ASP.NET MVC 6 Documentation

Release 0.0.1

Microsoft

August 25, 2015

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.

CHAPTER 1

Overview of ASP.NET MVC

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

CHAPTER 2

Getting Started

2.1 Building Your First MVC 6 Application

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn 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:

- *Implement the other CRUD operations*
- *Next steps*

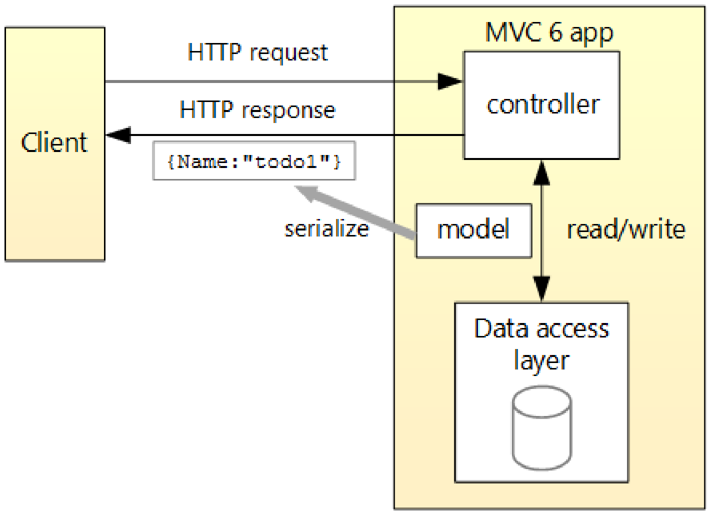
You can browse the source code for the sample app on GitHub.

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 Code using ASP.NET MVC 6.

2.2.2 Install Fiddler

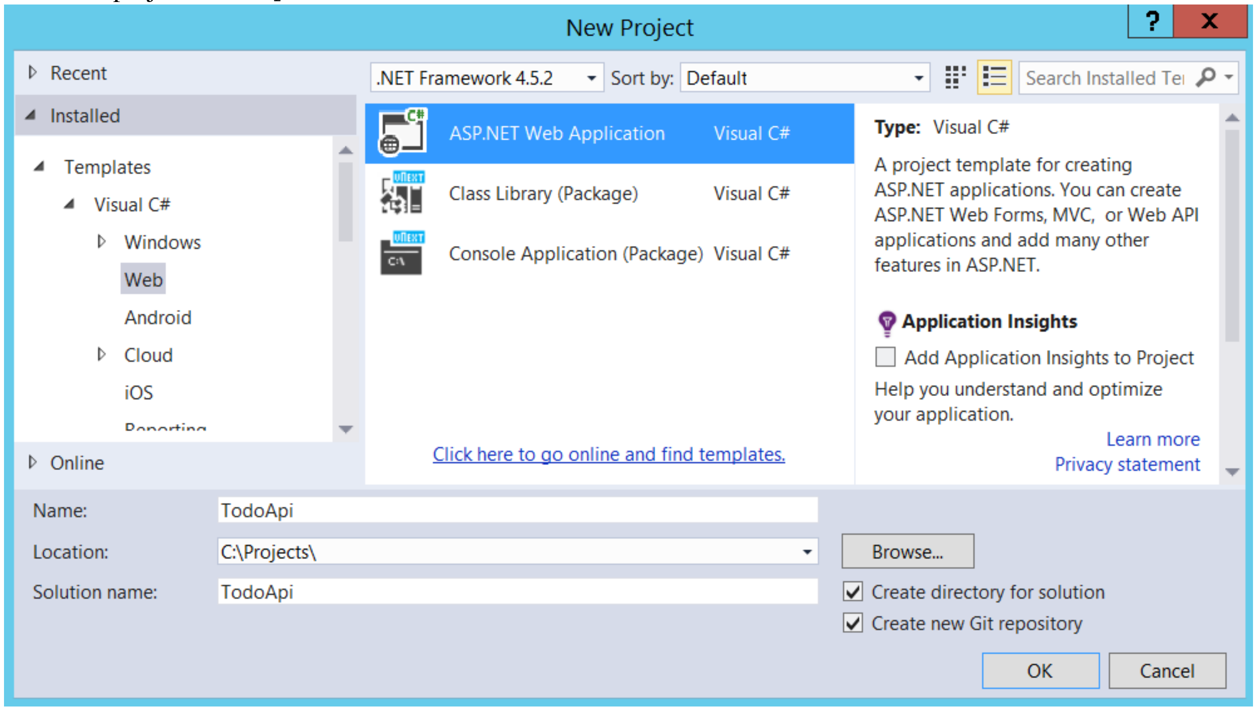
This step is optional.

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 which is a web debugging tool that lets you compose HTTP requests and view the raw HTTP responses. That lets use 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. It appears under **Installed** > **Templates** > **Visual C#** > **Web**. Name the project `TodoApi` and click **OK**.



Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Help you understand and optimize your application.

Name: TodoApi

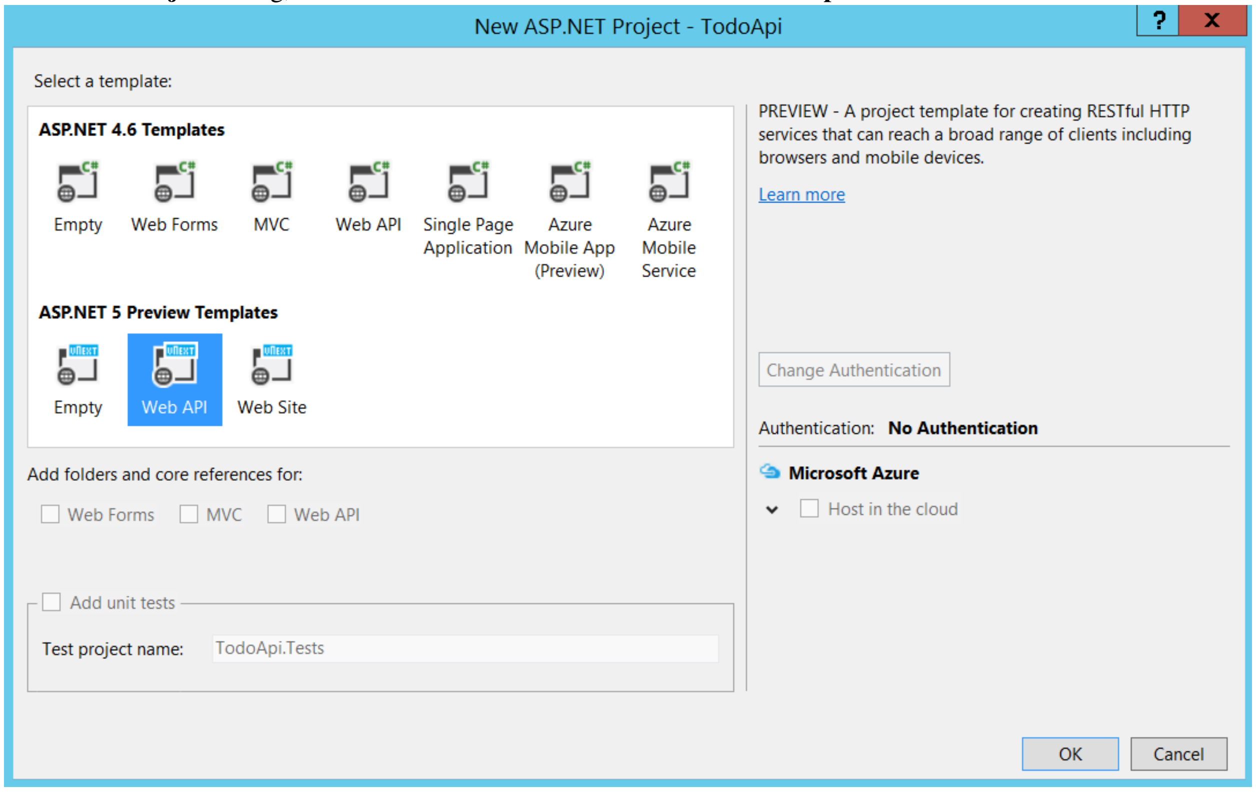
Location: C:\Projects\

Solution name: TodoApi

☑ Create directory for solution

☑ Create new Git repository

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



PREVIEW - A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

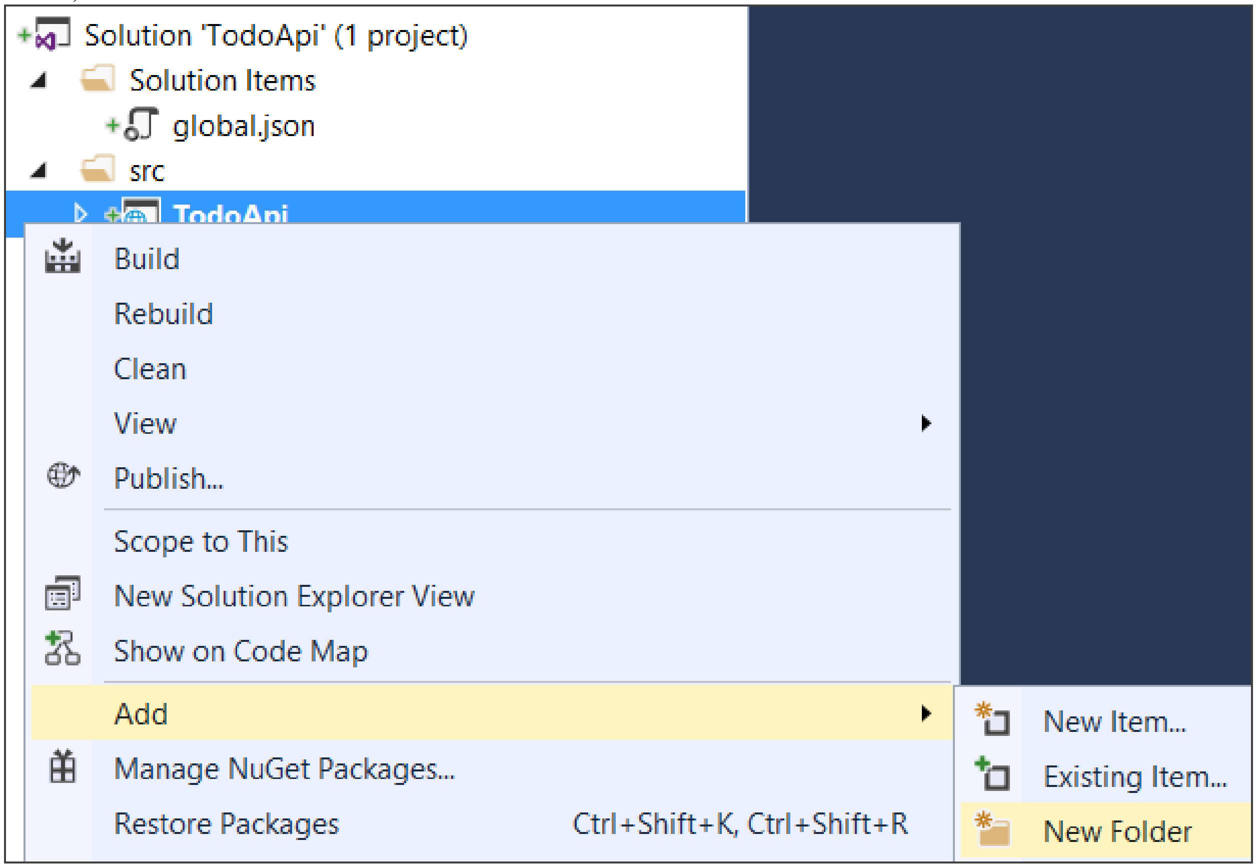
Authentication: **No Authentication**

Test project name: TodoApi.Tests

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.

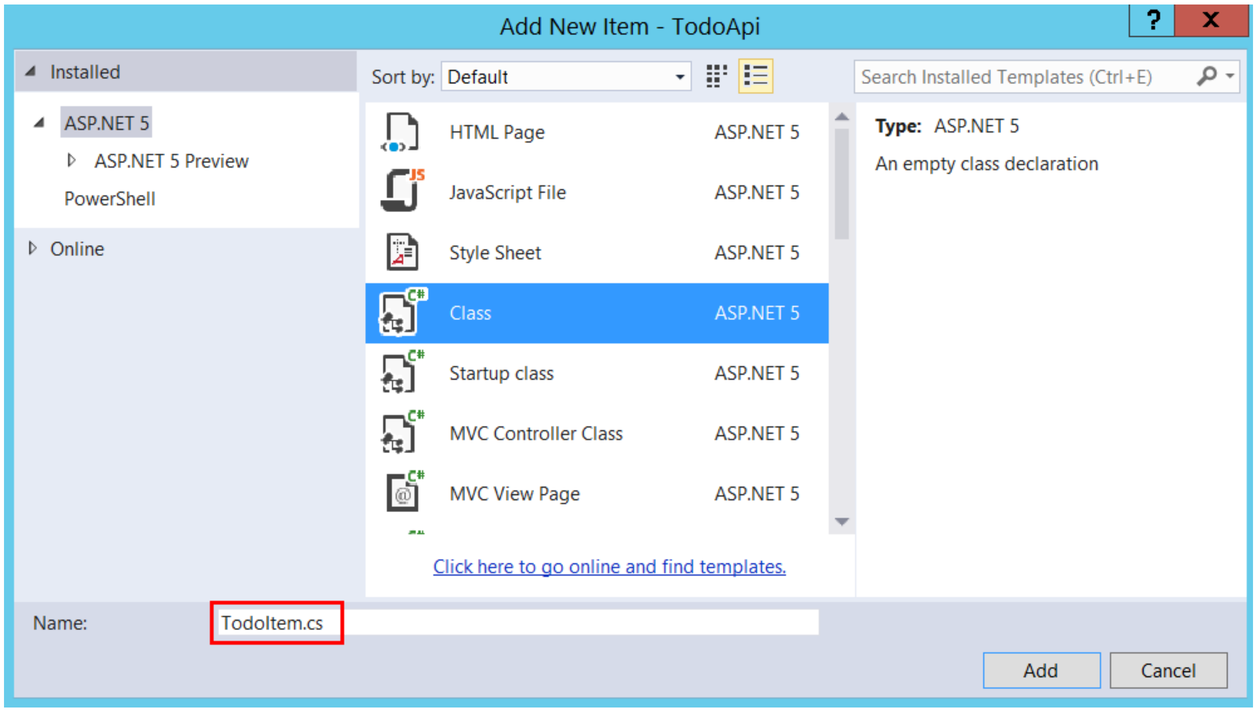
First, add a folder named "Models". In Solution Explorer, right-click the project. (The project appears under the "src" folder.) Select **Add** > **New Folder**. Name the folder *Models*.



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**.


Add New Item - TodoApi

Installed
ASP.NET 5
ASP.NET 5 Preview
PowerShell
Online

Sort by: Default

Search Installed Templates (Ctrl+E)

HTML Page	ASP.NET 5
JavaScript File	ASP.NET 5
Style Sheet	ASP.NET 5
Class	ASP.NET 5
Startup class	ASP.NET 5
MVC Controller Class	ASP.NET 5
MVC View Page	ASP.NET 5

Click here to go online and find templates.

Type: ASP.NET 5
An empty class declaration

Name: TodoItem.cs

Add Cancel

Replace the boilerplate 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.

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

```
using System.Collections.Generic;

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

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

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

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

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

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

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

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

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

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

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

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 System.Threading.Tasks;
```

In the ConfigureServices method, add the highlighted code:

```
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)
```

2.2.7 Add a controller

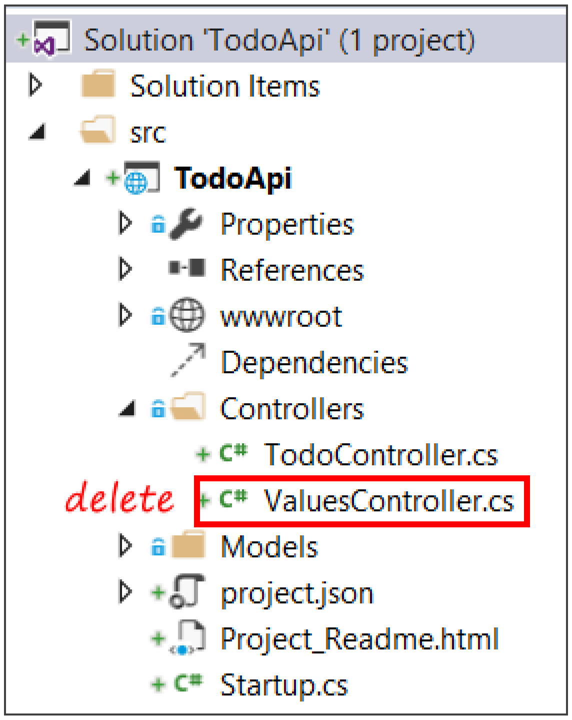
In Solution Explorer, right-click the *Controllers* folder. Select **Add** > **New Item**. In the **Add New Item** dialog, select the **Web API Controller Class** template. Name the class TodoController.

Replace the boilerplate code with the following:

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

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

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

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



2.2.8 Getting to-do items

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

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

[HttpGet("{id}", Name = "GetTodo")]
public IActionResult GetById(string id)
{
    var item = TodoItems.Find(id);
    if (item == null)
    {
        return HttpNotFound();
    }
    return new ObjectResult(item);
}
```

These methods implement the two GET methods:

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

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

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

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

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 the name of the controller class, minus the `Controller` suffix. For this sample, that's "todo".
- If the `[HttpGet]` attribute also has a template string, append that to the path.

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

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

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

To learn more about request routing in MVC 6, see Routing to Controller Actions.

Return values

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

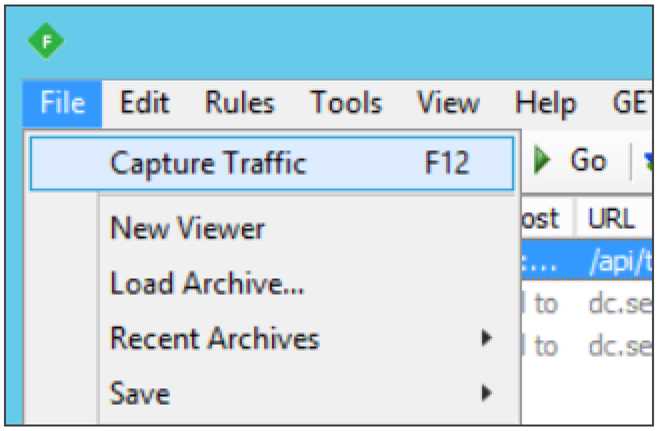
In contrast, the `GetById` method returns an `IActionResult` type, which represents a generic result type. That's because `GetById` has two expected return values:

- 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.

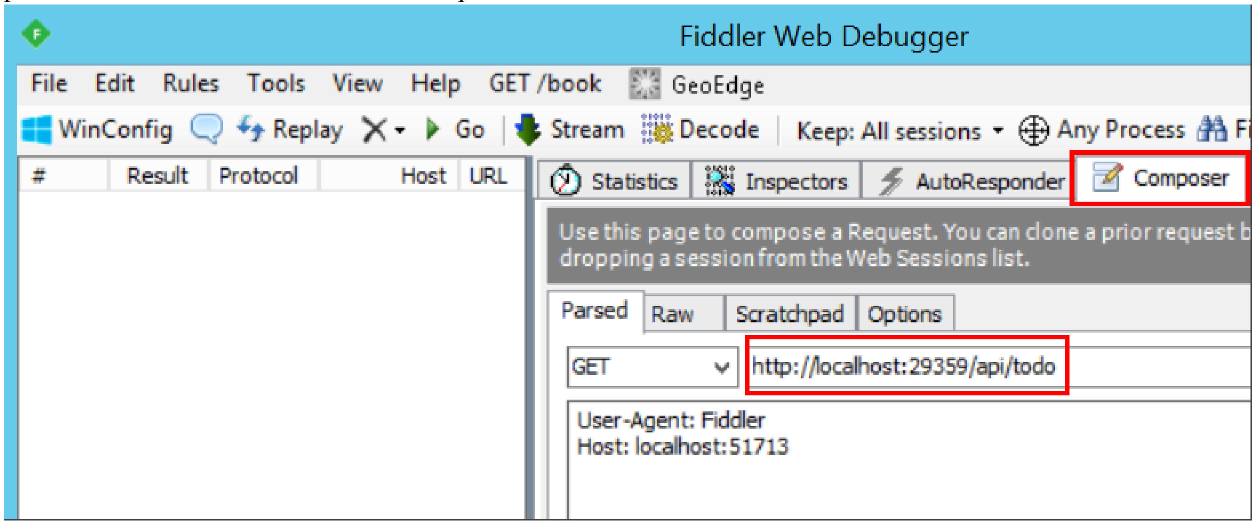
2.2.9 Use Fiddler to call the API

This step is optional, but it's useful to see the raw HTTP responses from the web API. In Visual Studio, press F5 to start debugging 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, 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.

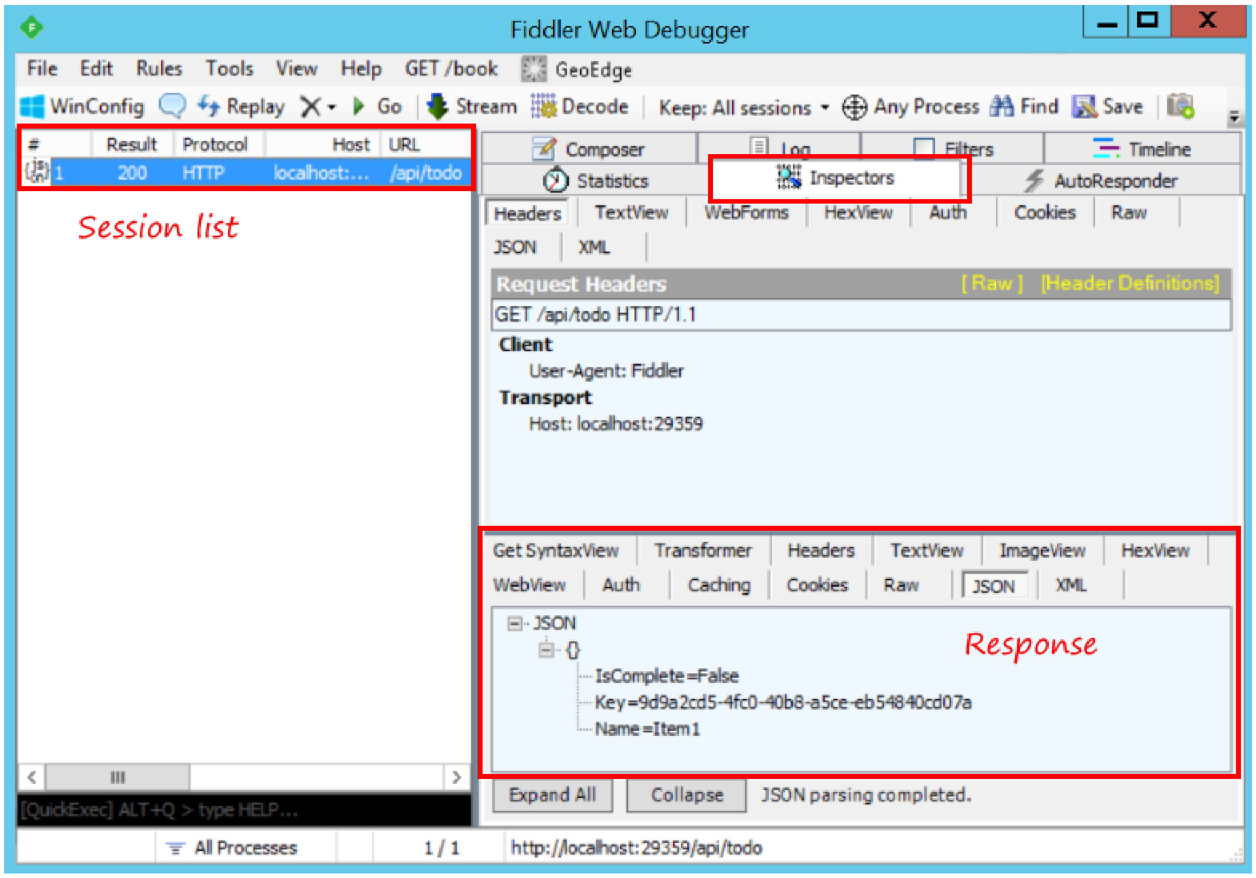


Use this page to compose a Request. You can clone a prior request b dropping a session from the Web Sessions list.

http://localhost:29359/api/todo

User-Agent: Fiddler
Host: localhost:51713

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.


Fiddler Web Debugger
File Edit Rules Tools View Help GET /book GeoEdge
WinConfig Replay Go Stream Decode Keep: All sessions Any Process Find Save
# Result Protocol Host URL
1 200 HTTP localhost:... /api/todo
Session list
Composer Log Filters Timeline
Statistics Inspectors AutoResponder
Headers TextView WebForms HexView Auth Cookies Raw
JSON XML
Get SyntaxView Transformer Headers TextView ImageView HexView
WebView Auth Caching Cookies Raw JSON XML
Response
Expand All Collapse
[QuickExec] ALT+Q > type HELP...
All Processes 1 / 1

Request Headers [Raw] [Header Definitions]

GET /api/todo HTTP/1.1

Client

User-Agent: Fiddler

Transport

Host: localhost:29359

```
JSON
  {}
    IsComplete=False
    Key=9d9a2cd5-4fc0-40b8-a5ce-eb54840cd07a
    Name=Item1
```

JSON parsing completed.

http://localhost:29359/api/todo

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

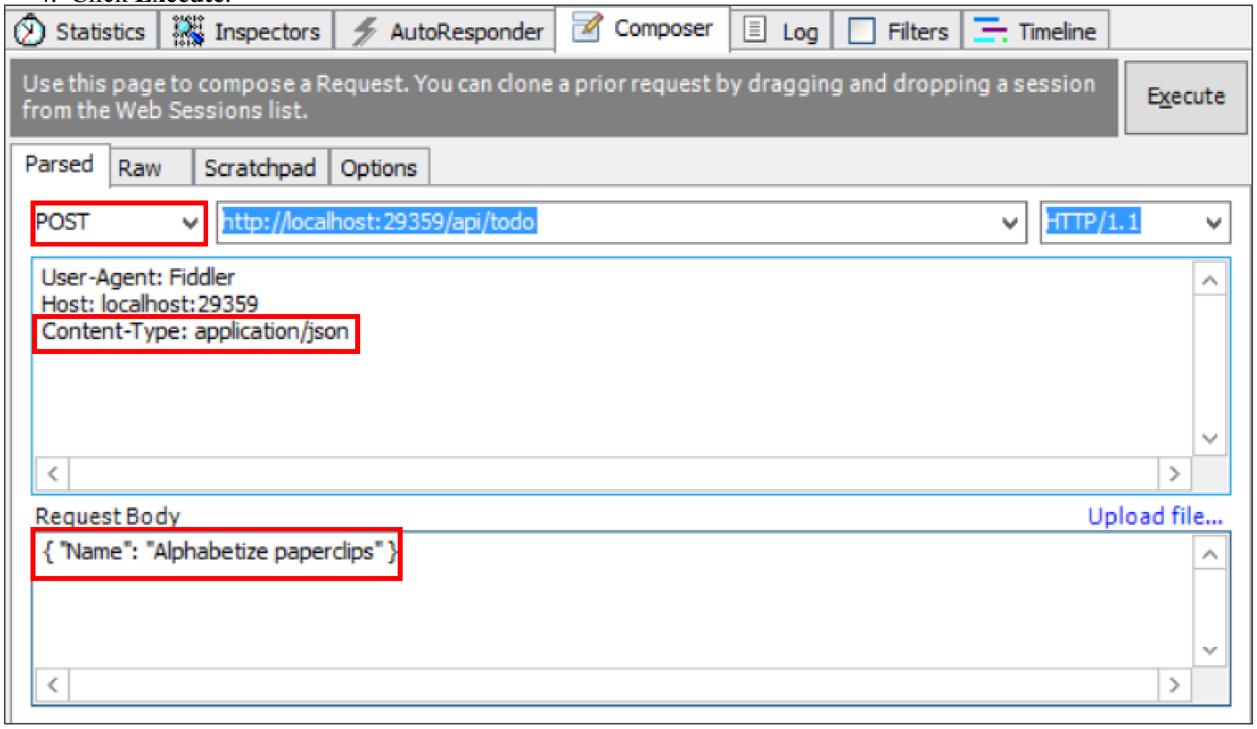
```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return HttpBadRequest();
    }
    TodoItems.Add(item);
    return CreatedAtRoute("GetTodo", new { controller = "Todo", id = item.Key }, item);
}
```

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, type the following: `{"Name":"<your to-do item>"}`
4. Click **Execute**.



Use this page to compose a Request. You can clone a prior request by dragging and dropping a session from the Web Sessions list.

POST http://localhost:29359/api/todo HTTP/1.1

```
User-Agent: Fiddler
Host: localhost:29359
Content-Type: application/json
```

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

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.

CHAPTER 3

Tutorials

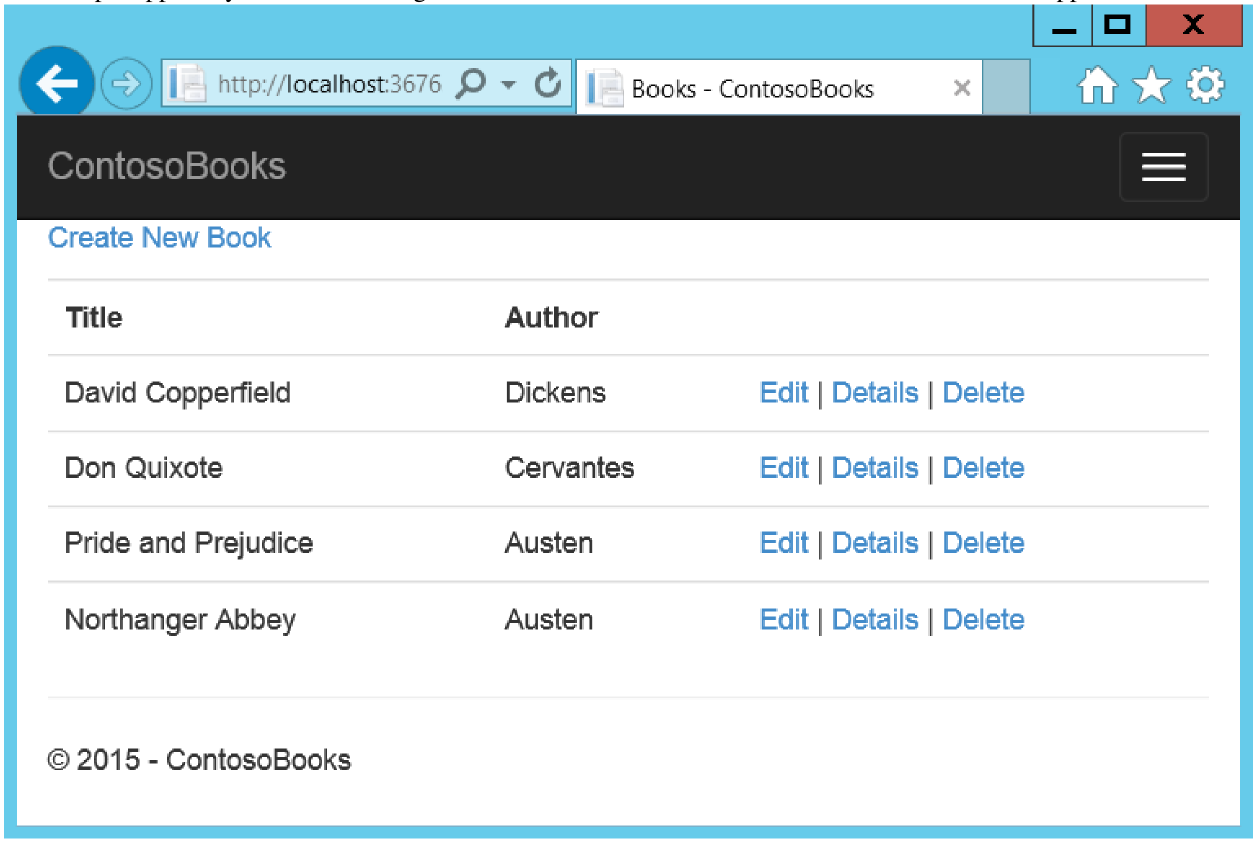
3.1 Get Started with Entity Framework 7 Code using ASP.NET MVC 6

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:



Title	Author	
David Copperfield	Dickens	Edit | Details | Delete
Don Quixote	Cervantes	Edit | Details | Delete
Pride and Prejudice	Austen	Edit | Details | Delete
Northanger Abbey	Austen	Edit | Details | Delete

© 2015 - ContosoBooks

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.)

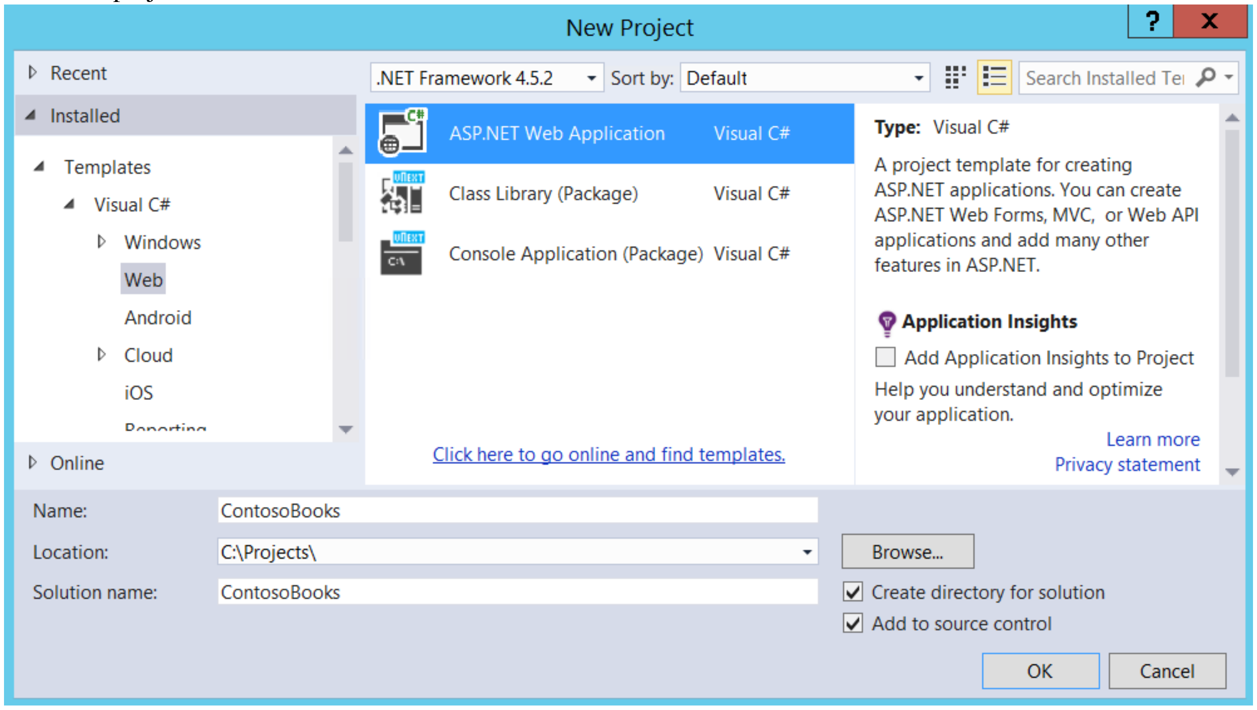
In this article:

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**.



Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Application Insights

☐ Add Application Insights to Project

Help you understand and optimize your application.

Name: ContosoBooks

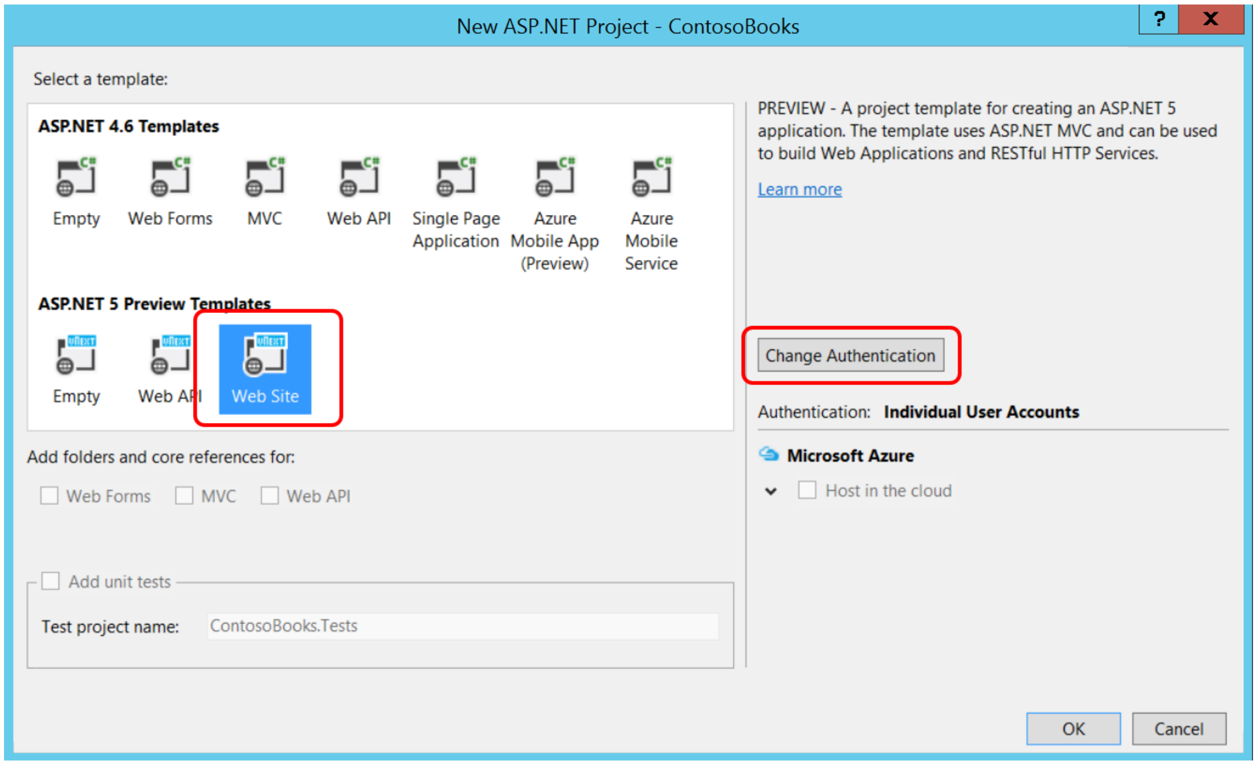
Location: C:\Projects\

Solution name: ContosoBooks

☑ Create directory for solution

☑ Add to source control

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

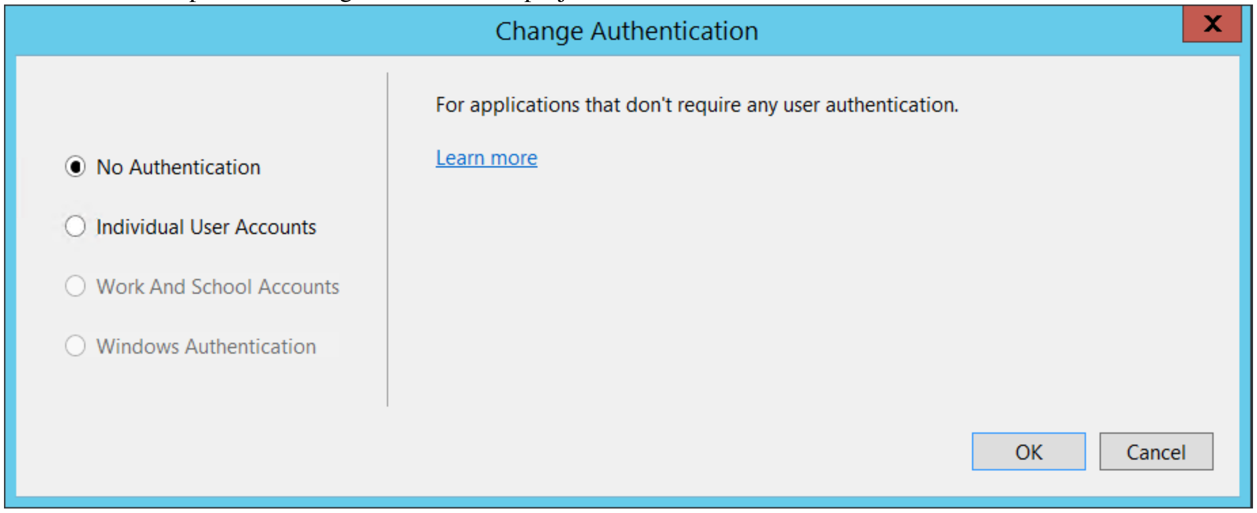


PREVIEW - A project template for creating an ASP.NET 5 application. The template uses ASP.NET MVC and can be used to build Web Applications and RESTful HTTP Services.

Authentication: **Individual User Accounts**

Test project name: ContosoBooks.Tests

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.



(●) No Authentication
( ) Individual User Accounts
( ) Work And School Accounts
( ) Windows Authentication

For applications that don't require any user authentication.

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>
```

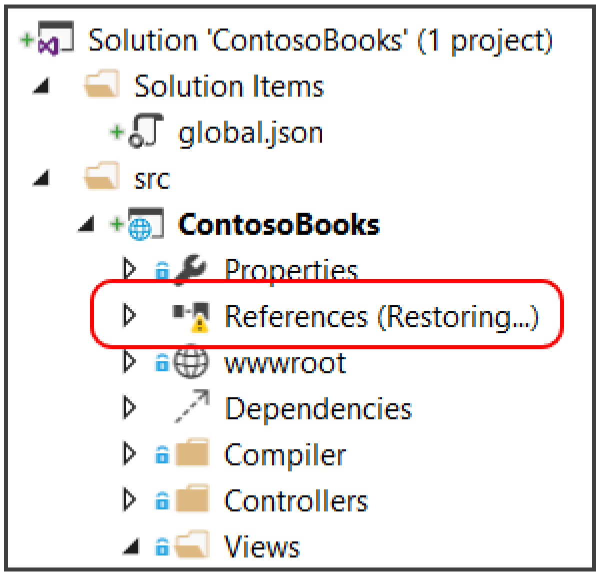
This adds a link to the Books page, which we haven't created yet. (That will come later in tutorial.)

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-beta5"
},
```

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

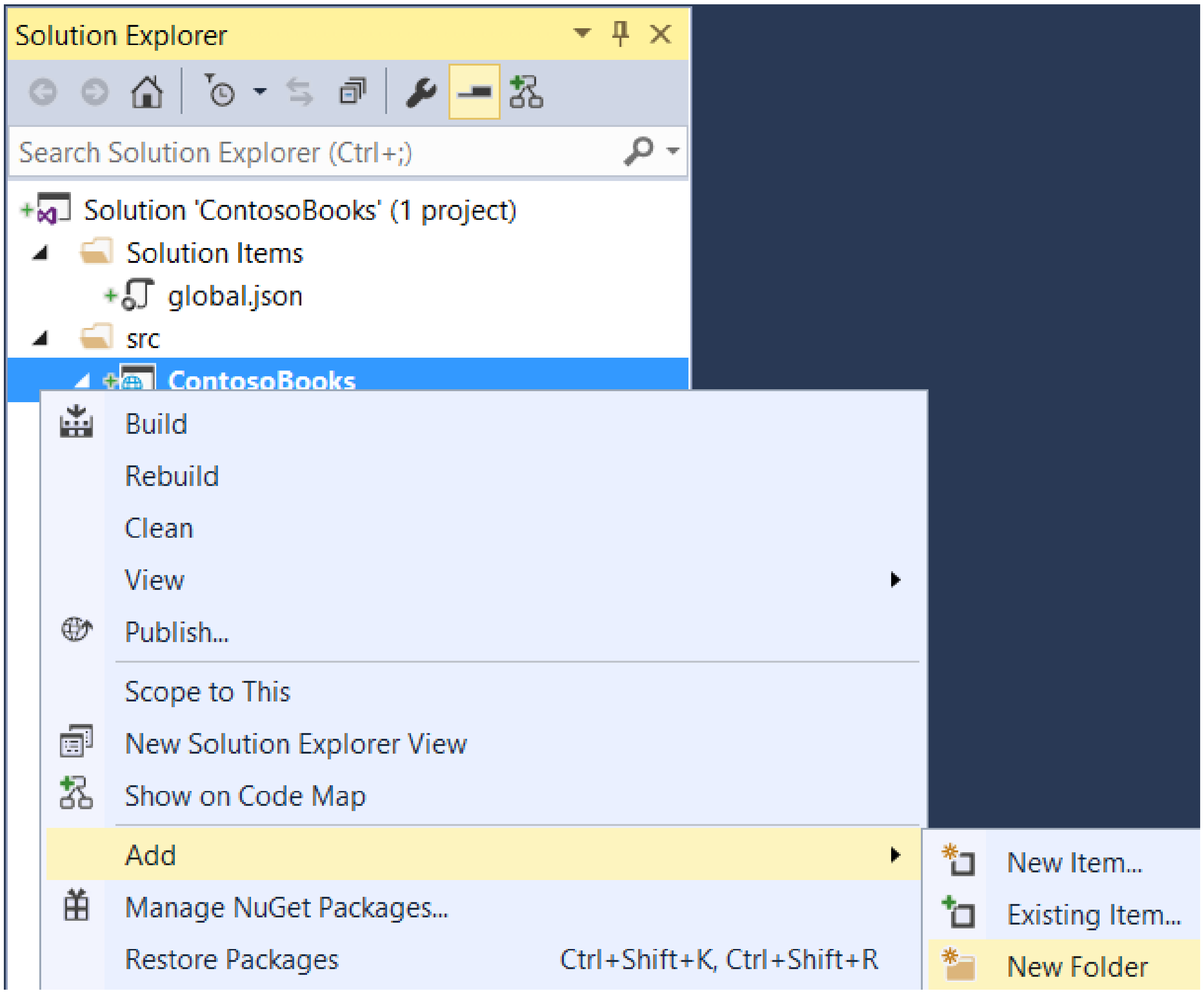


3.1.3 Create entity classes

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*.



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:

```
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;

namespace ContosoBooks.Models
{
    public static class SampleData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            var context = serviceProvider.GetService<BookContext>();
            if (context.Database.AsRelational().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.Get("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 ContosoBooks.Models;
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 Microsoft.Framework.Runtime;

namespace ContosoBooks
{
    public class Startup
    {
        public Startup(IHostingEnvironment env, IApplicationEnvironment appEnv)
        {
            // Setup configuration sources.
            var builder = new ConfigurationBuilder(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.Get("Data:ConnectionString"));
                });
        }

        // Configure is called after ConfigureServices is called.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory logge
        {
            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.UseErrorPage(ErrorPageOptions.ShowAll);
            }
            else
            {
                // Add Error handling middleware which catches all application specific errors and
                // send the request to the following path or controller action.
                app.UseErrorHandler("/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`.

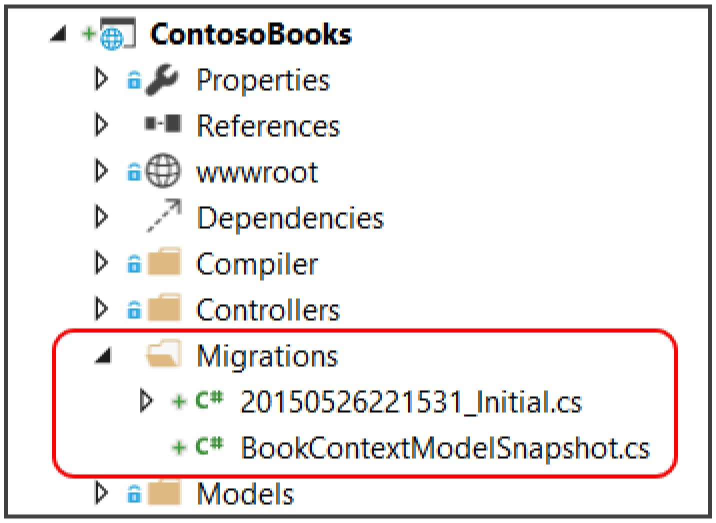
```
{
  "webroot": "wwwroot",
  "version": "1.0.0-*",

  "dependencies": {
    "Microsoft.AspNet.Diagnostics": "1.0.0-beta5",
    "Microsoft.AspNet.Mvc": "6.0.0-beta5",
    "Microsoft.AspNet.Mvc.TagHelpers": "6.0.0-beta5",
    "Microsoft.AspNet.Server.IIS": "1.0.0-beta5",
    "Microsoft.AspNet.Server.WebListener": "1.0.0-beta5",
    "Microsoft.AspNet.StaticFiles": "1.0.0-beta5",
    "Microsoft.AspNet.Tooling.Razor": "1.0.0-beta5",
    "Microsoft.Framework.Configuration.Json": "1.0.0-beta5",
    "Microsoft.Framework.Logging": "1.0.0-beta5",
    "Microsoft.Framework.Logging.Console": "1.0.0-beta5",
    "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0.0-beta5",
    "EntityFramework.SqlServer": "7.0.0-beta5",
    "EntityFramework.Commands": "7.0.0-beta5"
  },

  "commands": {
    "web": "Microsoft.AspNet.Hosting --config hosting.ini",
    "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-beta5
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-beta5` instructs the .NET Version Manager to add the 1.0.0-beta5 ASP.NET 5 runtime to the `PATH` environment variable for the current shell. For ASP.NET 5 Beta 5, the following is displayed:

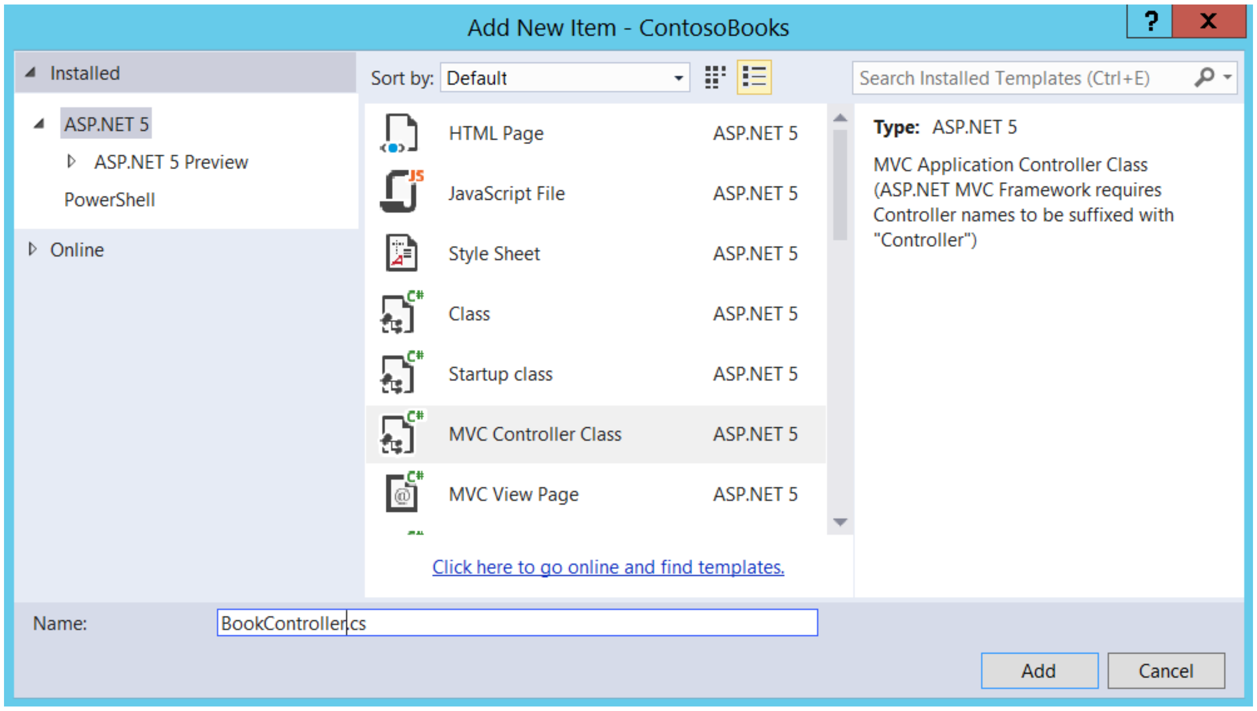
```
Adding C:\\Users\\<user>\\.dnx\\runtimes\\dnx-clr-win-x86.1.0.0-beta5\\bin to process PATH
```

- **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`.



Type: ASP.NET 5

MVC Application Controller Class (ASP.NET MVC Framework requires Controller names to be suffixed with "Controller")

Name: BookController.cs

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.Data.Entity.Storage;
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`:

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

private IEnumerable<SelectListItem> GetAuthorsListItems(int selected = -1)
{
    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*.

```
@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.

```
<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.

```
[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 (DataStoreException)
    {
        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:

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

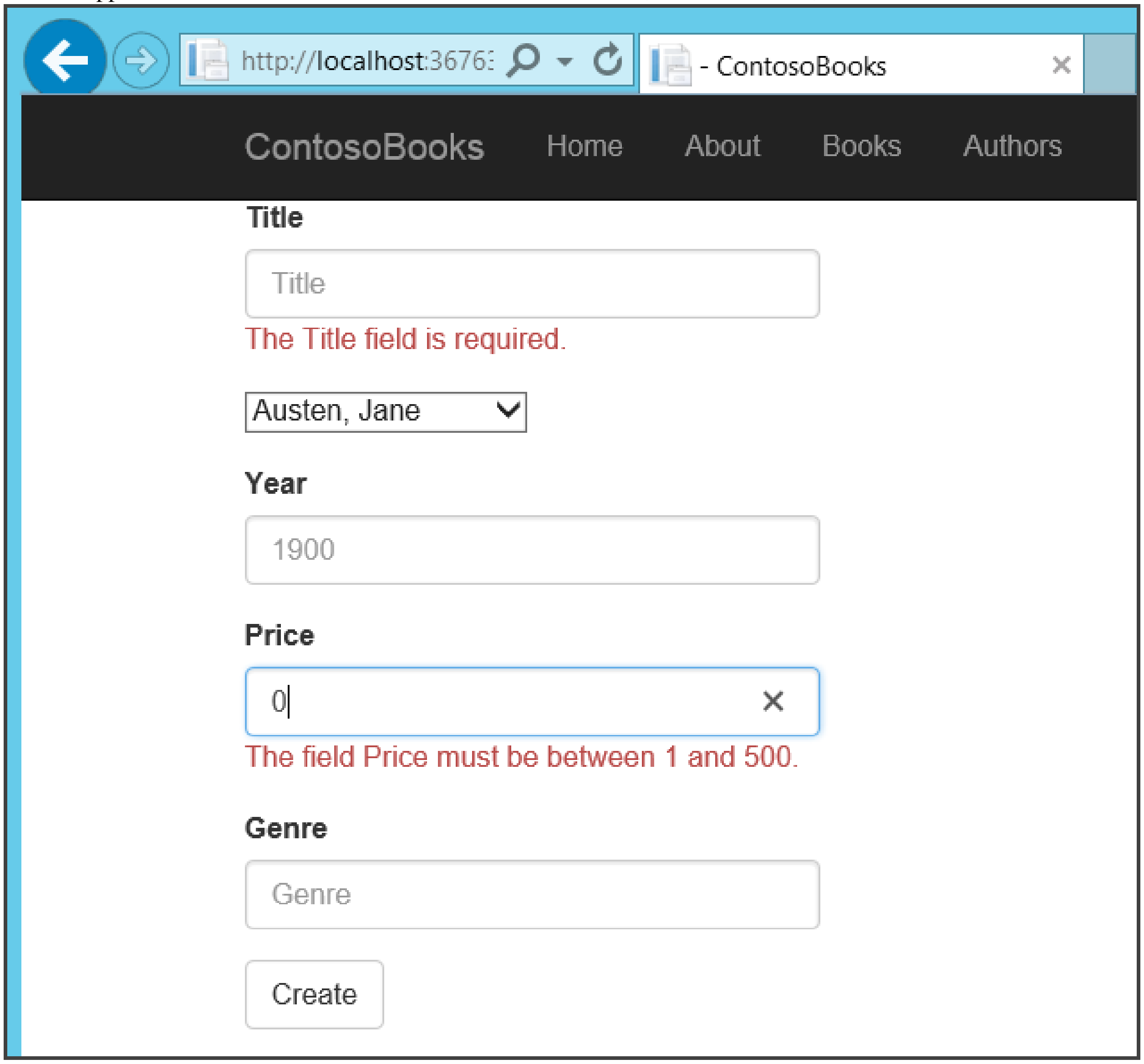
Add validation rules to the book model

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.

```
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**.



Title

Title

The Title field is required.

Austen, Jane

Year

1900

Price

0

The field Price must be between 1 and 500.

Genre

Genre

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 (DataStoreException)
    {
        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>
      <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`.

```
[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 (DataStoreException)
    {
        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: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

3.3 Creating Backend Services for Native Mobile Applications

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

CHAPTER 4

Models

4.1 Model Binding Request Data

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

4.2 Model Validation

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

4.3 Formatting

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

4.4 Custom Formatters

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

CHAPTER 5

Views

5.1 Razor Syntax

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.2 Dynamic vs Strongly Typed Views

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

Learn more about Dynamic vs Strongly Typed Views.

5.3 HTML Helpers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.4 Tag Helpers

5.4.1 Introduction to Tag Helpers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.4.2 Using Tag Helpers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.4.3 Authoring Tag Helpers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.4.4 Advanced Tag Helpers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.5 Partial Views

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.6 Dependency Injection in Views

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.7 View Components in MVC 6

By Rick Anderson

In this article:

5.7.1 Introducing view components

New to ASP.NET MVC 6, view components (VCs) are similar to partial views, but they are much more powerful. VCs include the same separation-of-concerns and testability benefits found between a controller and view. You can think of a VC as a mini-controller—it's responsible for rendering a chunk rather than a whole response. You can use VCs 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 VC 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 VC that gets and renders data depending on the user's claims. You can add this VC view to the layout page and have it get and render user-specific data throughout the whole application. ViewComponents don't use model binding, and only depend on the data you provide when calling into it.

A VC consists of two parts, the class (typically derived from `ViewComponent`) and the Razor view which calls methods in the VC class. Like the new ASP.NET controllers, a VC 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, VCs must be public, non-nested, non-abstract classes.

5.7.2 Download the ASP.NET 5 starter project

- Download and open the TodoList completed project.
- Open a command prompt in the project directory (*TodoList\src\TodoList*) and run the following commands:

```
dnvm use 1.0.0-beta5
dnx . ef migration apply
```

- **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-beta5` instructs the .NET Version Manager to add the 1.0.0-beta5 version of the ASP.NET 5 runtime to the `PATH` environment variable for the current shell. For ASP.NET 5 Beta 5, the following is displayed:

```
Adding C:\\Users\\<user>\\.dnx\\runtimes\\dnx-clr-win-x86.1.0.0-beta5\\bin to process PATH
```

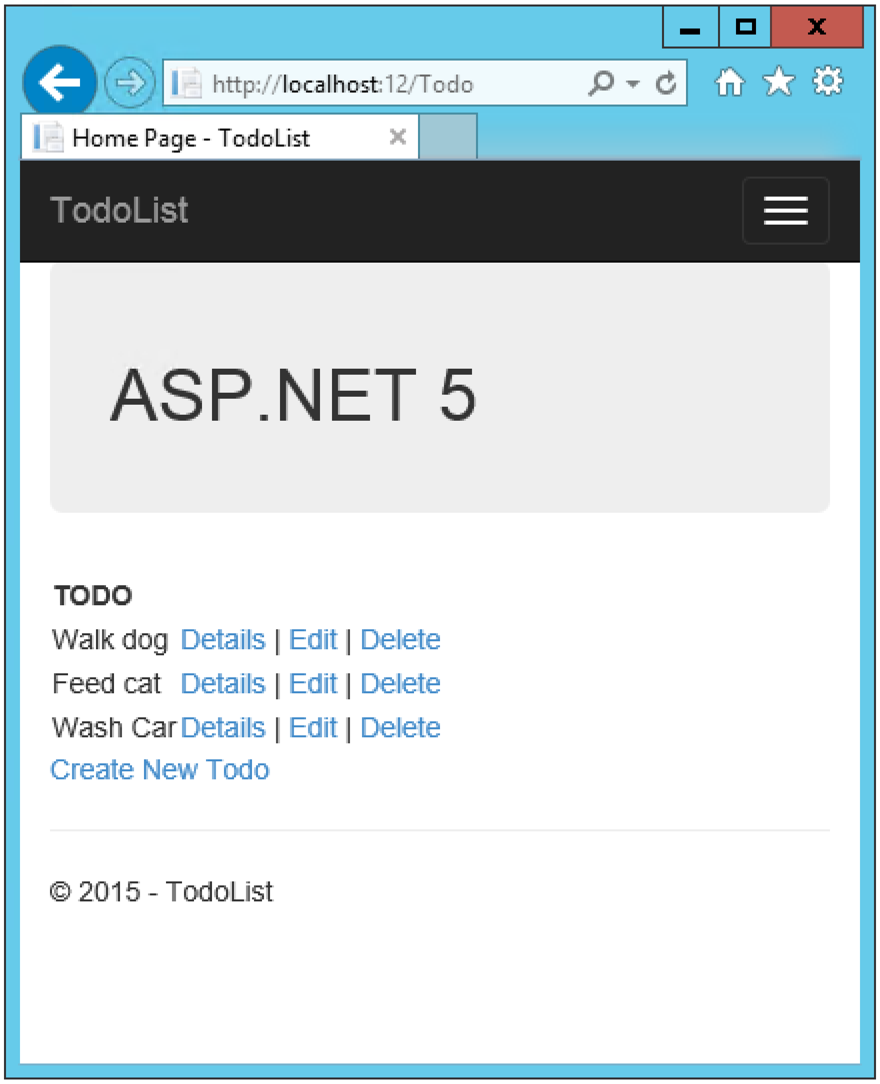
- DNX is the .NET Execution Environment.

- **dnx . ef migration add Initial** : DNX is the .NET Execution Environment. The `ef migration apply` command runs pending migration code.

5.7.3 Run the app

Note: With this version of Visual Studio 2015, you might have to exit Visual Studio after loading the project before you can run it.

Run the app, click the **Todo** link and create a couple *Todo* items. Make at least one of the *Todo* items with priority 1 and `IsDone` false (not checked).



TODO
Walk dog Details | Edit | Delete
Feed cat Details | Edit | Delete
Wash Car Details | Edit | Delete
Create New Todo

© 2015 - TodoList

5.7.4 Examine the view component class

- Examine the *srcTodoListViewComponentsPriorityListViewComponent.cs* file:

```
using System.Linq;
using Microsoft.AspNet.Mvc;
using TodoList.Models;

namespace TodoList.ViewComponents
{
  public class PriorityListViewComponent : ViewComponent
  {
        private readonly ApplicationDbContext db;

        public PriorityListViewComponent(ApplicationDbContext context)
        {
          db = context;
        }

        public IViewComponentResult Invoke(int maxPriority)
        {
          var items = db.TodoItems.Where(x => x.IsDone == false &&
                                  x.Priority <= maxPriority);

          return View(items);
        }
  }
}
```

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 VC. For example, we could have named the class `XYZ`, and applied the `ViewComponent` attribute:

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

The `[ViewComponent]` attribute above tells the view component selector to use the name `PriorityList` when looking for the views associated with the component, and to use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.

- The component uses 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`.

5.7.5 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 VCs 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 *ViewCom-*

ponent 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.

```
@model IEnumerable<TodoList.Models.TodoItem>

<h3>Priority Items</h3>
<ul>
        @foreach (var todo in Model)
        {
                <li>@todo.Title</li>
        }
</ul>
```

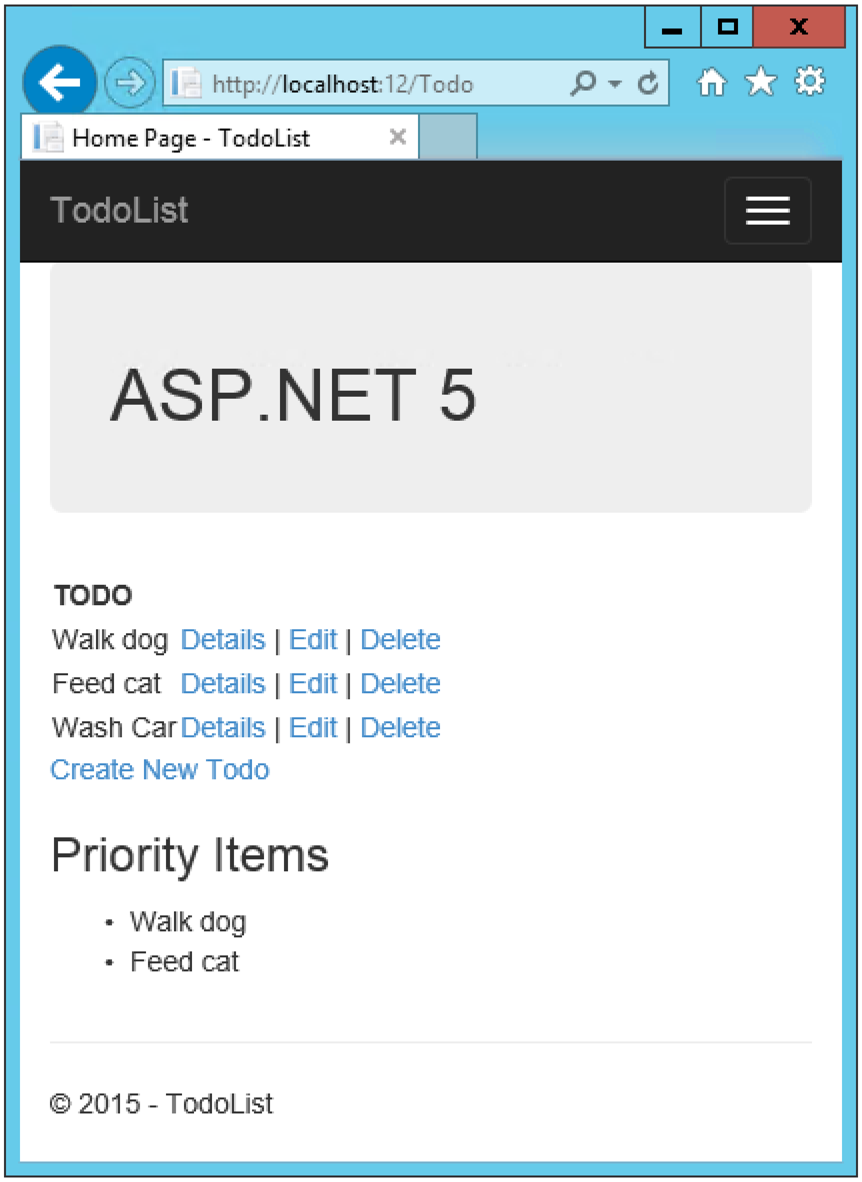
The Razor view takes a list of `TodoItems` and displays them. If the VC `invoke` method doesn't pass the name of the view (as in our sample), *Default* is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view.

4. Add a `div` containing a call to the priority list component to the bottom of the *views\todo\index.cshtml* file:

```
            @* Markup removed for brevity *@
            <div>@Html.ActionLink("Create New Todo", "Create", "Todo") </div>
      </div>

      <div class="col-md-4">
            @Component.Invoke("PriorityList", 1)
      </div>
    </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)



ASP.NET 5

TODO

Walk dog Details | Edit | Delete

Feed cat Details | Edit | Delete

Wash Car Details | Edit | Delete

Create New Todo

Priority Items

- Walk dog
- Feed cat

© 2015 - TodoList

5.7.6 Add InvokeAsync to the priority 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.

```
using System.Threading.Tasks;

public class PriorityListViewComponent : ViewComponent
{
   private readonly ApplicationDbContext db;

   public PriorityListViewComponent(ApplicationDbContext context)
   {
```

```
                db = context;
        }

        // Synchronous Invoke removed.

        public async Task<IViewComponentResult> InvokeAsync(int maxPriority, bool isDone)
        {
                var items = await GetItemsAsync(maxPriority, isDone);
                return View(items);
        }

        private Task<IQueryable<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
                return Task.FromResult(GetItems(maxPriority, isDone));
        }
        private IQueryable<TodoItem> GetItems(int maxPriority, bool isDone)
        {
                var items = db.TodoItems.Where(x => x.IsDone == isDone &&
                                     x.Priority <= maxPriority);

                string msg = "Priority <= " + maxPriority.ToString() +
                                        " && isDone == " + isDone.ToString();
                ViewBag.PriorityMessage = msg;

                return items;
        }
    }
```

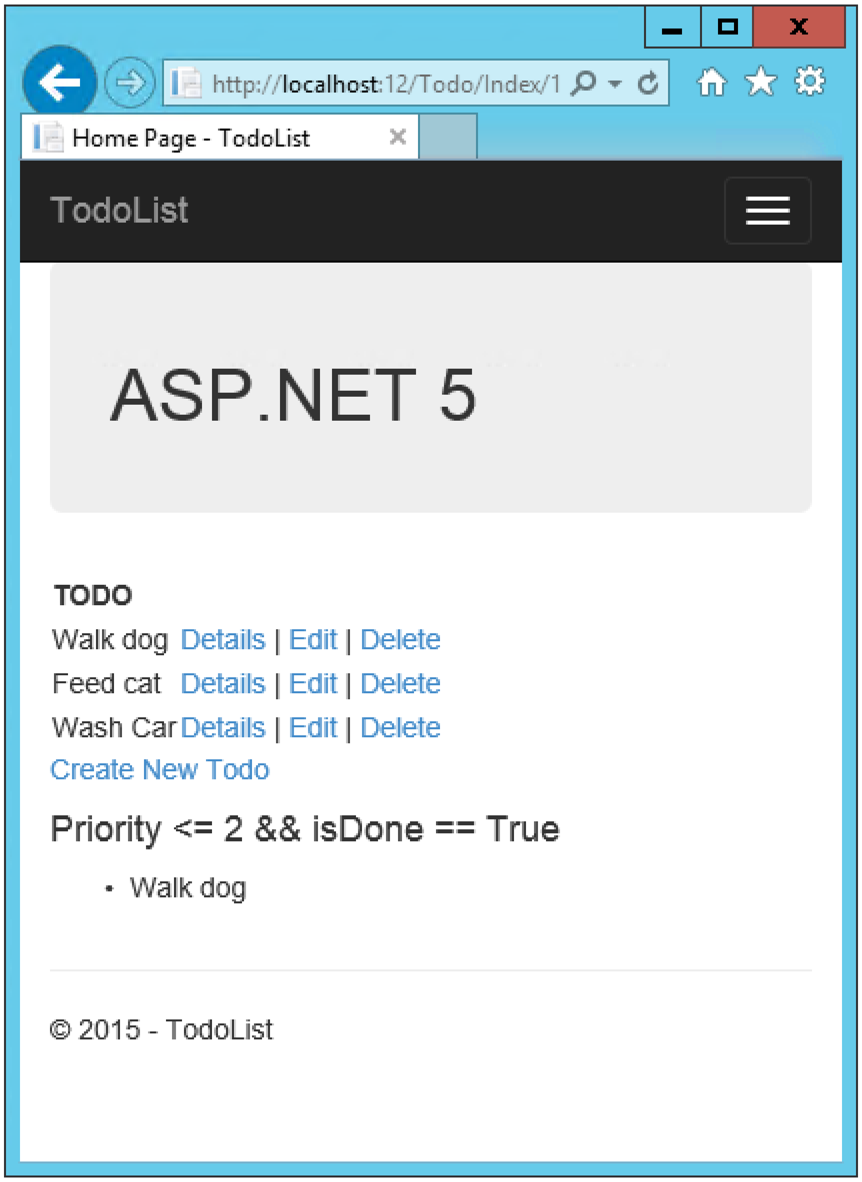
Update the VC Razor view (*TodoList\src\TodoList\Views\ToDo\Components\PriorityList\Default.cshtml*) to show the priority message :

```
    @model IEnumerable<TodoList.Models.TodoItem>

            <h4>@ViewBag.PriorityMessage</h4>
            <ul>
                    @foreach (var todo in Model)
                    {
                            <li>@todo.Title</li>
                    }
            </ul>
```

Finally, update the *views\todo\index.cshtml* view:

```
            @* Markup removed for brevity. *@

            <div class="col-md-4">
                    @await Component.InvokeAsync("PriorityList", 2, true)
            </div>
    </div>
```

The following image reflects the changes we made to the priority VC and Index view:



ASP.NET 5

TODO

Walk dog Details | Edit | Delete
Feed cat Details | Edit | Delete
Wash Car Details | Edit | Delete
Create New Todo

Priority <= 2 && isDone == True

- Walk dog

© 2015 - TodoList

5.7.7 Specifying a view name

A complex VC 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.

```
    public async Task<IViewComponentResult> InvokeAsync(int maxPriority, bool isDone)
    {
            string MyView = "Default";
            // If asking for all completed tasks, render with the "PVC" view.
            if (maxPriority > 3 && isDone == true)
            {
                    MyView = "PVC";
            }
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(MyView, items);
    }
```

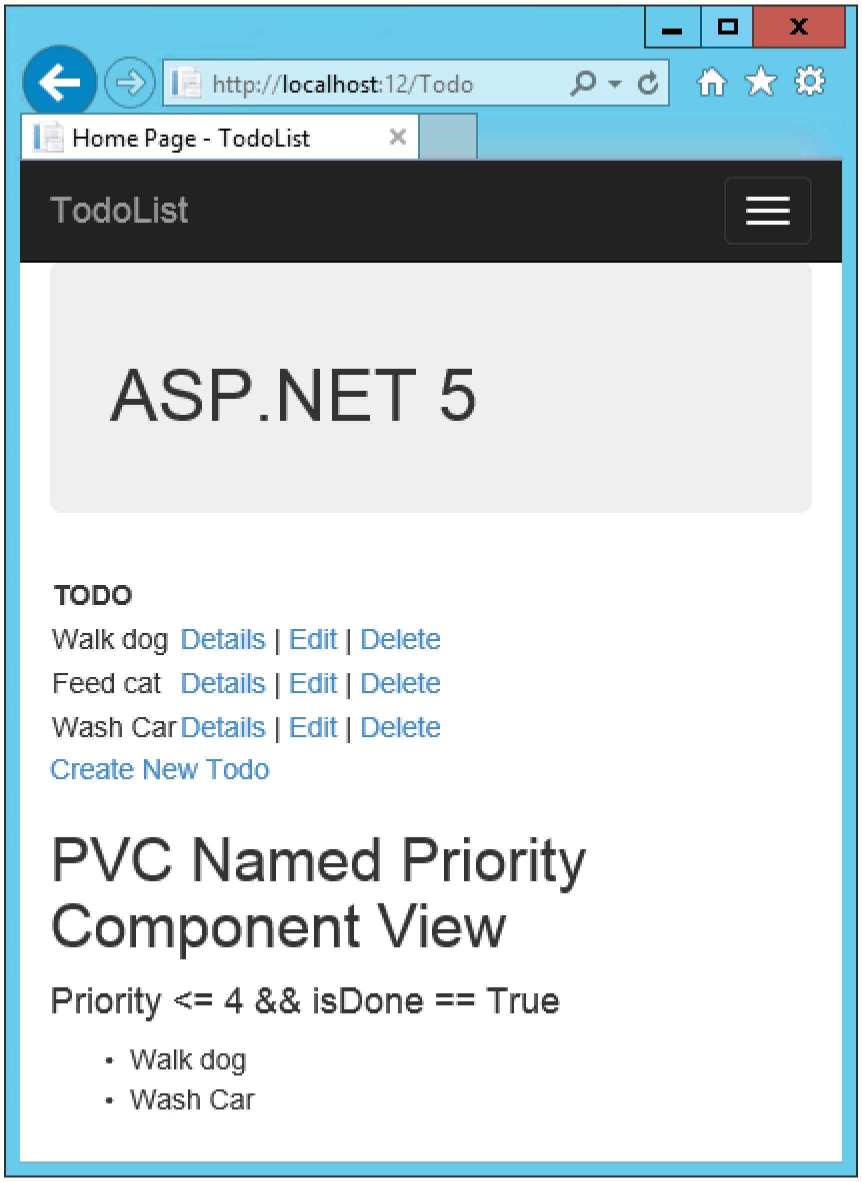
Examine the *Views\Todo\Components\PriorityList\PVC.cshtml* view. I changed the PVC view to verify it's being used:

```
    @model IEnumerable<TodoList.Models.TodoItem>

    <h2> PVC Named Priority Component View</h2>
    <h4>@ViewBag.PriorityMessage</h4>
    <ul>
            @foreach (var todo in Model)
            {
                    <li>@todo.Title</li>
            }
    </ul>
```

Finally, update *Views\TodoIndex.cshtml*

```
@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.



ASP.NET 5

TODO

Walk dog Details | Edit | Delete
Feed cat Details | Edit | Delete
Wash Car Details | Edit | Delete
Create New Todo

PVC Named Priority Component View

Priority <= 4 && isDone == True

- Walk dog
- Wash Car

5.7.8 Injecting a service into a view

ASP.NET MVC 6 now supports injection into a view from a class. Unlike a VC class, there are no restrictions other than the class must be must be public, non-nested and non-abstract. 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.

The StatisticsService class:

```
using System.Linq;
using System.Threading.Tasks;
using TodoList.Models;

namespace TodoList.Services
{
   public class StatisticsService
   {
          private readonly ApplicationDbContext db;

          public StatisticsService(ApplicationDbContext context)
          {
                 db = context;
          }

          public async Task<int> GetCount()
          {
                 return await Task.FromResult(db.TodoItems.Count());
          }

          public async Task<int> GetCompletedCount()
          {
                 return await Task.FromResult(
                         db.TodoItems.Count(x => x.IsDone == true));
          }

          public async Task<double> GetAveragePriority()
          {
                 if (db.TodoItems.Count() == 0)
                 {
                        return 0.0;
                 }

                 return await Task.FromResult(
                         db.TodoItems.Average(x =>x.Priority));
          }
   }
}
```

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

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

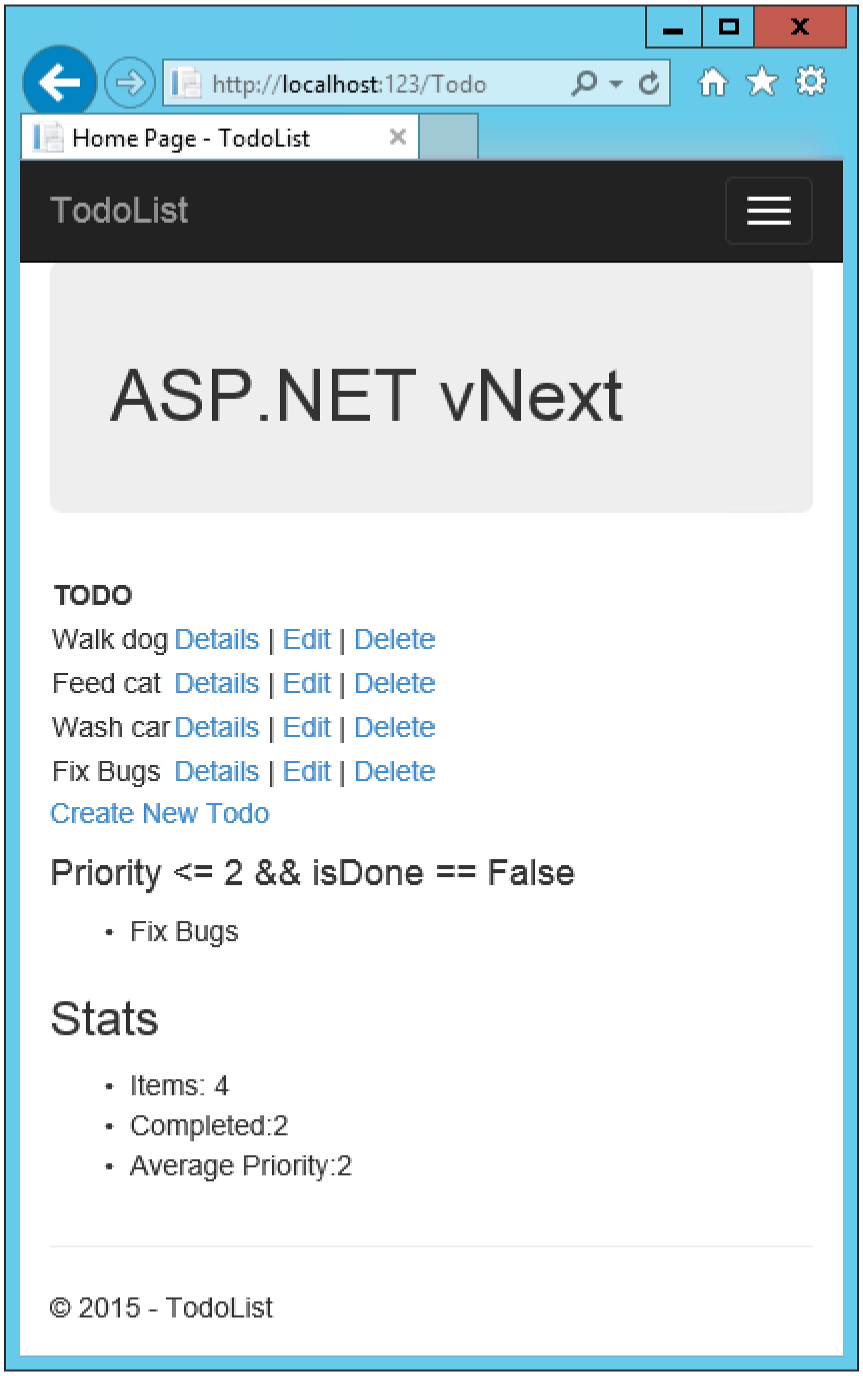
Add markup calling the StatisticsService to the end of the file:

```
      @* Markup removed for brevity *@
               <div>@Html.ActionLink("Create New Todo", "Create", "Todo") </div>
        </div>
        <div class="col-md-4">
               @await Component.InvokeAsync("PriorityList", 4, true)
               <h3>Stats</h3>
               <ul>
                      <li>Items: @await Statistics.GetCount()</li>
                      <li>Completed:@await Statistics.GetCompletedCount()</li>
                      <li>Average Priority:@await Statistics.GetAveragePriority()</li>
               </ul>
        </div>
     </div>
```

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

```
     public void ConfigureServices(IServiceCollection services)
     {
        // Add Application settings to the services container.
        services.Configure<AppSettings>(Configuration.GetConfigurationSection("AppSettings"));
        // Code removed for brevity.
        // Add MVC services to the services container.
        services.AddMvc();
        services.AddTransient<TodoList.Services.StatisticsService>();
    }
```

The statistics are displayed:



ASP.NET vNext

TODO

Walk dog Details | Edit | Delete

Feed cat Details | Edit | Delete

Wash car Details | Edit | Delete

Fix Bugs Details | Edit | Delete

Create New Todo

Priority <= 2 && isDone == False

- Fix Bugs

Stats

- Items: 4
- Completed:2
- Average Priority:2

© 2015 - TodoList

5.7.9 Addition Resources

- Understanding ASP.NET 5 Web Apps
- Introducing .NET Core

5.8 Creating a Custom View Engine

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

5.9 Building Mobile Specific Views

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

CHAPTER 6

Controllers

6.1 Actions and Action Results

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.2 Routing to Controller Actions

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.3 Error Handling

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.4 Filters

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.5 Dependency Injection and Controllers

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.6 Testing Controller Logic

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.7 Areas

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

6.8 Working with the Application Model

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

CHAPTER 7

Security

7.1 Authorization Filters

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

7.2 Enforcing SSL

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

7.3 Anti-Request Forgery

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn 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": "1.0.0-beta4"
  },
```

7.4.2 Configure CORS

To configure CORS, call `ConfigureCors` in the `ConfigureServices` method of your `Startup` class, as shown here:

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

CHAPTER 8

Migration

8.1 Migrating From ASP.NET MVC 5 to MVC 6

By Steve Smith

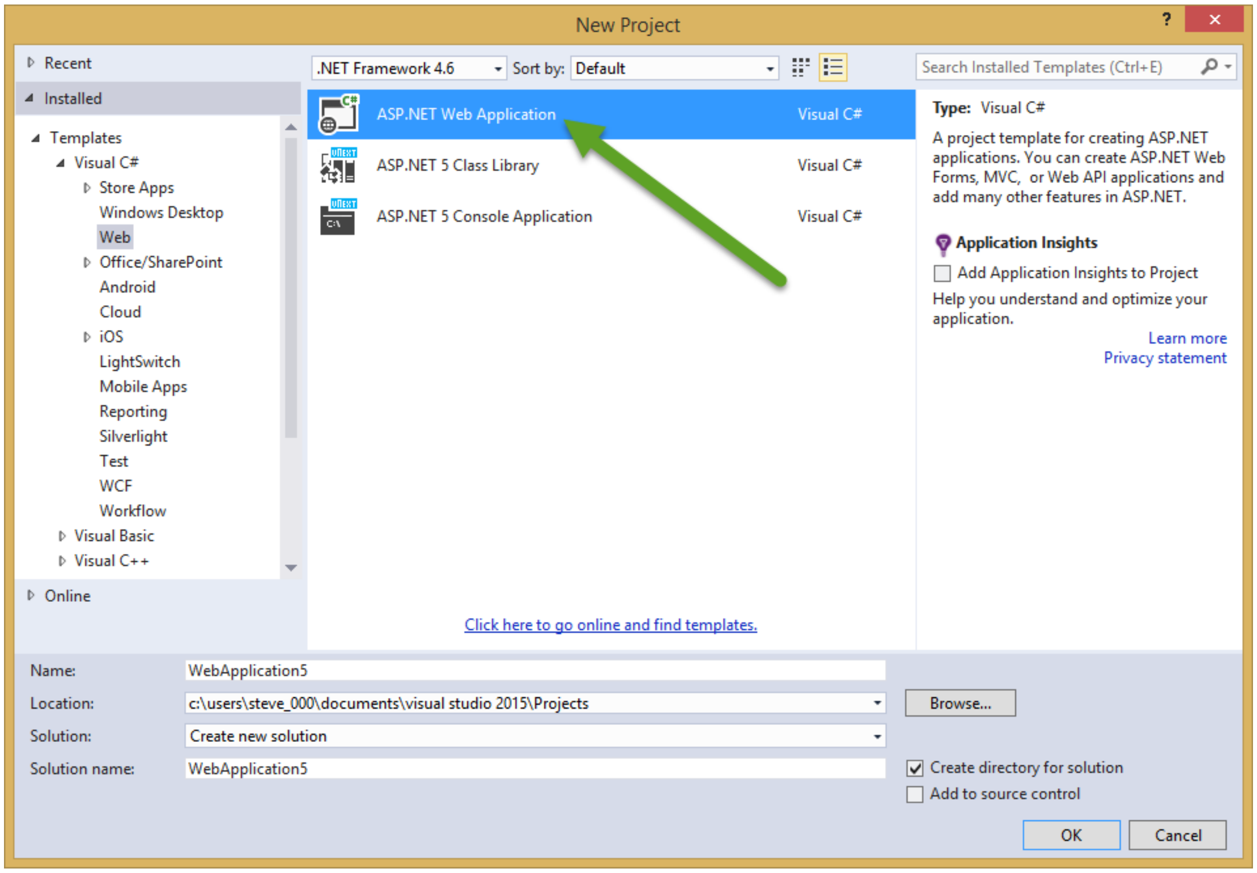
Migrating from ASP.NET MVC 5 to ASP.NET 5 and MVC 6 requires a few steps to complete, since ASP.NET 5 introduces a number of new concepts. In this article you will learn how to migrate from the ASP.NET MVC 5 default project template to ASP.NET MVC 6, including initial setup, basic controllers and views,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

In this article:

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.



Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Application Insights

☐ Add Application Insights to Project

Help you understand and optimize your application.

Name: WebApplication5

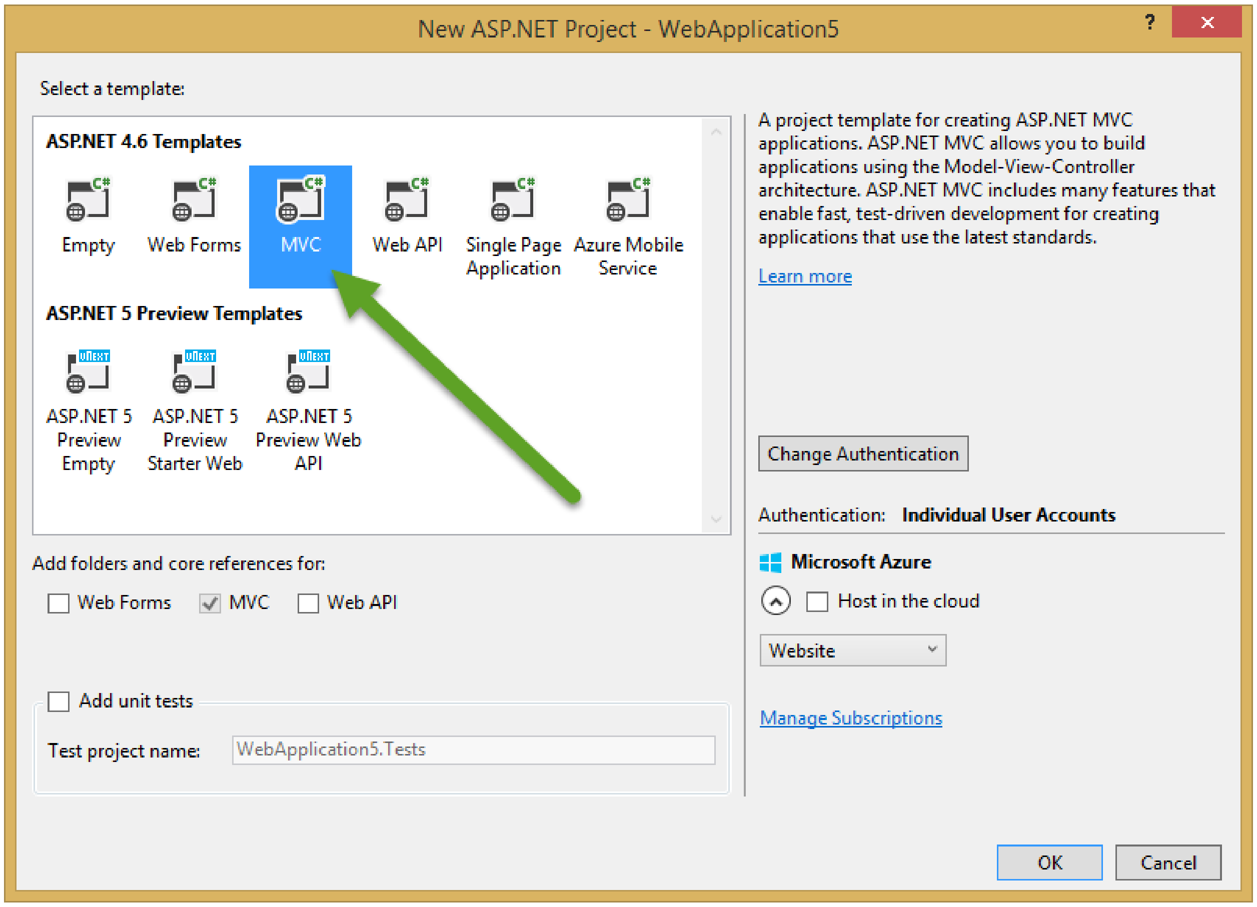
Location: c:\users\steve_000\documents\visual studio 2015\Projects

Solution: Create new solution

Solution name: WebApplication5

☑ Create directory for solution

☐ Add to source control


New ASP.NET Project - WebApplication5

Select a template:

ASP.NET 4.6 Templates: Empty, Web Forms, MVC (selected), Web API, Single Page Application, Azure Mobile Service

ASP.NET 5 Preview Templates: ASP.NET 5 Preview Empty, ASP.NET 5 Preview Starter Web, ASP.NET 5 Preview Web API

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Change Authentication

Authentication: **Individual User Accounts**

Microsoft Azure

[ ] Host in the cloud

Website

Manage Subscriptions

Add folders and core references for:

[ ] Web Forms [✓ MVC] [ ] Web API

[ ] Add unit tests

Test project name: WebApplication5.Tests

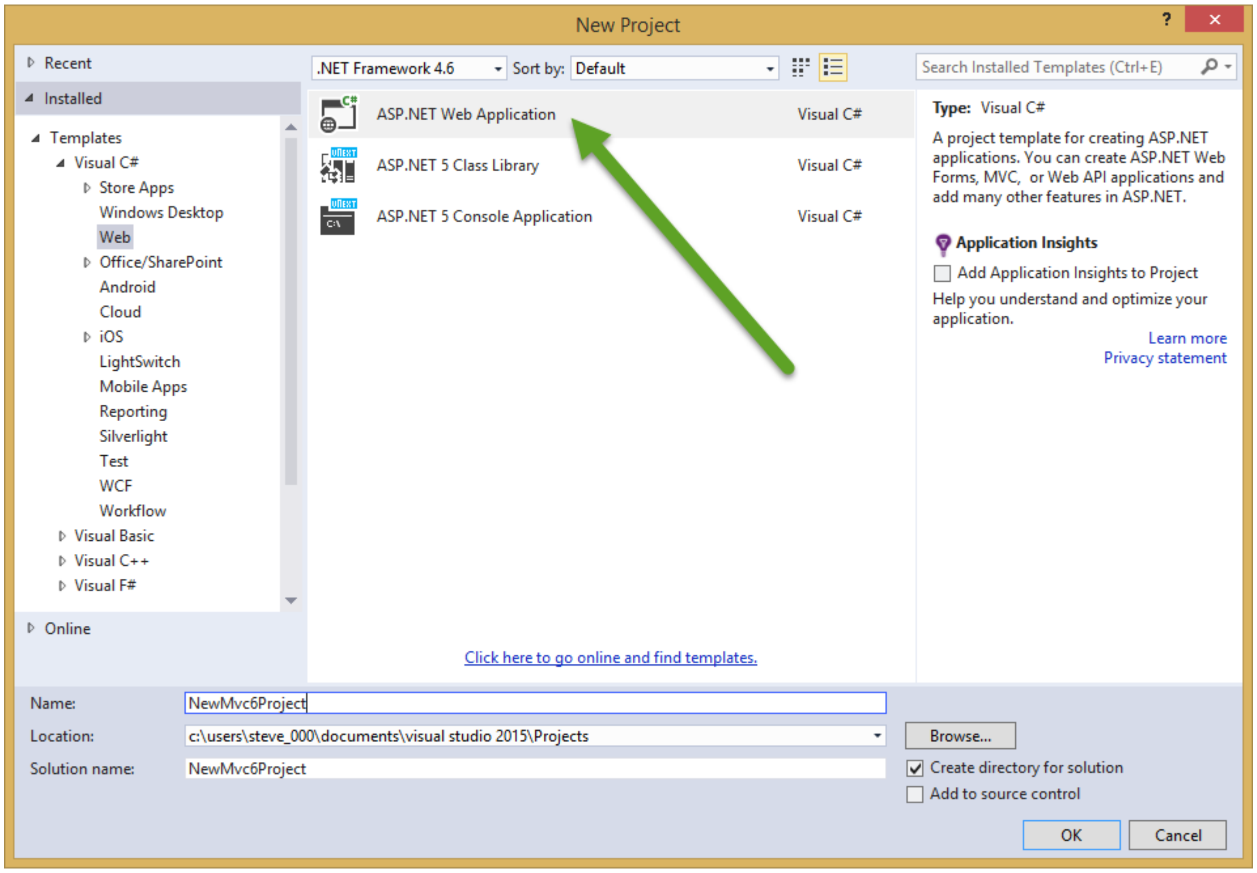
OK Cancel

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.



Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

[ ] Add Application Insights to Project

Help you understand and optimize your application.

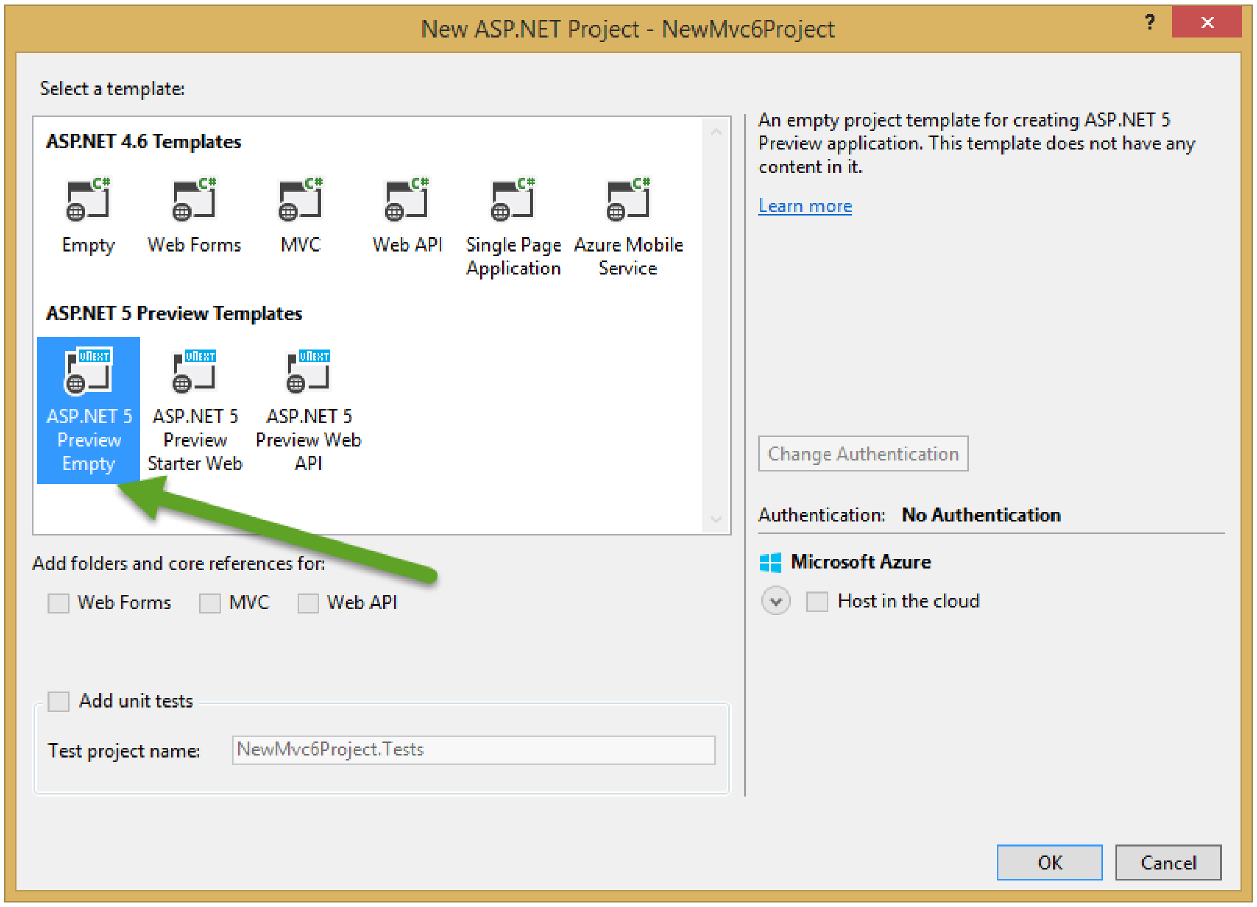
Name: NewMvc6Project

Location: c:\users\steve_000\documents\visual studio 2015\Projects

Solution name: NewMvc6Project

[✓] Create directory for solution

[ ] Add to source control



An empty project template for creating ASP.NET 5 Preview application. This template does not have any content in it.

Authentication: **No Authentication**

Test project name: NewMvc6Project.Tests

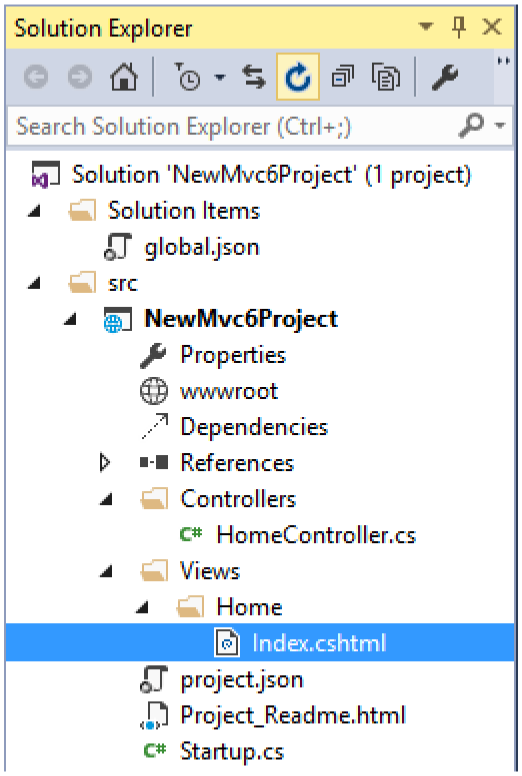
This migration will start from an empty template. If you're already familiar with ASP.NET 5 and its starter templates and there are features in a starter template you would like to take advantage of, you may wish to start from another template. The next step is to configure the site to use MVC. This 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?}");
        });
}
```

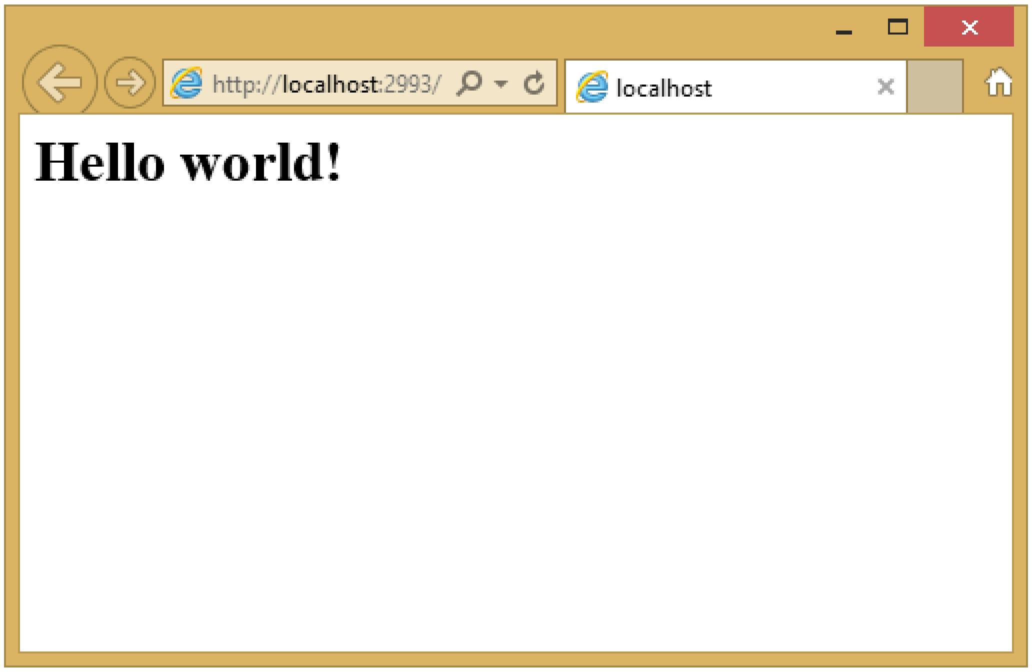
At this point we are ready to create a simple Controller and View. Add a Controllers folder and a Views folder to the project. Add an MVC Controller called HomeController.cs class to the Controllers folder and a Home folder in the Views folder. Finally, add an Index.cshtml MVC View Page to the Views/Home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



Modify Index.cshtml to show a welcome message:

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

Run the application - you should see Hello World output in your browser.

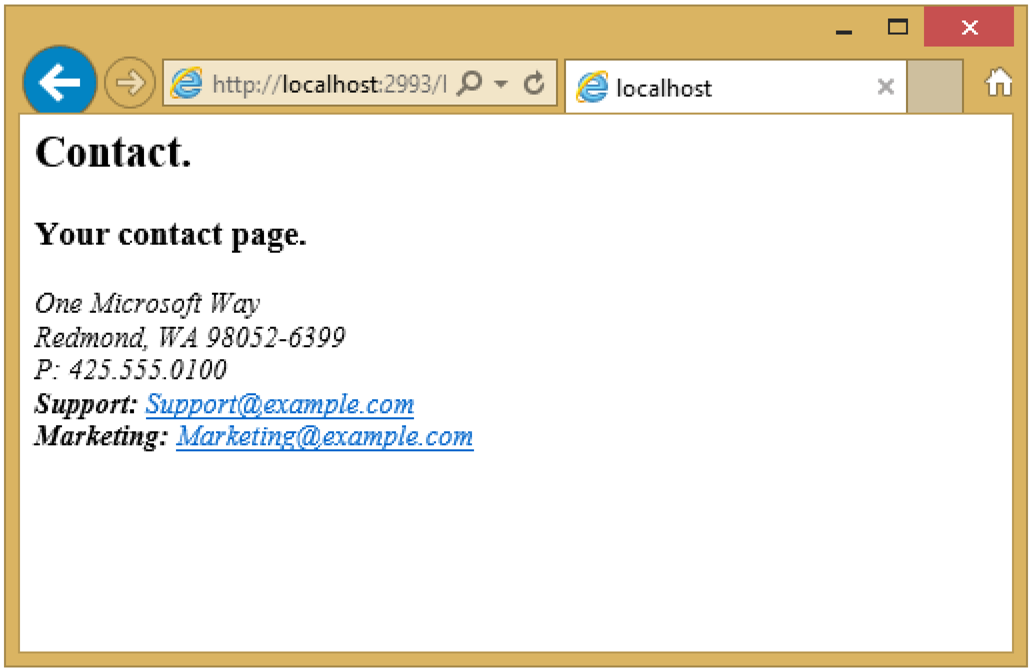


8.1.3 Migrate Basic Controllers, Views, and Static Content

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):



Contact.

Your contact page.

One Microsoft Way
Redmond, WA 98052-6399
P: 425.555.0100
Support: Support@example.com
Marketing: Marketing@example.com

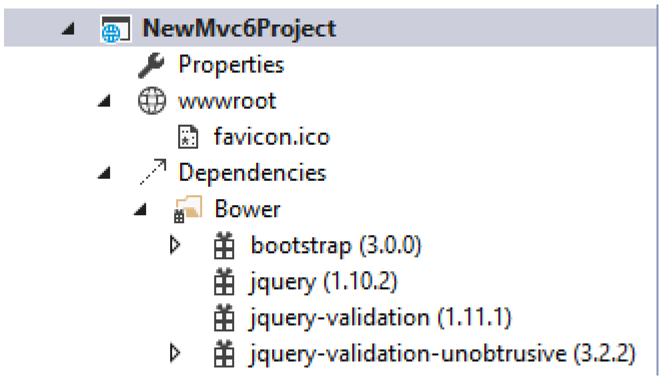
In MVC 5 and previous versions of ASP.NET, static content was hosted from the root of the web project, and was intermixed with server-side files. In MVC 6, all static content files are hosted from the /wwwroot folder, so we will need to adjust where we are storing our static content files. For instance, 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:



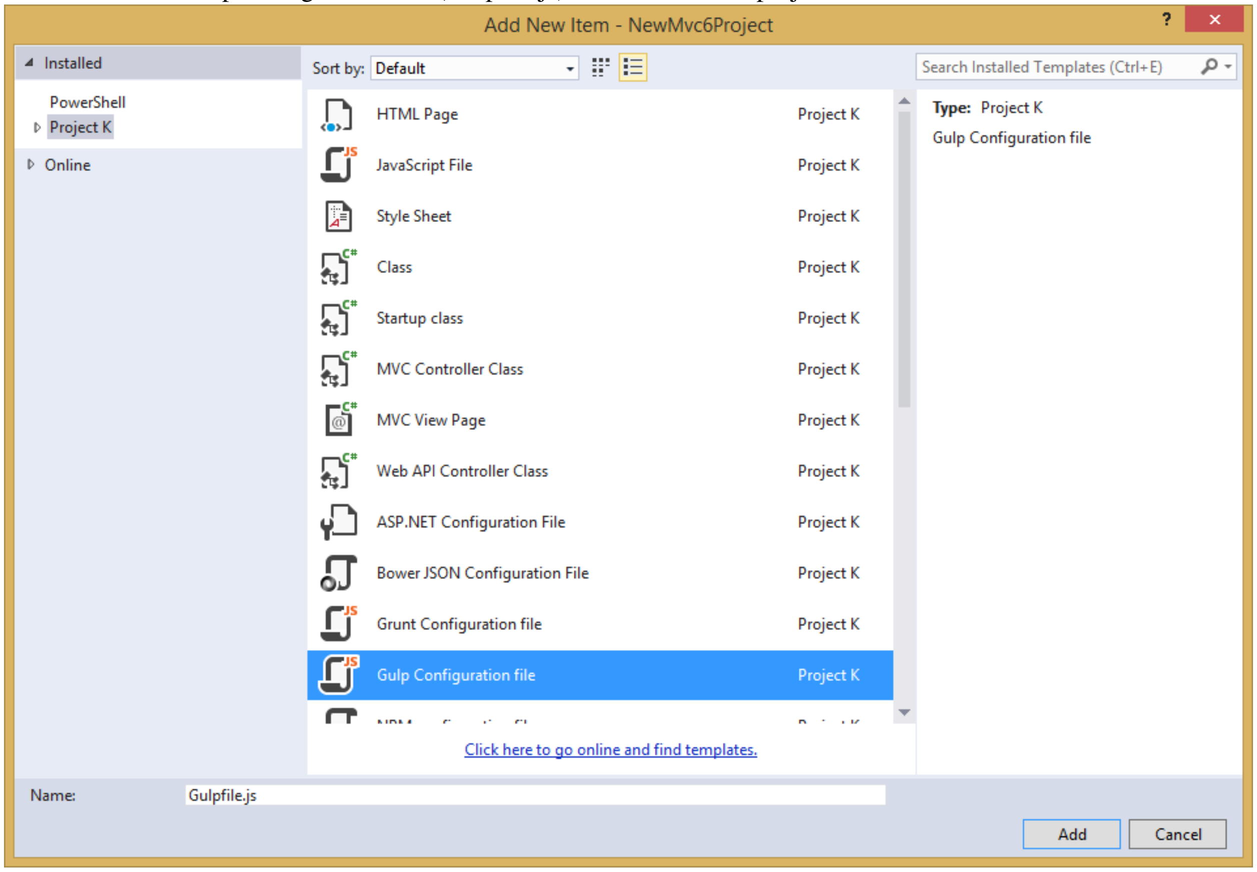
```
NewMvc6Project
    Properties
    wwwroot
        favicon.ico
    Dependencies
        Bower
            bootstrap (3.0.0)
            jquery (1.10.2)
            jquery-validation (1.11.1)
            jquery-validation-unobtrusive (3.2.2)
```

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 *guld-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:

```
{
    "version": "1.0.0",
    "name": "NewMvc6Project",
    "private": true,
    "devDependencies": {
        "gulp": "3.8.11",
        "rimraf": "2.3.2",
        "gulp-concat": "2.5.2"
    }
}
```

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.


Add New Item - NewMvc6Project
Installed
PowerShell
Project K
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
HTML Page Project K
JavaScript File Project K
Style Sheet Project K
Class Project K
Startup class Project K
MVC Controller Class Project K
MVC View Page Project K
Web API Controller Class Project K
ASP.NET Configuration File Project K
Bower JSON Configuration File Project K
Grunt Configuration file Project K
Gulp Configuration file Project K
Click here to go online and find templates.
Add
Cancel

Type: Project K

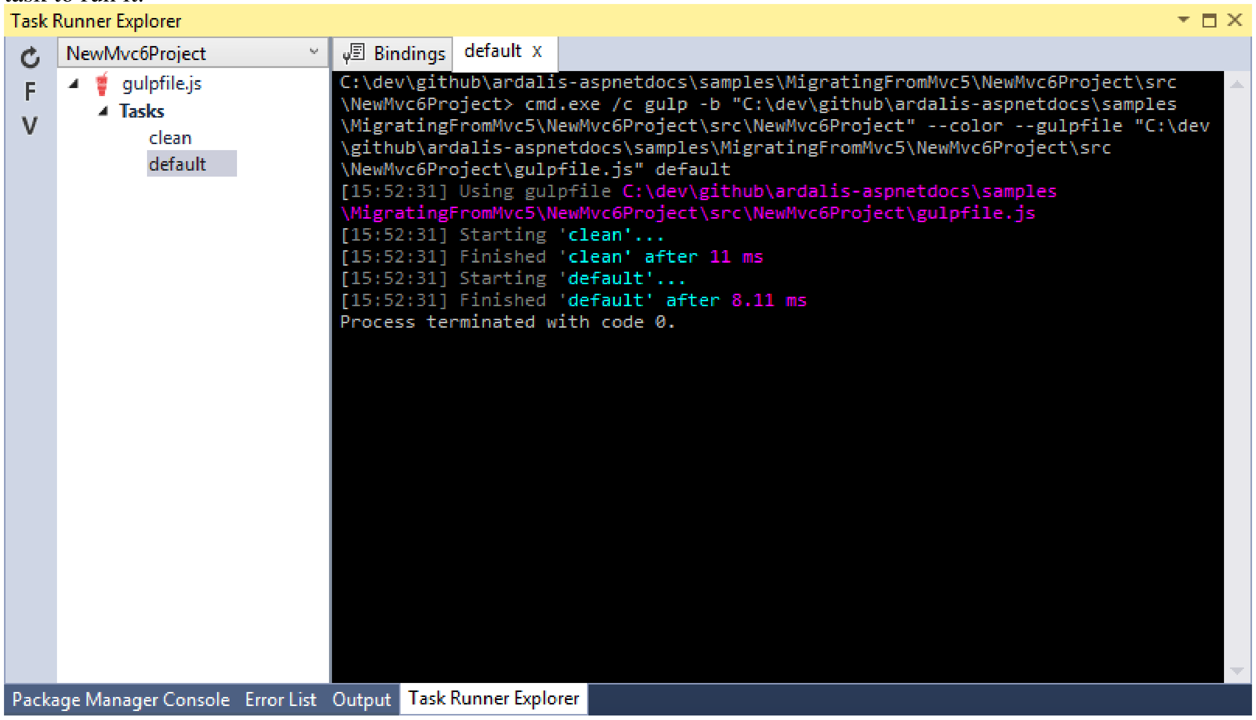
Gulp Configuration file

Name: Gulpfile.js

We need to configure Gulp to use Bower, and then register tasks associated with this configuration. Modify Gulpfile.js to match this file:

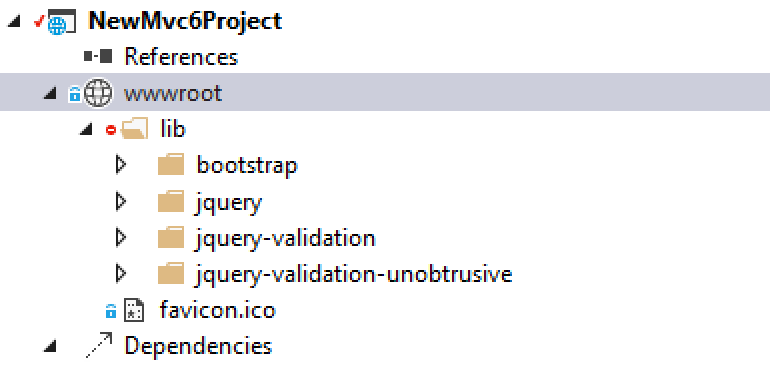
```
var gulp = require('gulp');
var rimraf = require('rimraf');

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.



```
C:\dev\github\ardalis-aspnetdocs\samples\MigratingFromMvc5\NewMvc6Project\src
\NewMvc6Project> cmd.exe /c gulp -b "C:\dev\github\ardalis-aspnetdocs\samples
\MigratingFromMvc5\NewMvc6Project\src\NewMvc6Project" --color --gulpfile "C:\dev
\github\ardalis-aspnetdocs\samples\MigratingFromMvc5\NewMvc6Project\src
\NewMvc6Project\gulpfile.js" default
[15:52:31] Using gulpfile C:\dev\github\ardalis-aspnetdocs\samples
\MigratingFromMvc5\NewMvc6Project\src\NewMvc6Project\gulpfile.js
[15:52:31] Starting 'clean'...
[15:52:31] Finished 'clean' after 11 ms
[15:52:31] Starting 'default'...
[15:52:31] Finished 'default' after 8.11 ms
Process terminated with code 0.
```

The output should show that the process completed without errors, and you should see that it copied some packages to the wwwrootlib folder. Open the wwwrootlib 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)

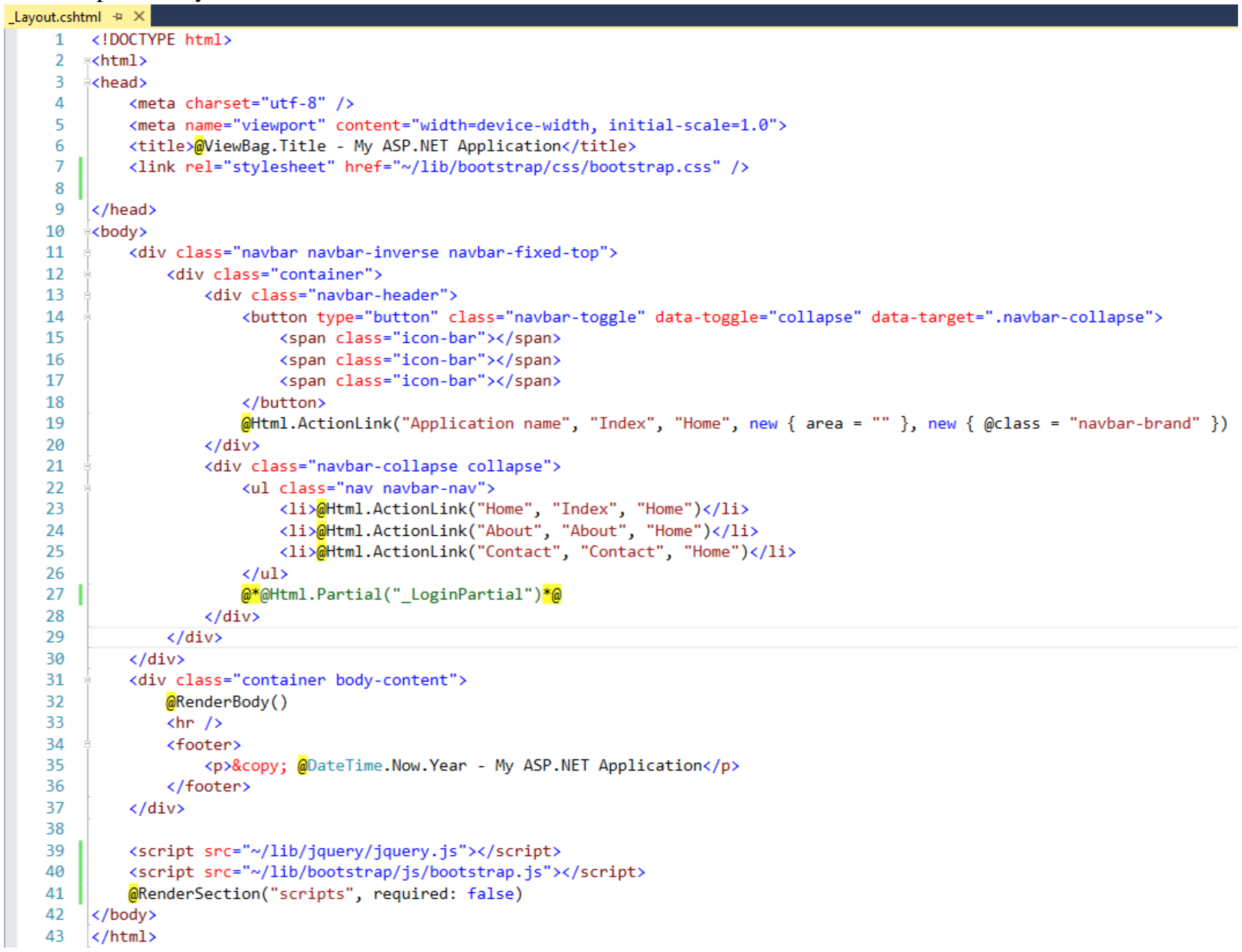
The CSS link to use:

```
<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:



```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />

</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @*@Html.Partial("_LoginPartial")*@
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="~/lib/jquery/jquery.js"></script>
    <script src="~/lib/bootstrap/js/bootstrap.js"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

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

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*)

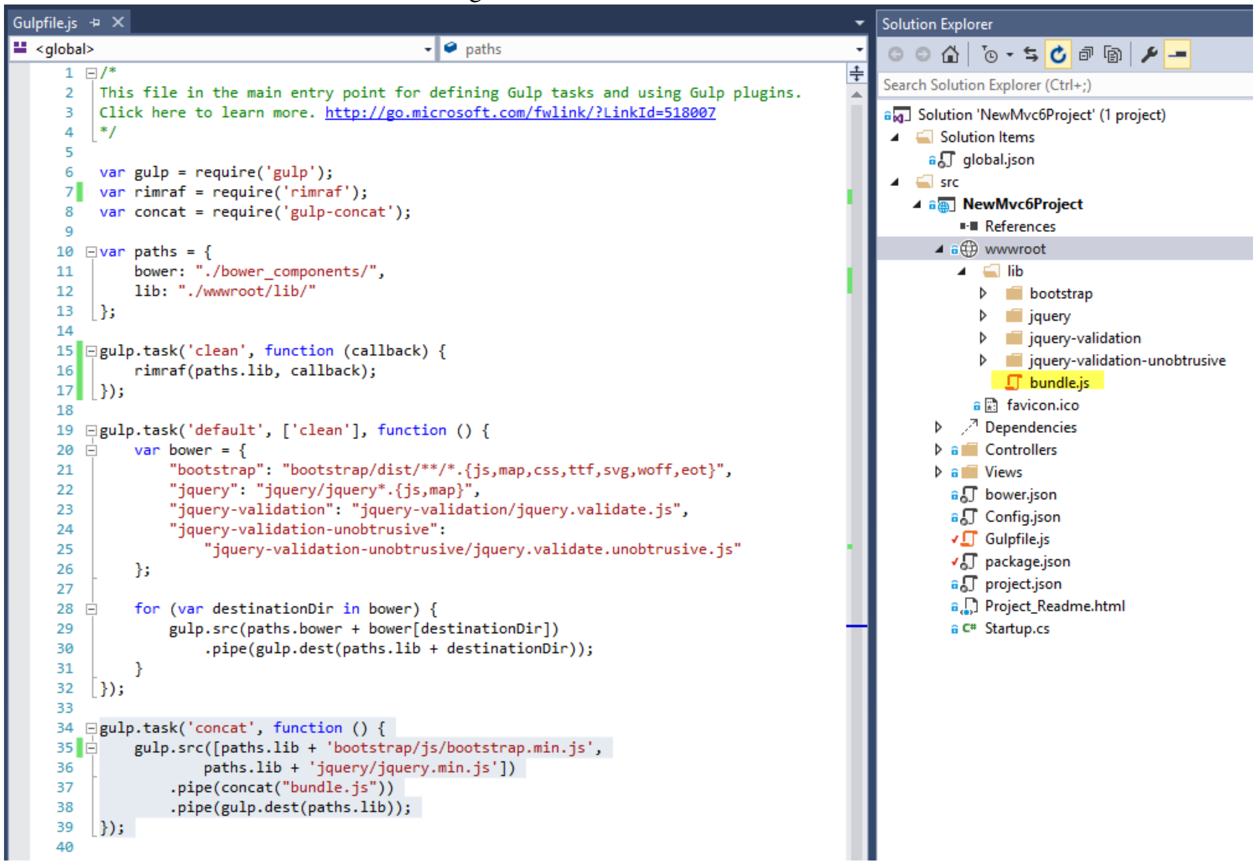
To simply bundle the jQuery and bootstrap scripts together into a single, minified, file, we can use the gulp-concat task. First, update package.json to require gulp-concat in “devDependencies”:

```
"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:

```
/*
This file in the main entry point for defining Gulp tasks and using Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
*/

var gulp = require('gulp');
var rimraf = require('rimraf');
var concat = require('gulp-concat');

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));
    }
});

gulp.task('concat', function () {
    gulp.src([paths.lib + 'bootstrap/js/bootstrap.min.js',
            paths.lib + 'jquery/jquery.min.js'])
        .pipe(concat("bundle.js"))
        .pipe(gulp.dest(paths.lib));
});
```

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:



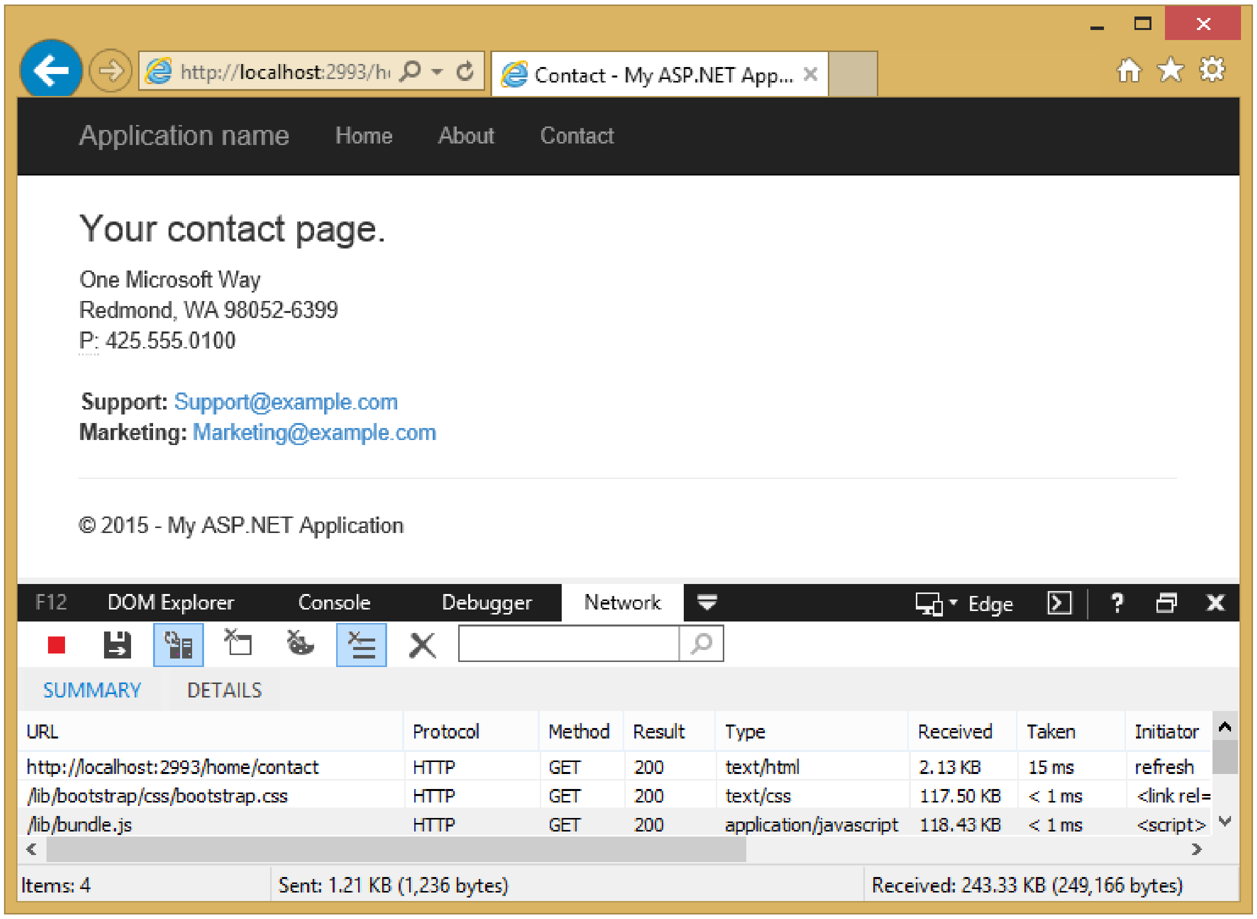
```
/*
This file in the main entry point for defining Gulp tasks and using Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
*/

var gulp = require('gulp');
var rimraf = require('rimraf');
var concat = require('gulp-concat');

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));
    }
});

gulp.task('concat', function () {
    gulp.src([paths.lib + 'bootstrap/js/bootstrap.min.js',
            paths.lib + 'jquery/jquery.min.js'])
        .pipe(concat("bundle.js"))
        .pipe(gulp.dest(paths.lib));
});
```

```
Solution 'NewMvc6Project' (1 project)
  Solution Items
    global.json
  src
    NewMvc6Project
      References
      wwwroot
        lib
          bootstrap
          jquery
          jquery-validation
          jquery-validation-unobtrusive
          bundle.js
        favicon.ico
      Dependencies
      Controllers
      Views
      bower.json
      Config.json
      Gulpfile.js
      package.json
      project.json
      Project_Readme.html
      Startup.cs
```

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:



Your contact page.

One Microsoft Way
Redmond, WA 98052-6399
P: 425.555.0100

Support: Support@example.com
Marketing: Marketing@example.com

© 2015 - My ASP.NET Application

URL	Protocol	Method	Result	Type	Received	Taken	Initiator
http://localhost:2993/home/contact	HTTP	GET	200	text/html	2.13 KB	15 ms	refresh
/lib/bootstrap/css/bootstrap.css	HTTP	GET	200	text/css	117.50 KB	< 1 ms	<link rel=
/lib/bundle.js	HTTP	GET	200	application/javascript	118.43 KB	< 1 ms	<script>

Items: 4 Sent: 1.21 KB (1,236 bytes) Received: 243.33 KB (249,166 bytes)

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:

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

public Startup(IHostingEnvironment env)
{
        // Setup configuration sources.
        Configuration = new Configuration()
                .AddJsonFile("config.json")
                .AddEnvironmentVariables();
}
```

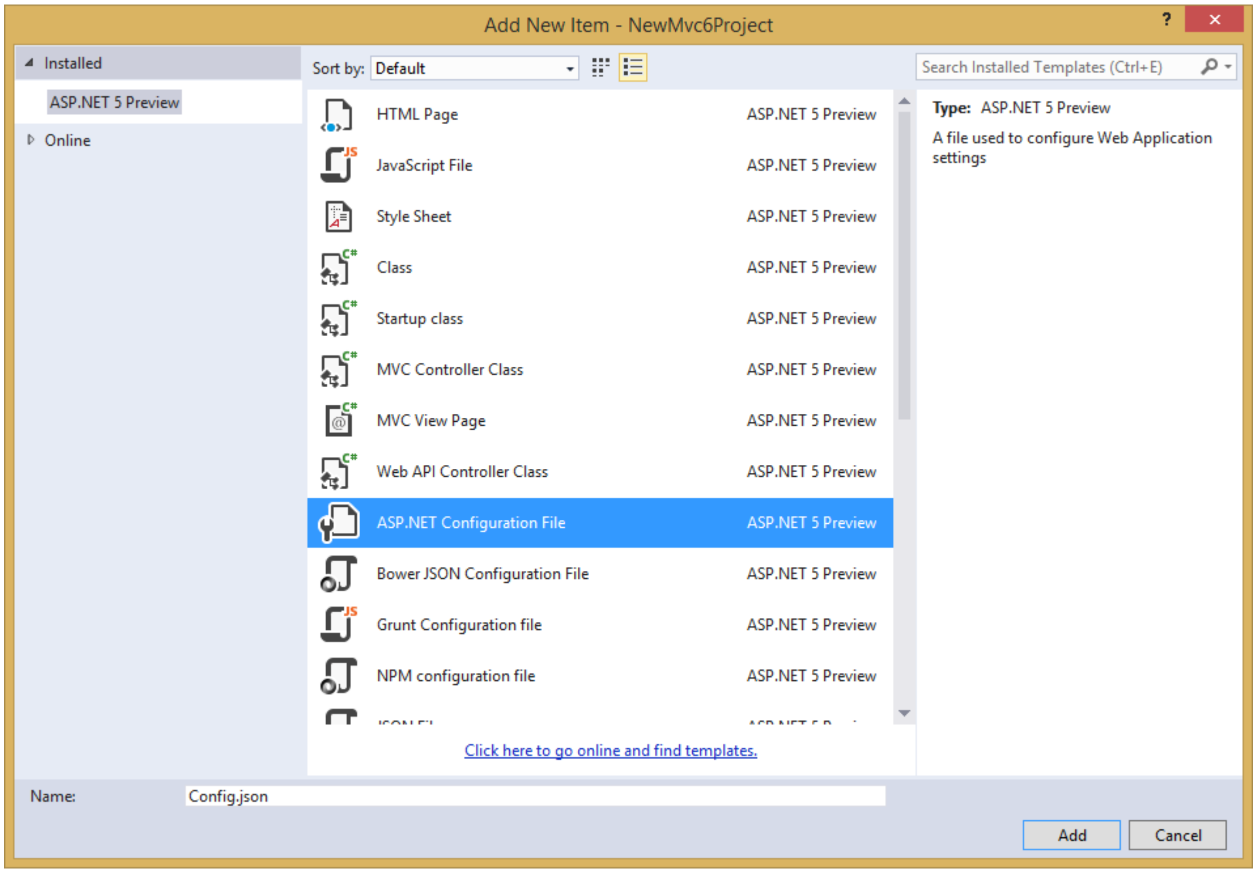
Note that at this point the Startup.cs file will not compile, as we still need to add some using statements and pull in some dependencies. Add the following two using statements:

```
using Microsoft.Framework.ConfigurationModel;
using Microsoft.AspNet.Hosting;
```

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

```
{
        "webroot": "wwwroot",
        "version": "1.0.0-*",
        "dependencies": {
                "Microsoft.AspNet.Server.IIS": "1.0.0-beta3",
                "Microsoft.AspNet.Mvc": "6.0.0-beta3",
                "Microsoft.Framework.ConfigurationModel.Json": "1.0.0-beta3"
        },
        ...
}
```

Finally, add a config.json file to the root of the project.



Sort by: Default

Type: ASP.NET 5 Preview

A file used to configure Web Application settings

Name: Config.json

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 <connectionStrings> 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)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

This class configures attribute routing, although it's not actually being used in the project, 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:

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

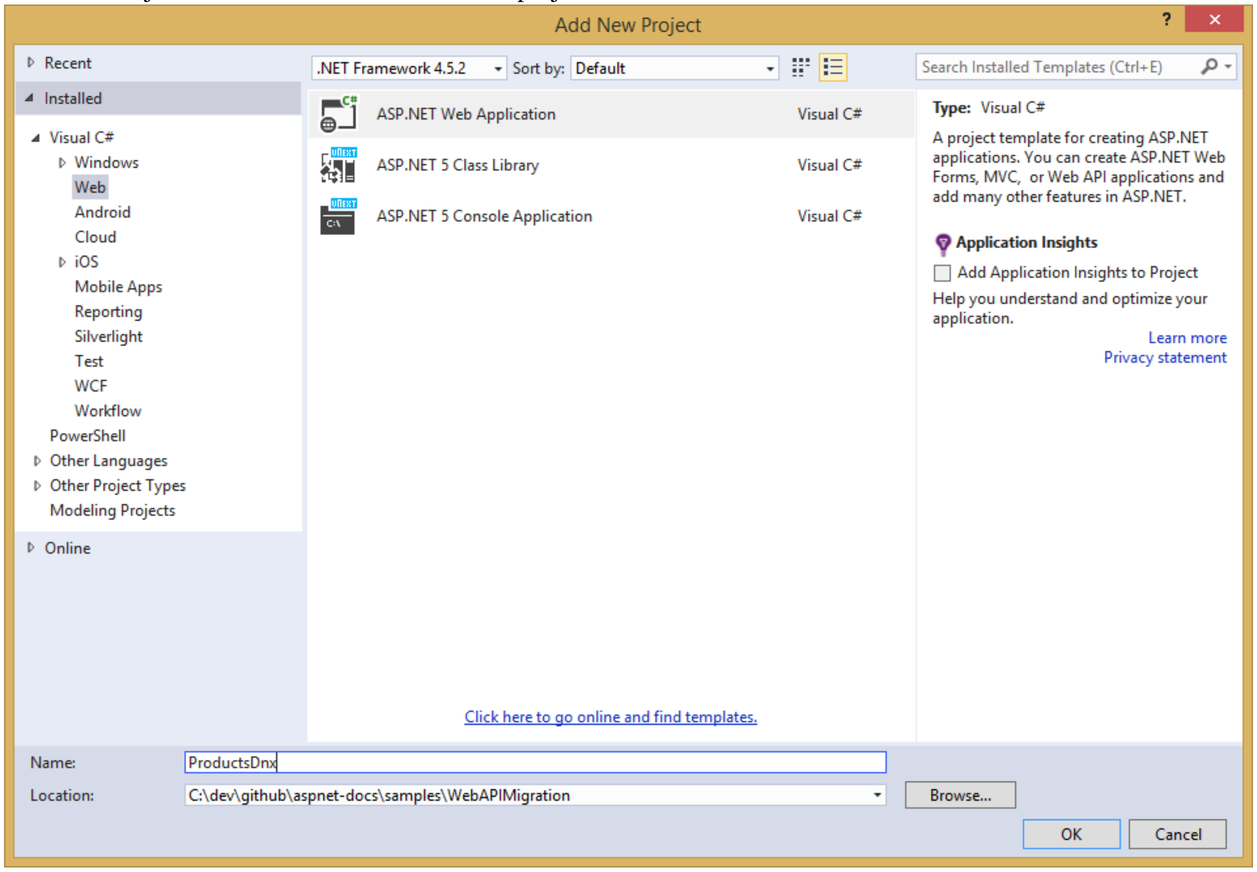
Finally, the model, Product, used by the ProductsApp, is a simple class:

```
namespace ProductsApp.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

Now that we have a simple project from which to start, we can demonstrate how to migrate this Web API 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'.



Type: Visual C#

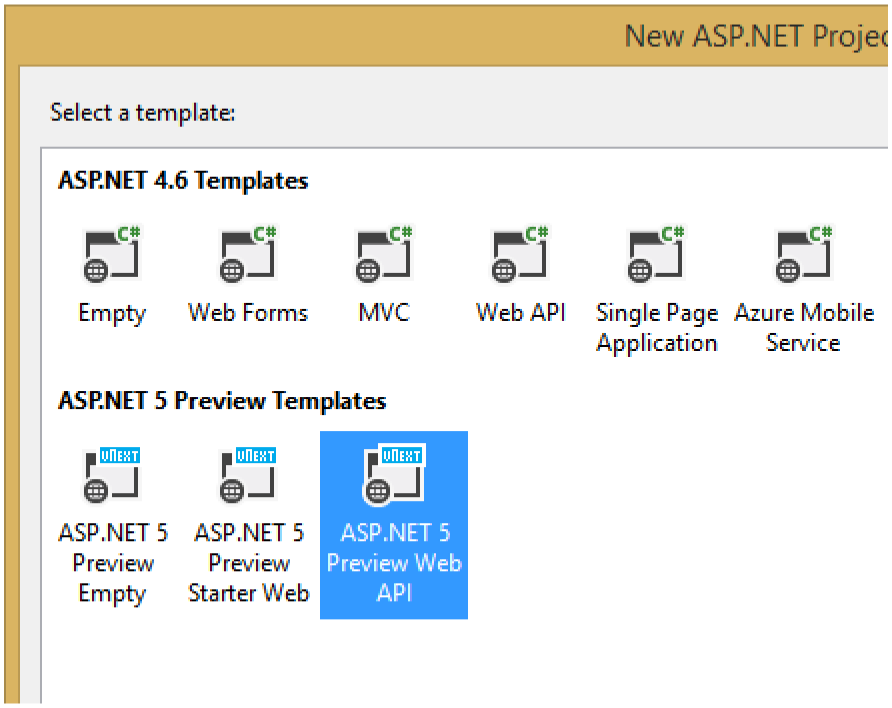
A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Help you understand and optimize your application.

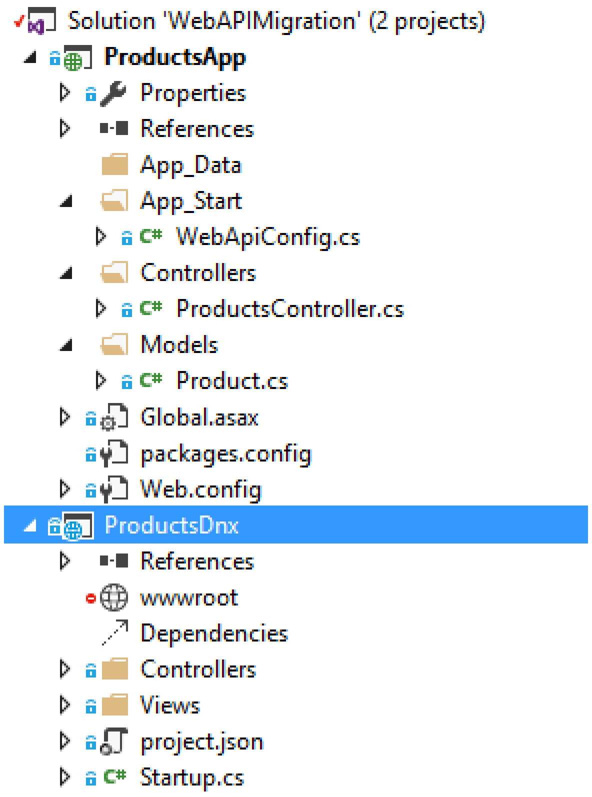
Name: ProductsDnx

Location: C:\dev\github\aspnet-docs\samples\WebAPIMigration

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).

```
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:

```
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()
        {
            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:

```
using Microsoft.AspNet.Mvc;
using ProductsDnx.Models;
using System.Collections.Generic;
using System.Linq;

namespace ProductsDnx.Controllers
{
    [Route("api/[controller]")]
    public class ProductsController : Controller
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        // /api/products
        [HttpGet]
        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        // /api/products/1
        [HttpGet("{id}")]
        public IActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return HttpNotFound();
            }
            return new ObjectResult(product);
        }
        }
}
```

You should now be able to run the migrated project and browse to /api/products, and you should see the full list of 3 products. Browse to /api/products/1 and you should see the first product.

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",
            </li>
            <li><a href="javascript:document.getElementById('logoutForm').submit()">Log off</a></li>
        </ul>
    }
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li>@Html.ActionLink("Register", "Register", "Account", routeValues: null, htmlAttributes: ne
        <li>@Html.ActionLink("Log in", "Login", "Account", routeValues: null, htmlAttributes: new {
    </ul>
}
```

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

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.

CHAPTER 9

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