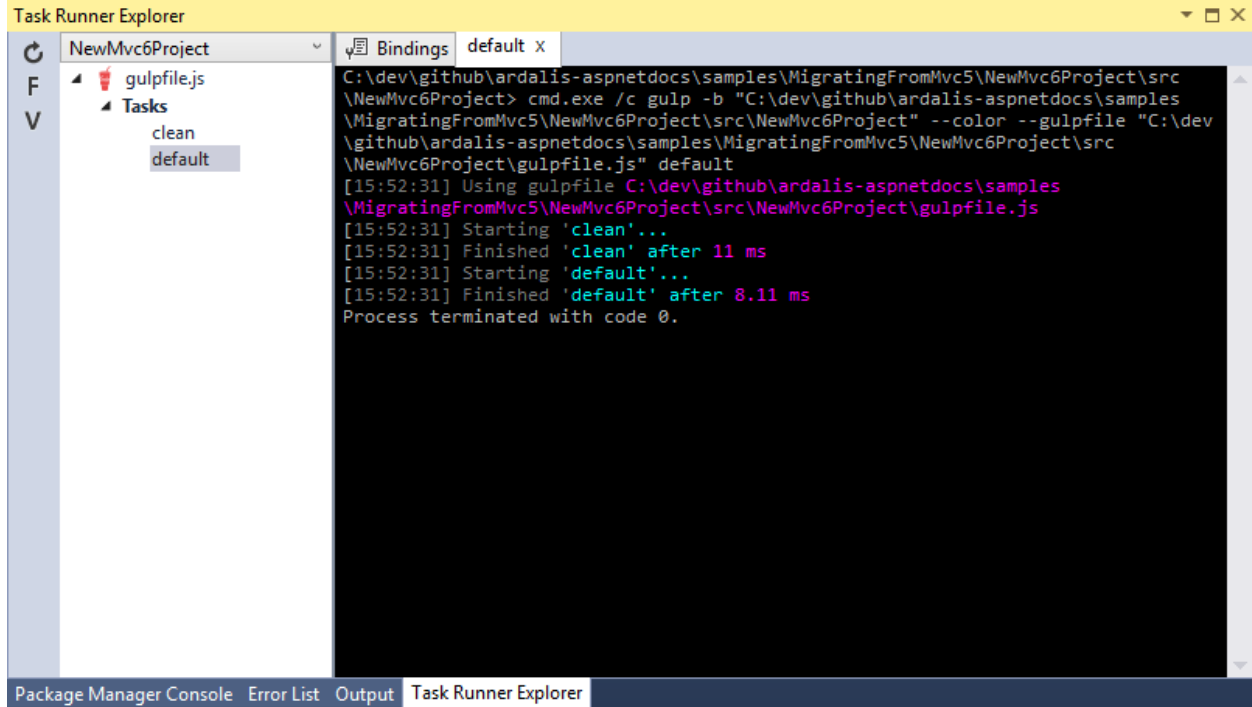
The output should show that the process completed without errors, and you should see that it copied some packages to the wwwroot/lib folder. Open the wwwroot/lib folder in project explorer, and you should find that the client-side dependencies (bootstrap, jquery, etc.) have all been copied into this folder:

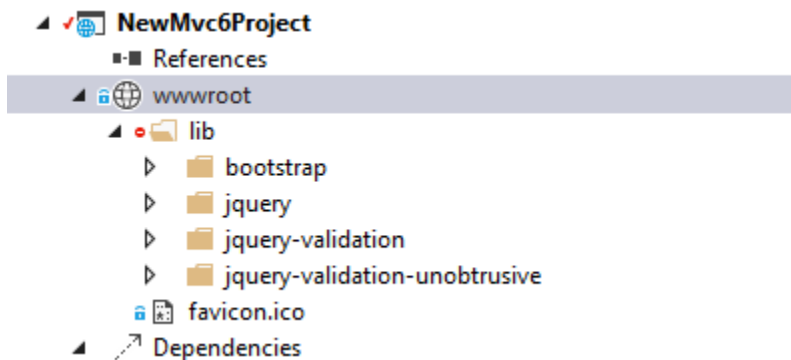Now that the required bootstrap files are available in the wwwroot folder, the next step is to modify our Views to include references to these files. Copy the _ViewStart.cshtml file from the original project's Views folder into the new project's Views folder. In this case, it references /Shared/_Layout.cshtml, which is the next file we need to copy (create a new Shared folder in /Views and copy _Layout.cshtml from the old project to it). Open _Layout.cshtml and make the following changes:

- Replace @Styles.Render("~/Content/css") with a <link> element to load bootstrap.css (see below)

- Remove @Scripts.Render("~/bundles/modernizr")

- Comment out the line with @Html.Partial("_LoginPartial") - we'll return to it shortly (surround the line with @*...*@)

- Replace @Scripts.Render("~/bundles/jquery") with a <script> element (see below)
- Replace @Scripts.Render("~/bundles/bootstrap") with a <script> element (see below)

The CSS link to use:

```
<link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />
```

The script tags to use:

```
<script src="~/lib/jquery/jquery.js"></script>
<script src="~/lib/bootstrap/js/bootstrap.js"></script>
```

The complete _Layout.cshtml file should look like this at the moment:

```
_Layout.cshtml
    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>@ViewBag.Title - My ASP.NET Application</title>
    7        <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />
    8
    9    </head>
   10    <body>
   11        <div class="navbar navbar-inverse navbar-fixed-top">
   12            <div class="container">
   13                <div class="navbar-header">
   14                    <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
   15                        <span class="icon-bar"></span>
   16                        <span class="icon-bar"></span>
   17                        <span class="icon-bar"></span>
   18                    </button>
   19                    @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
   20                </div>
   21                <div class="navbar-collapse collapse">
   22                    <ul class="nav navbar-nav">
   23                        <li>@Html.ActionLink("Home", "Index", "Home")</li>
   24                        <li>@Html.ActionLink("About", "About", "Home")</li>
   25                        <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
   26                    </ul>
   27                    @*@Html.Partial("_LoginPartial")*@
   28                </div>
   29            </div>
   30        </div>
   31        <div class="container body-content">
   32            @RenderBody()
   33            <hr />
   34            <footer>
   35                <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
   36            </footer>
   37        </div>
   38
   39        <script src="~/lib/jquery/jquery.js"></script>
   40        <script src="~/lib/bootstrap/js/bootstrap.js"></script>
   41        @RenderSection("scripts", required: false)
   42    </body>
   43    </html>
```

View the site in the browser. It should now load correctly, with the expected styles in place.

8.1.4 Configure Bundling

The ASP.NET MVC 5 starter web template utilized ASP.NET's built-in support for bundling. In ASP.NET MVC 6, this functionality is better performed using client build steps, like we have already configured to manage our client-side dependencies. Instead of maintaining bundling functionality in a static configuration class that runs on the server, the minification and combination of files is done as part of the build process, using Gulp.

You can learn more about configuring Gulp here.(*TODO*)

To simply bundle the jQuery and bootstrap scripts together into a single, minified, file, we can use the gulp-concat task. First, update package.json to require gulp-concat in "devDependencies":

```
"devDependencies": {
"gulp": "3.8.11",
"rimraf": "2.3.2",
"gulp-concat": "2.5.2"
}
```

Save the package.json file and the new package should be installed. You can confirm by checking in the Dependencies/NPM folder to see that the gulp-concat package is listed there. Next, we will add a concat task to Gulpfile.js. Add the highlighted sections:

```
1   /*
2   This file in the main entry point for defining Gulp tasks and using Gulp plugins.
3   Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
4   */
5
6   var gulp = require('gulp');
7   var rimraf = require('rimraf');
8   var concat = require('gulp-concat');
9
10  var paths = {
11      bower: "./bower_components/",
12      lib: "./wwwroot/lib/"
13  };
14
15  gulp.task('clean', function (callback) {
16      rimraf(paths.lib, callback);
17  });
18
19  gulp.task('default', ['clean'], function () {
20      var bower = {
21          "bootstrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}",
22          "jquery": "jquery/jquery*.{js,map}",
23          "jquery-validation": "jquery-validation/jquery.validate.js",
24          "jquery-validation-unobtrusive":
25              "jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"
26      };
27
28      for (var destinationDir in bower) {
29          gulp.src(paths.bower + bower[destinationDir])
30                      .pipe(gulp.dest(paths.lib + destinationDir));
31      }
32  });
33
34  gulp.task('concat', function () {
35      gulp.src([paths.lib + 'bootstrap/js/bootstrap.min.js',
36              paths.lib + 'jquery/jquery.min.js'])
37          .pipe(concat("bundle.js"))
38          .pipe(gulp.dest(paths.lib));
39  });
```

Save Gulpfile.js, then open the Task Runner Explorer. Right click on the concat task and run it. You should see the output, which should show that it runs without errors. In your solution explorer, you should see the bundle.js file in wwwroot/lib. You can see all of this working in the screenshot:
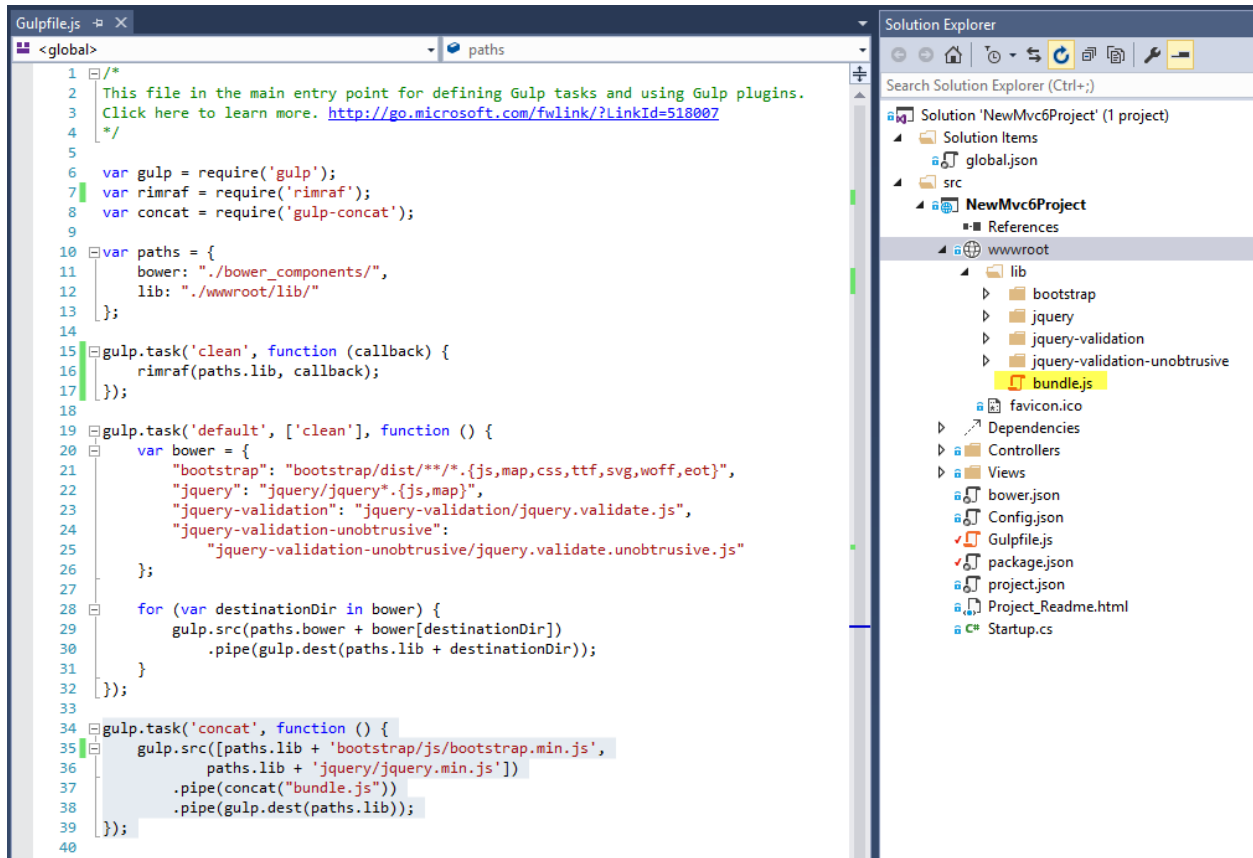
All that remains it to update _Layout.cshtml and replace the last two <script> elements with a single <script> element that loads bundle.js:

```
<script src="~/lib/bundle.js"></script>
```

Refresh the site in a browser, and you can see that the calls to load jQuery.js and bootstrap.js have been replaced with a single call to bundle.js:
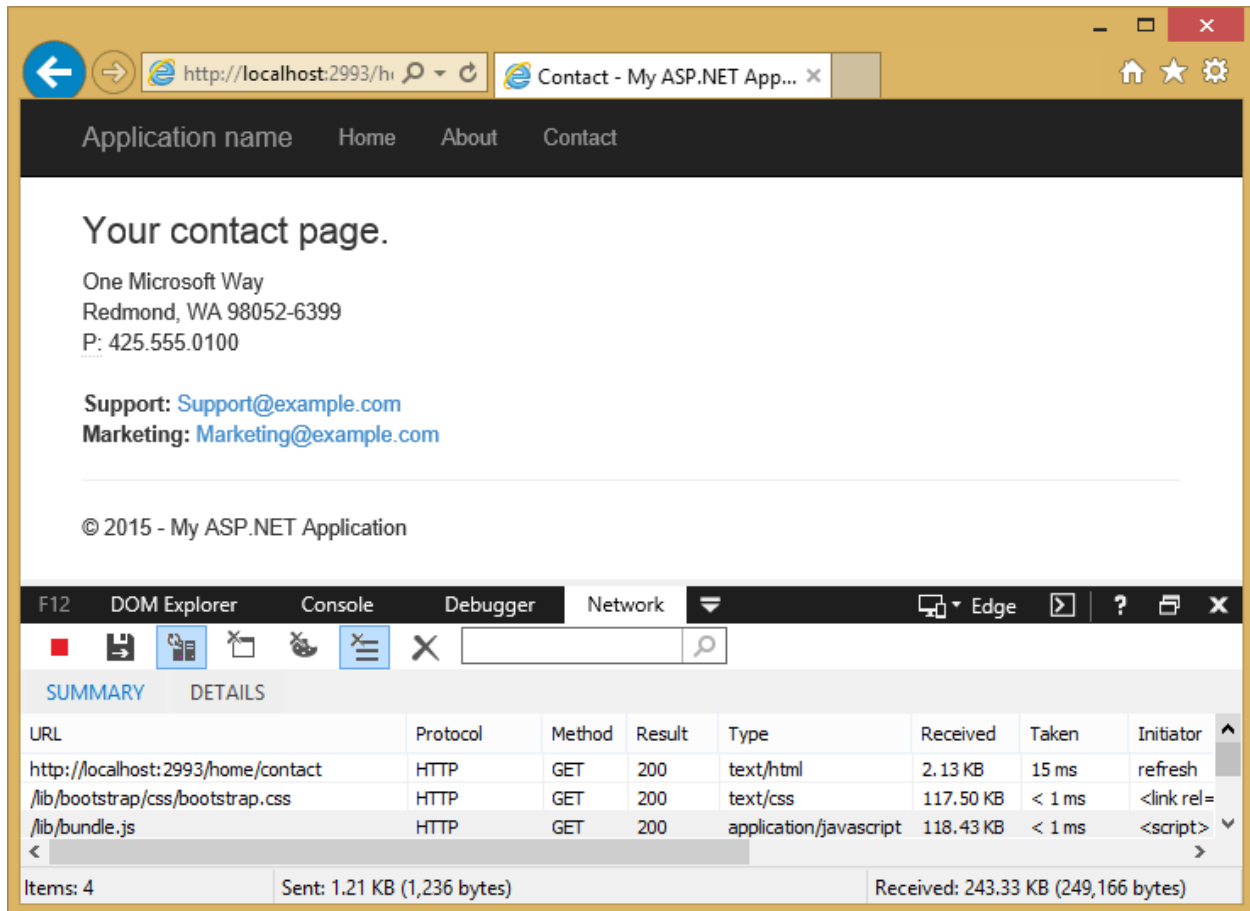
8.1.5 Summary

Migrating from ASP.NET MVC 5 to ASP.NET MVC 6 requires several steps, but is not overly difficult. Basic features like the models, views, and controllers that comprise an MVC application can be migrated largely without changes. Most of the changes affect static content and features related to static content, like bundling, as well as configuration steps for the application. By following the steps in this example, you should be able to quickly migrate most ASP.NET MVC 5 applications.

8.2 Migrating Configuration From ASP.NET MVC 5 to MVC 6

By Steve Smith

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:

- Set up Configuration
- Migrate Configuration Settings from web.config

You can download the finished source from the project created in this article HERE (**TODO**).

8.2.1 Set up 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 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, if desired, but typically ASP.NET 5 projects will place configuration values in a JSON-formatted file, such as config.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 configure Configuration using the default MVC 6 settings, add the following constructor to the Startup.cs class in the root of the project:

```csharp
public IConfiguration Configuration { get; set; }

public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
{
        var configurationBuilder = new ConfigurationBuilder()
                .SetBasePath(appEnv.ApplicationBasePath)
                .AddJsonFile("config.json");
        Configuration = configurationBuilder.Build();
}
```

Note that at this point the Startup.cs file will not compile, as we still need to add some using statements and pull in some dependencies. Add the following two using statements:

```csharp
using Microsoft.Framework.Configuration;
```

Next, open project.json and add the Microsoft.Framework.ConfigurationModel.Json dependency:

```json
{
        "webroot": "wwwroot",
        "version": "1.0.0-*",
        "dependencies": {
                ...
                "Microsoft.Framework.Configuration": "1.0.0-beta8",
                "Microsoft.Framework.Configuration.Json": "1.0.0-beta8"
        },
        ...
}
```

Finally, add a config.json file to the root of the project.
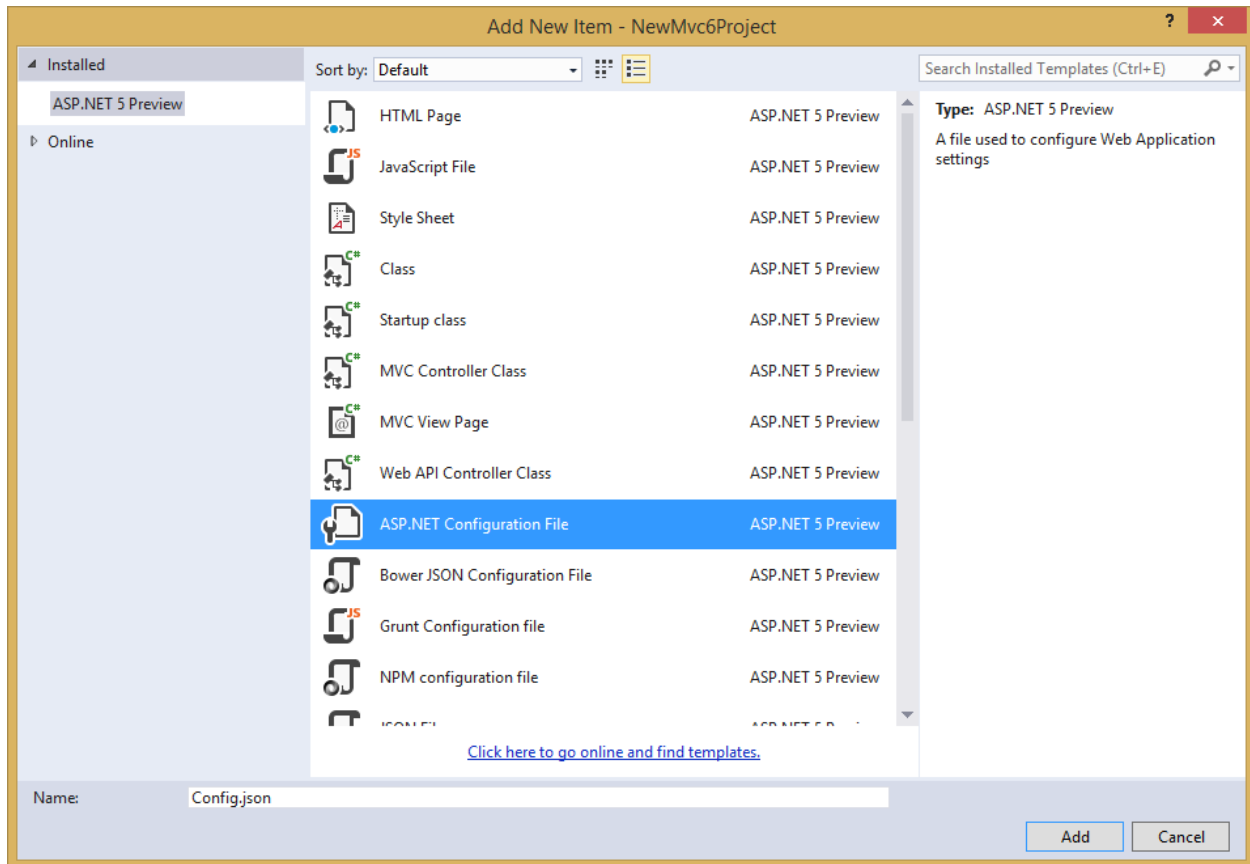
8.2.2 Migrate Configuration Settings from Web.config

Our ASP.NET MVC 5 project included the required database connection string in Web.config, in the <connection-Strings> element. In our MVC 6 project, we are going to store this information in the config.json file. Open Config.json, and you should see that it already includes the following:

```
{
        "Data": {
                "DefaultConnection": {
                        "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trust
                }
        }
}
```

Change the name of the Database from _CHANGE_ME. In the case of this migration, we are going to point to a new database, which we'll name NewMvc6Project to match our migrated project name.

8.2.3 Summary

ASP.NET 5 places all Startup logic for the application in a single file in which 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.

8.3 Migrating From ASP.NET Web API 2 to MVC 6

By Steve Smith

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.

8.3.1 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:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

WebApiConfig is defined in App_Start, and has just one static Register method:

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

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
```

```
11          {
12              // Web API configuration and services
13
14              // Web API routes
15              config.MapHttpAttributeRoutes();
16
17              config.Routes.MapHttpRoute(
18                  name: "DefaultApi",
19                  routeTemplate: "api/{controller}/{id}",
20                  defaults: new { id = RouteParameter.Optional }
21              );
22          }
23      }
24  }
```

This class configures attribute routing, although it's not actually being used in the project, as well as the routing table that Web API 2 uses. 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.

8.3.2 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'.
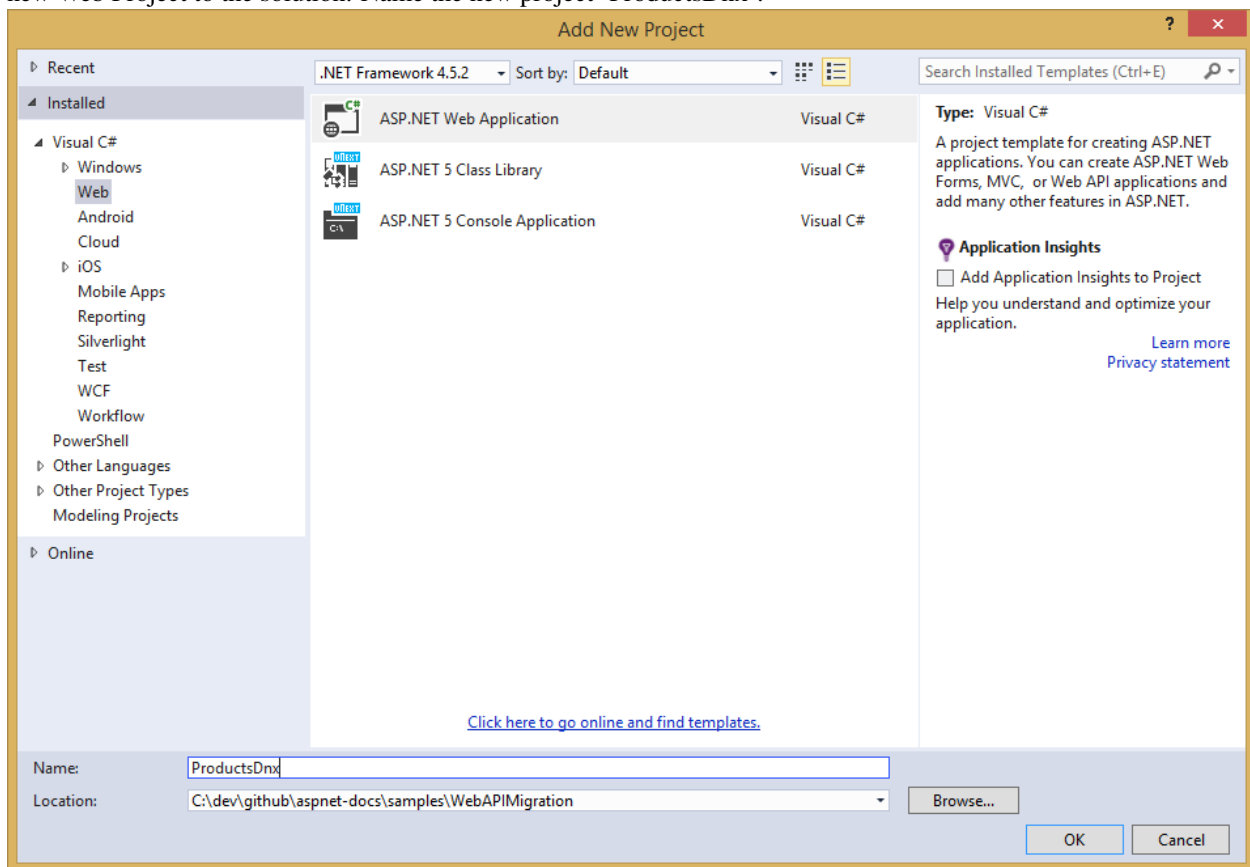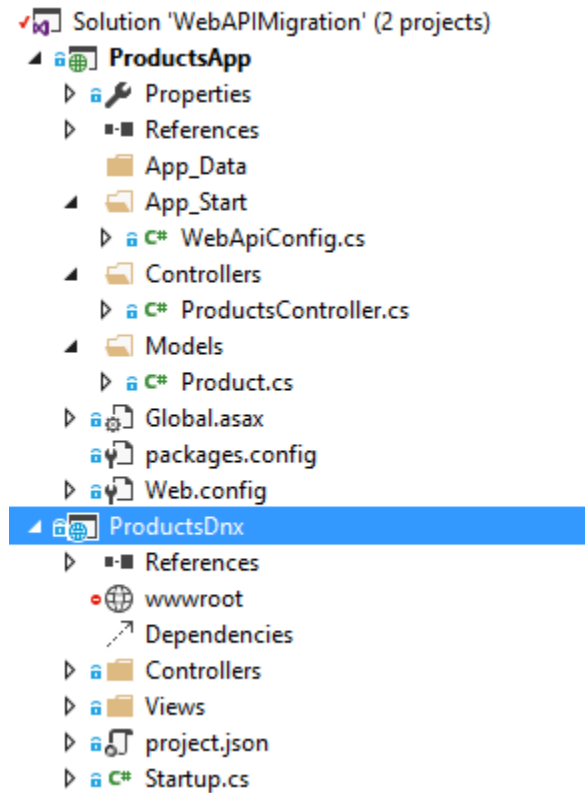

Next, choose the ASP.NET 5 Web API template project. We will migrate the ProductsApp contents to this new project.
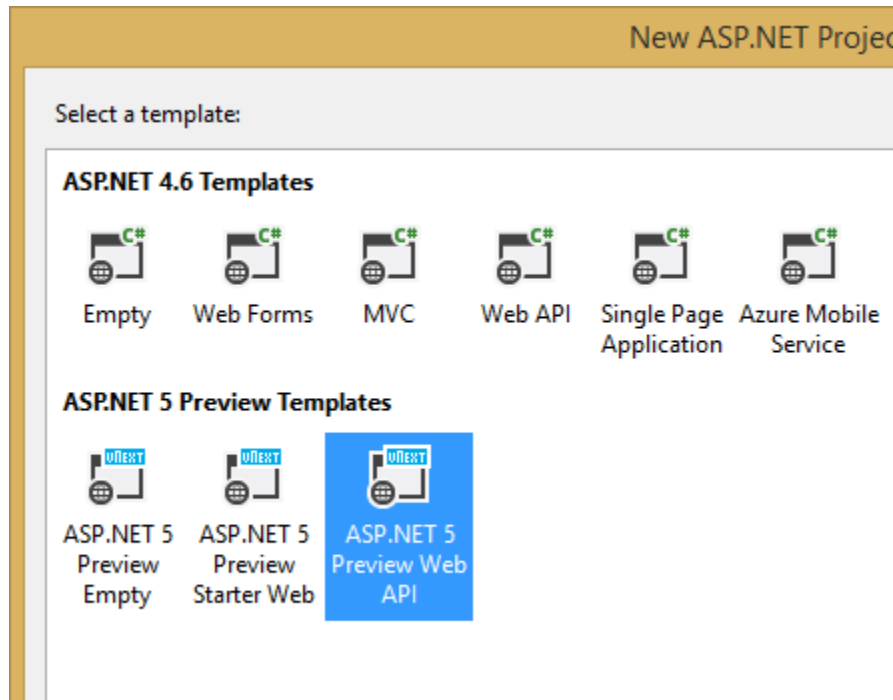
Delete the Project_Readme.html file from the new project. Your solution should now look like this:

8.3.3 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).

```csharp
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.Http;
using Microsoft.Framework.DependencyInjection;

namespace ProductsDnx
{
    public class Startup
    {
        public Startup(IHostingEnvironment env)
        {
        }

        // This method gets called by a runtime.
        // Use this method to add services to the container
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }

        // Configure is called after ConfigureServices is called.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseStaticFiles();
            // Add MVC to the request pipeline.
            app.UseMvc();
        }
    }
}
```

Assuming you want to use attribute routing in your project going forward, you don't need to do any additional configuration. You can simply apply the attributes as needed to your controllers and actions,, as is done in the sample ValuesController.cs class that is included in the Web API starter project:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNet.Mvc;

namespace ProductsDnx.Controllers
{
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        // GET: api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
```

```
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]* that are replaced at runtime with the name of the controller or action to which the attribute has been applied. This serves to reduce the number of magic strings in the project, and 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, but we can't yet test it because there are changes we must make before ProductsController will compile.

8.3.4 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 three 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 *IActionResult*

- 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.

8.3.5 Summary

Migrating a simple Web API 2 project to MVC 6 is fairly straightforward, thanks to the fact that Web API has been merged with MVC 6 in 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.

8.3.6 Related Resources

Create a Web API in MVC 6

8.4 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)**.

8.4.1 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
                        // 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.

8.4.2 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", n
            </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 { i
```

```
        </ul>
}
```

At this point, you should be able to refresh the site in your browser.

8.4.3 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